A SUMMER INTERNSHIP REPORT

ON

ELECTRONIC CIRCUIT ANALYSIS USING PSPICE& XILINX VIVADO

ALARM CLOCK USING VERILOG

SUBMITTED FOR THE PARTIAL FULFILMENT
OF

B. TECH.

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

SUBMITTED BY:

Anuj Agarwal
2100270310044

5th Semester - Third Year
Section- ECE-1

SUBMITTED TO:

Ms. Uma Sharma
(ASST. PROF.)
ECE DEPT.

Ajay Kumar Garg Engineering College, Ghaziabad
27th 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 **Dr. Pankaj Goel**, **Asst. Prof. Uma Sharma and Asst. Prof. Prashant Mani Tripathi** 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.

Anuj Agarwal
2100270310044
5thSem-Third Year
Section- ECE-1

TABLE OF CONTENTS

Chapter 1. Introduction to Verilog HDL & Xilinx Vivado Design Suite 4-14

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

Chapter 2. Introduction to PSPICE 15-18

2.1 Introduction

2.2 Key features of PSPICE Simulator

2.3 Xilinx Vivado Design flow

Chapter 3. Mini Project Description 19-38

3.1 Introduction to Alarm Clock (Verilog/VHDL/PSPICE)

3.2 Objectives

3.3 Description of Project

3.3.1 Block Diagram/ Flow Chart

3.3.2 Algorithm Description

3.3.3 Simulation and Synthesis Results

References 39

CHAPTER- 1

INTRODUCTION TO VERILOG HDL

1.1 Introduction:

1.1.1 Introduction to Verilog HDL:

What is Verilog HDL?

Verilog Hardware Description Language (HDL) is a powerful programming language used in the design, verification, and implementation of digital circuits. It allows engineers to describe the behavior and structure of electronic systems, making it an essential tool in digital design.

Key Features:

1. **Conciseness:** Verilog enables the concise representation of complex digital circuits, making it easier to design and understand the functionality of digital systems.
2. **Hierarchy:** Verilog supports a hierarchical design approach, allowing designers to break down complex systems into modular components. This hierarchical structure enhances reusability and simplifies system management.
3. **Simulation:** Verilog facilitates simulation, enabling designers to model and test the behavior of a digital circuit before actual implementation. This helps catch errors and refine designs early in the development process.
4. **Parallelism:** Verilog's parallel execution model reflects the parallel nature of hardware, making it a natural fit for hardware description.
5. **Synthesis:** Verilog code can be synthesized into hardware, transforming the high-level description into a netlist of gates and flip-flops that can be implemented on an FPGA or ASIC.

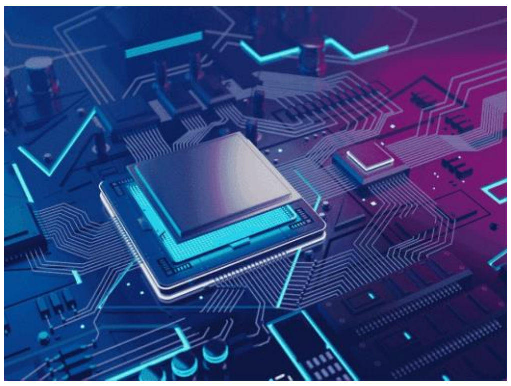


Fig. 1.1 Verilog HDL

1.1.2 Introduction to Xilinx Vivado Design Suite:

What is Vivado Design Suite?

Xilinx Vivado Design Suite is an integrated development environment (IDE) used for the design, verification, and implementation of digital systems on Xilinx FPGAs (Field-Programmable Gate Arrays). It provides a comprehensive set of tools for all stages of the FPGA development process.

Key Components:

1. **Vivado IDE:** The central interface for project management, design entry, synthesis, implementation, and debugging. It streamlines the design flow by integrating various tools into a unified environment.
2. **Vivado HLS (High-Level Synthesis):** Allows designers to describe and optimize algorithms at a higher level of abstraction using C, C++, or SystemC. HLS then transforms these high-level descriptions into RTL (Register Transfer Level) code.
3. **IP Integrator:** Simplifies the integration of predefined Intellectual Property (IP) blocks into a design, reducing development time and ensuring consistency.
4. **Vivado Simulator:** Facilitates functional simulation of the design, allowing engineers to verify the correctness of their code before synthesis and implementation.
5. **Vivado High-Level Synthesis:** Converts C, C++, or SystemC code into RTL, providing an efficient way to design and optimize complex algorithms.
6. **Implementation Tools:** These tools perform place and route, generating a configuration bitstream that can be loaded onto the target FPGA.
7. **Conclusion:** Verilog HDL and Xilinx Vivado Design Suite are integral components in the digital design ecosystem. Verilog provides a robust language for describing digital circuits, while Vivado offers a comprehensive suite of tools for designing and implementing FPGA-based systems efficiently. Together, they enable engineers to create complex digital designs with a balance of abstraction and control.

1.2 Keywords and Identifiers:

1.2.1.1 Keywords in Verilog HDL:

module: Defines a module, which is a collection of hardware components.

input/output: Specifies input and output ports of a module.

reg/wire: Declares registers and wires, representing storage elements and connections.

always: Defines a block of code that is executed continuously.

if/else: Conditional statements for making decisions in the code.

case: Used for a multi-way branch based on the value of an expression.

assign: Assigns a value to a wire or register.

always_ff/always_comb: Used for describing sequential and combinational logic, respectively.

initial: Specifies a block of code that is executed only once at the beginning of simulation.

for/while: Loops for repetitive execution of code.

1.2.1.2 Identifiers in Verilog HDL:

1. **Module Names:** Names given to modules in Verilog, typically starting with a letter or an underscore.
    - **module my_module;**
2. **Signal Names:** Identifiers for wires and registers used to represent data flow.
    - **reg [7:0] data_in;**
    - **wire [7:0] result;**
3. **Instance Names:** Names given to instances of modules when instantiated within another module.
    - **my_module instance1 (.input_A(a), .input_B(b), .output_Z(z));**
4. **Variable Names:** Identifiers for variables used within procedural blocks like always or initial.
    - **int count = 0;**
5. **Port Names:** Identifiers used for module ports in the module definition.
    - **module my_module (input a, output b);**

1.2.2.1 Keywords in Xilinx Vivado Design Suite:

1. **Vivado IDE: Integrated Development Environment for FPGA design.**
2. **IP Integrator:** Tool for integrating Intellectual Property (IP) blocks into a design.
3. **Vivado HLS:** High-Level Synthesis tool for converting C, C++, or SystemC code into RTL.

4. **Simulation:** Process of testing and verifying the functionality of a design before implementation.
5. **Bitstream:** Configuration file generated for programming the FPGA.
6. **Place and Route:** The process of mapping a design onto the physical resources of an FPGA.
7. **Constraint:** Rules and requirements for the synthesis and implementation process.
8. **Block Design:** Hierarchical representation of a design using IP blocks in the Vivado IDE.
9. **Synthesis:** Process of converting high-level RTL code into a netlist of logic gates.
10. **Implementation:** Process of turning a synthesized design into a configuration bitstream.

1.2.2.2 Identifiers in Xilinx Vivado Design Suite:

Project Names: Names given to Vivado projects.

create_project my_project

Block Design Names: Identifiers for hierarchical designs created using IP blocks.

design_1

Constraint File Names: Identifiers for files specifying design constraints.

my_constraints.xdc

Instance Names: Identifiers for instances of Intellectual Property blocks in a design.

my_ip_0

Bitstream Names: Identifiers for configuration bitstream files generated for programming the FPGA.

my_design.bit

1.3 Data Types and Modelling Styles:

1.3.1 Data Types in Verilog HDL:

1. **Wire:**
    - Represents a continuous assignment or a net in the hardware.
    - Used for connecting different parts of the circuit.
    - Example: wire [7:0] data_wire;

2. **Reg:**
    - Represents a register or a storage element.
    - Can be used to store and manipulate data.

- Example: reg [3:0] counter;

3. **Integer:**
   - Used for integer data types in procedural blocks.
   - Commonly used for loop counters and indexing arrays.
   - Example: integer i;
4. **Real:**
   - Represents real numbers in procedural blocks.
   - Example: real voltage;
5. **Time:**
   - Represents time values in simulation.
   - Used for specifying delays and timing information.
   - Example: time period = 10 ns;
6. **Parameter:**
   - Allows the definition of constants that can be changed at compile-time.
   - Example: parameter WIDTH = 8;
7. **Enum:**
   - Represents enumerated data types with symbolic names.
   - Used to improve code readability.
   - Example: typedef enum {IDLE, ACTIVE, DONE} state;
8. **Array:**
   - Used to represent arrays of data.
   - Allows the modeling of multi-dimensional data structures.
   - Example: reg [7:0] memory [0:255];

1.3.2 Modeling Styles in Verilog HDL:

1. **Behavioral Modeling:**
   - Describes the functionality and behavior of a design without specifying the implementation details.
   - Uses high-level constructs such as always blocks to model the desired behavior.
   - Focuses on what the circuit should do rather than how it should be implemented.
2. **RTL (Register-Transfer Level) Modeling:**
   - Describes how data moves between registers and the operations that occur between them.
   - Uses constructs like always_ff for clocked logic and always_comb for combinational logic.
   - Represents the flow of data at the register-transfer level.

3. **Dataflow Modeling:**
    - Describes the flow of data through the circuit.
    - Uses concurrent assignments to represent combinational logic.
    - Well-suited for expressing mathematical relationships between signals.
4. **Structural Modeling:**
    - Describes a design as a hierarchy of interconnected modules.
    - Uses instantiation and connection of lower-level modules to represent the entire system.
    - Allows for a top-down design approach, breaking down complex systems into manageable modules.
5. **Gate-Level Modeling:**
    - Describes a design using primitive logic gates.
    - Specifies the exact gates and connections used in the design.
    - Rarely used for large-scale designs due to its low abstraction level.
6. **FSM (Finite State Machine) Modeling:**
    - Represents designs that have different states and transitions between states.
    - Uses case or if-else statements to model state transitions.
    - Commonly used for control-intensive designs.
7. **Testbench Modeling:**
    - Describes the environment used for testing and verifying the functionality of a design.
    - Includes stimulus generation, checking expected results, and reporting.

These modeling styles provide flexibility in expressing different aspects of digital designs, allowing designers to choose the most suitable approach based on the design requirements and complexity.

1.4 Verilog programming for various combinational and Sequential circuits:

Combinational Circuits:

1. **2-to-1 Multiplexer:**

Boolean expression: $Y = \bar{S}.I0 + S.I2$

```
module mux_2x1_gl(
input I0,I1,S,
output Y);
wire sb,a,b;

not(sb,S);
and(a,sb,I0);
and(b,S,I1);
or(Y,a,b);

endmodule
```

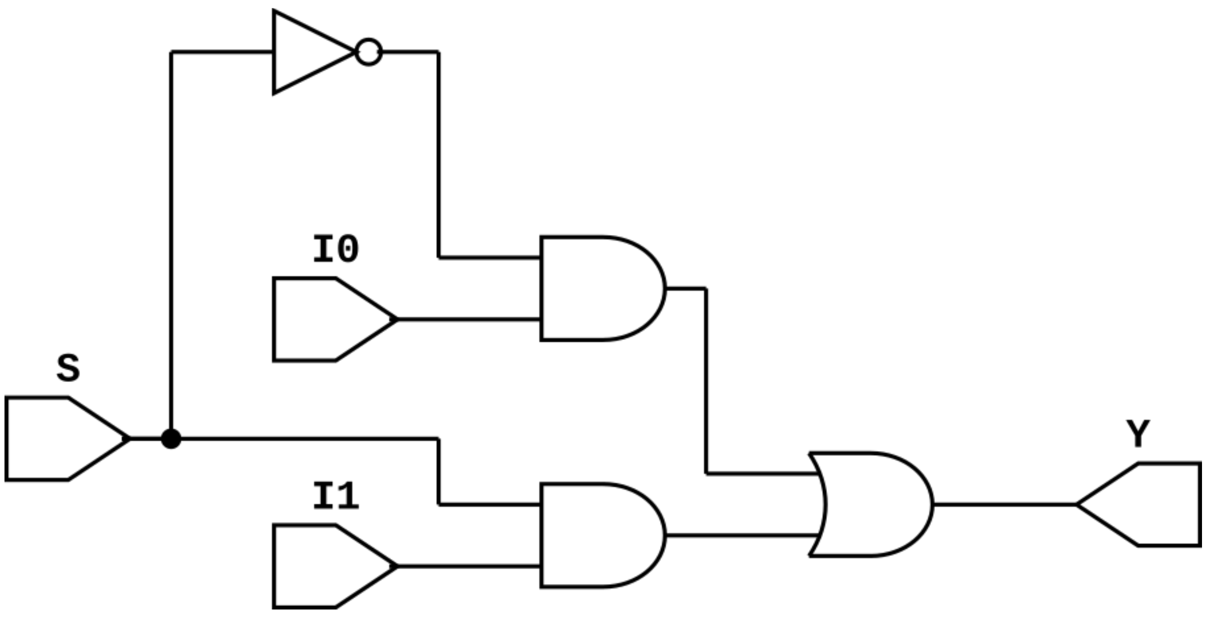


Fig. 1.2 Flow Chart of above program

Table 1. Truth Table of 2x1 MUX

S	Y
0	I0
1	I1

2. 4-to-1 Multiplexer:

```
module m41(out, a, b, c, d, s0, s1);
```

```
output out;

input a, b, c, d, s0, s1;

wire sobar, s1bar, T1, T2, T3, T4;

not (s0bar, s0), (s1bar, s1);

and (T1, a, s0bar, s1bar), (T2, b, s0bar, s1),(T3, c, s0, s1bar), (T4, d, s0, s1);

or(out, T1, T2, T3, T4);

endmodule
```

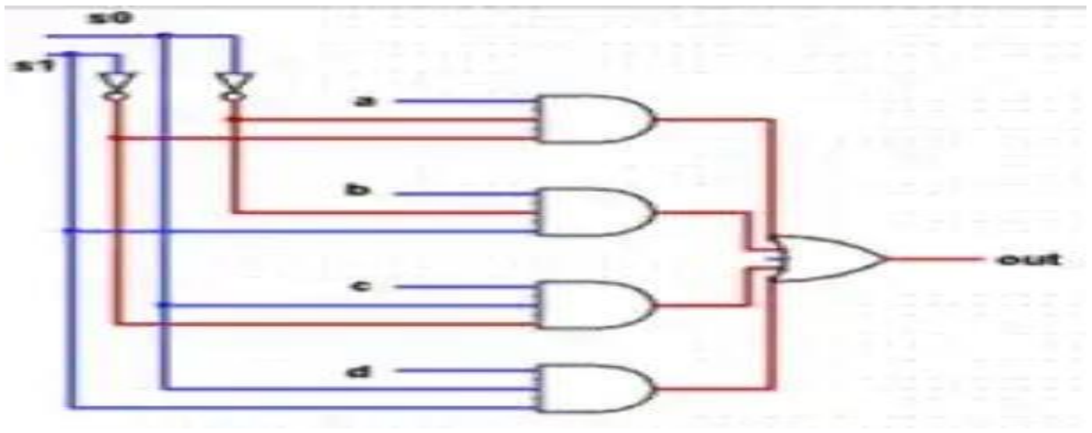


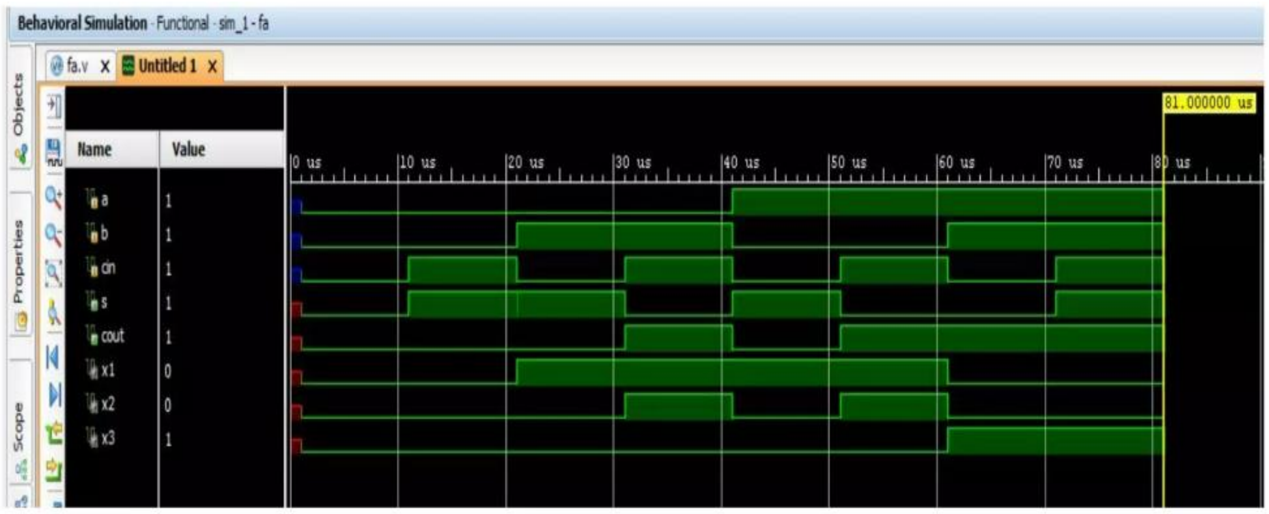
Fig. 1.3 Logic Diagram of 4x1 MUX

Table 2. Truth Table of 4x1 MUX

INPUT	S1	S0	out
a	0	0	a
b	0	1	b
c	1	0	c
d	1	1	d

3. Full Adder:

```
module fa(
        input a,
        input b,
        input cin,
        output s,
        output cout);
        wire x1,x2,x3;
        xor(x1,a,b);
        and(x3,a,b);
        xor(s,x1,cin);
        and(x2,x1,cin);
        or(cout,x2,x3);
endmodule
```



Name	Value
a	1
b	1
cin	1
s	1
cout	1
x1	0
x2	0
x3	1

Fig. 1.4 Simulation of above program

Table 3. Truth Table of Full Adder

a	***b***	***cin***	***s***	***cout***
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

Sequential Circuits:

1. D Flip-Flop:

```
module nand_gate(c,a,b);
input a,b;
output c;
assign c = ~(a&b);
endmodule

module not_gate(f,e);
input e;
output f;
assign f= ~e;
endmodule

module d_ff_struct(q,qbar,d,clk);
input d,clk;
output q, qbar;
not_gate not1(dbar,d);
nand_gate nand1(x,clk,d);
nand_gate nand2(y,clk,dbar);
nand_gate nand3(q,qbar,y);
nand_gate nand4(qbar,q,x);
endmodule
```

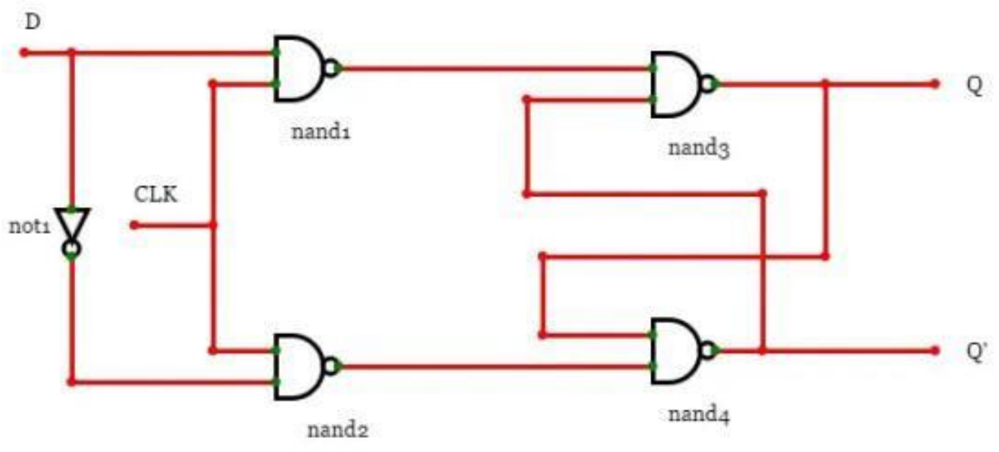


Fig. 1.5 Logic Diagram of D Flip Flop

Table 4. Truth Table of NAND Gate AND D Flip Flop

A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0

D	Q
0	0
1	1

CHAPTER- 2

INTRODUCTION TO PSPICE

2.1 Introduction:

What is PSpice?

PSpice (Personal Simulation Program with Integrated Circuit Emphasis) is a widely used electronic circuit simulation tool developed by Cadence Design Systems. It is designed for the analysis of electrical circuits, enabling engineers and designers to simulate the behavior of circuits before building physical prototypes. PSpice is particularly popular for its versatility and capability to model complex electrical components and systems.

2.2 Key Features and Capabilities:

1. **Circuit Simulation:**
    - PSpice allows users to model and simulate analog and digital circuits, including linear and nonlinear components, as well as passive and active devices.
2. **Behavioral Modeling:**
    - It supports behavioral modeling, allowing designers to use mathematical expressions to describe the behavior of custom components or entire circuits.
3. **Transient Analysis:**
    - PSpice can perform transient analysis to simulate the circuit's behavior over time, showing how signals evolve in response to changes.
4. **AC and DC Analysis:**
    - It provides capabilities for AC and DC analysis, allowing engineers to analyze frequency response, gain, phase shift, and other parameters under different operating conditions.
5. **Monte Carlo Analysis:**
    - Monte Carlo analysis in PSpice helps assess the impact of component tolerances and variations on circuit performance.
6. **Sensitivity Analysis:**
    - Sensitivity analysis helps identify critical components or parameters affecting circuit performance.
7. **Parametric Sweeps:**

- PSpice enables users to perform parametric sweeps, exploring the effects of varying component values on circuit behavior.

8. **Digital Simulation:**
    - It supports the simulation of digital circuits, facilitating the analysis of logic gates, flip-flops, and other digital components.
9. **Integration with Schematics:**
    - PSpice is often used in conjunction with schematic capture tools, providing a user-friendly interface for designing circuits graphically.
10. **Graphical Output:**
    - PSpice generates graphical outputs such as voltage and current waveforms, Bode plots, and other visual representations of circuit behavior.

Applications:

11. **Electronic Circuit Design:**
    - PSpice is widely used in the design and analysis of electronic circuits, from simple amplifier circuits to complex mixed-signal systems.
12. **Prototyping and Validation:**
    - Engineers use PSpice to prototype and validate circuit designs before moving to physical implementation, saving time and resources.
13. **Education:**
    - PSpice is commonly used in educational settings to teach electronics and circuit analysis due to its user-friendly interface and powerful simulation capabilities.
14. **Research and Development:**
    - Researchers leverage PSpice for exploring new circuit topologies, analyzing performance under different conditions, and optimizing designs.

2.3 Xilinx Vivado Design flow:

The Xilinx Vivado Design Suite provides a comprehensive design flow for the development of digital systems on Xilinx FPGAs (Field-Programmable Gate Arrays). Here's an overview of the typical Vivado Design Flow:

1. **Project Creation:**
    - Open Vivado and create a new project using the Project Manager.
    - Specify the project name, location, and select the target FPGA device.

- Define the design sources, which include HDL files (e.g., Verilog, VHDL), constraints, and simulation files.

2. **Design Entry:**
    - Choose a design entry method, either through HDL coding, schematic-based entry, or high-level synthesis (HLS) using Vivado HLS.
    - Write or import the hardware description code (e.g., Verilog) that defines the functionality of the digital system.
3. **Simulation:**
    - Create a testbench and simulate the design using the Vivado Simulator or an external simulator like ModelSim.
    - Verify the functionality of the design and fix any issues identified during simulation.
4. **Synthesis:**
    - Run synthesis to convert the high-level RTL (Register Transfer Level) code into a gate-level netlist.
    - This step also involves optimizing the design for the target FPGA architecture.
5. **Implementation:**
    - Floorplan the design, which involves placing and routing the logic elements on the FPGA.
    - Implement the design by running the Vivado Implementation tools. This step includes:
    - Place and Route (P&R): Placing the logic elements and routing the connections.
    - Bitstream Generation: Creating a configuration bitstream file that can be loaded onto the FPGA.
6. **Timing Analysis:**
    - Perform static timing analysis to ensure that the design meets timing constraints.
    - Identify critical paths and optimize the design for better performance.
7. **Verification:**
    - Verify the design using post-implementation simulation to ensure that the implemented design behaves as expected.
    - Perform hardware debugging and address any issues that arise.
8. **Generate Programming File:**
    - Generate the final programming file (bitstream) that will be used to configure the FPGA.
    - This file contains information about how the FPGA resources should be configured to implement the design.
9. **Configuration:**

- Configure the FPGA with the generated bitstream using Vivado or other programming tools.
- Connect the FPGA to the host system or other components as required.

10. Validation:

- Validate the implemented design on the hardware to ensure proper functionality.
- Perform additional testing and validation steps to ensure that the design meets all requirements.

11. Documentation:

- Generate documentation for the project, including reports on synthesis, implementation, and timing analysis.
- Document constraints, pin assignments, and any other relevant information.

12. Integration with Software:

- If applicable, integrate the hardware design with software using the appropriate tools.
- Develop software drivers and applications to interact with the implemented hardware.

13. Debugging and Optimization:

- Debug and optimize the design as needed, addressing any issues that arise during integration or testing.
- Fine-tune the design for better performance or resource utilization.

14. Revision Control:

- Use version control tools to manage different versions of the project and track changes.

CHAPTER- 3

MINI PROJECT DESCRIPTION

PROJECT WORK

VERILOG IMPLEMENTATION OF DIGITAL CIRCUIT DESIGN ON FPGA (FIELD PROGRAMMABLE GATE ARRAY) ON VIVADO

TOPIC: ALARM CLOCK USING VERILOG

Alarm Clock Using Verilog

Verilog code for an alarm clock on FPGA is presented in this project. The Verilog code is fully synthesizable for FPGA implementation.
The simple alarm clock is shown in the following figure. The alarm clock outputs a real-time clock with a 24-hour format and also provides an alarm feature. Users also can set the clock time through switches.

A clock in verilog with Alarm

We are generating a clock with 7 output signals including Alarm signal, Hour, Minute, and seconds. All the signals are discussed in detail further.

The clock generated is in a 24 - hour format. We can give an initial time value to the system when reset signal=1 or by turning the signal LD_time=1.

You can further set the alarm time by turning LD_alarm=1. The alarm is enabled or disabled using the input AL_ON. The alarm rings only if AL_ON is 1.

STOP_al signal is used to stop the alarm. The input clock given as input is 10Hz.

We have generated a clk with time period 1 second from this input clock and used it to increment seconds and further minutes and hours.

Input Signals:

reset : Active high reset pulse, to set the time to the input hour and minute (as defined by the H_in1, H_in0, M_in1, and M_in0 inputs) and the second to 00. It should also set the alarm value to 0.00.00, and to set the Alarm (output) low.For normal operation, this input pin should be 0.

clk : A 10Hz input clock. This should be used to generate each real-time second

H_in1 : A 2-bit input used to set the most significant hour digit of the clock (if LD_time=1),or the most significant hour digit of the alarm (if LD_alarm=1). Valid values are 0 to 2.

H_in0 : A 4-bit input used to set the least significant hour digit of the clock (if LD_time=1) or the least significant hour digit of the alarm (if LD_alarm=1). Valid values are 0 to 9.

M_in1 : A 4-bit input used to set the most significant minute digit of the clock (if LD_time=1),or the most significant minute digit of the alarm (if LD_alarm=1). Valid values are 0 to 5

M_in0 : A 4-bit input used to set the least significant minute digit of the clock (if LD_time=1),or the least significant minute digit of the alarm (if LD_alarm=1). Valid values are 0 to 9.

LD_time : If LD_time=1, the time should be set to the values on the inputs H_in1, H_in0, M_in1, and M_in0. The second time should be set to 0.If LD_time=0, the clock should act normally (i.e. second should be incremented every 10 clock cycles).

LD_alarm : If LD_alarm=1, the alarm time should be set to the values on the inputs H_in1, H_in0, M_in1, and M_in0.If LD_alarm=0, the clock should act normally.

STOP_al : If the Alarm (output) is high, then STOP_al=1 will bring the output back low.

AL_ON : If high, the alarm is ON (and Alarm will go high if the alarm time equals the real time). If low the the alarm function is OFF.

<u>Output Signals:</u>

Alarm : This will go high if the alarm time equals the current time, and AL_ON is high. This will remain high, until STOP_al goes high, which will bring Alarm back low.

H_out1 : The most significant digit of the hour. Valid values are 0 to 2.

H_out0 : The least significant digit of the hour. Valid values are 0 to 9.

M_out1 : The most significant digit of the minute. Valid values are 0 to 5.

M_out0 : The least significant digit of the minute. Valid values are 0 to 9.

S_out1 : The most significant digit of the minute. Valid values are 0 to 5.

S_out0 : The least significant digit of the minute. Valid values are 0 to 9.

Internal Signals:

clk_1s : 1-s clock

tmp_1s : count for creating 1-s clock

tmp_hour, tmp_minute, tmp_second : counter for clock hour, minute and second

c_hour1,a_hour1 : The most significant hour digit of the temp clock and alarm.

c_hour0,a_hour0 : The least significant hour digit of the temp clock and alarm.

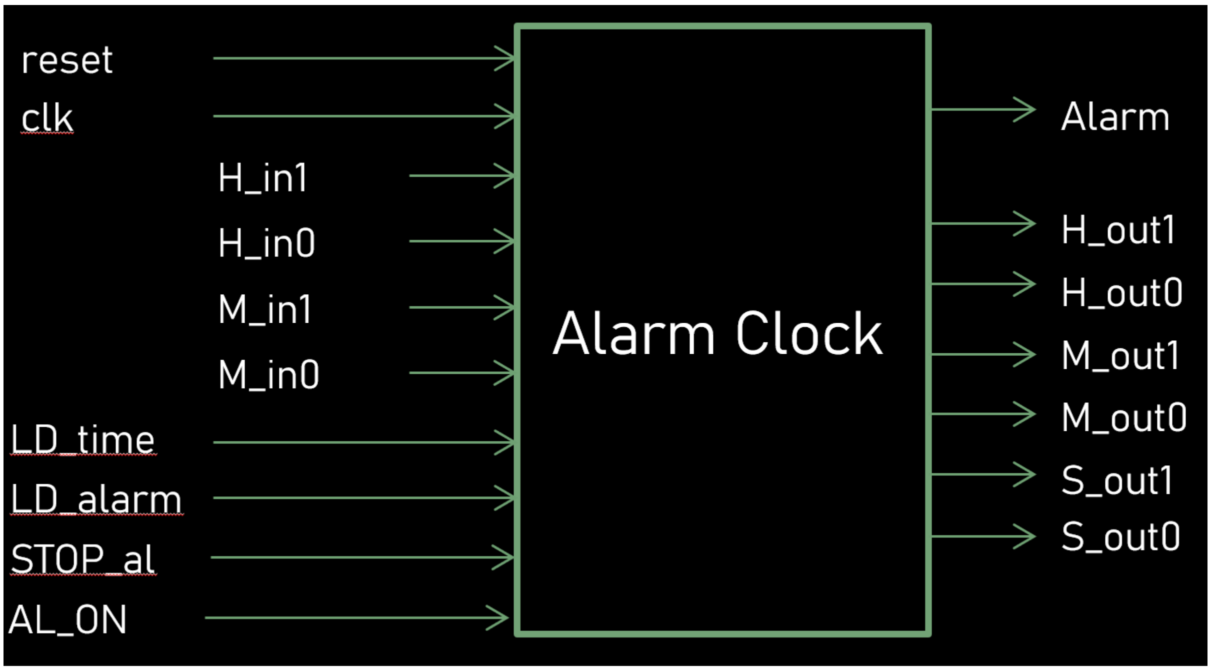
c_min1,a_min1 : The most significant minute digit of the temp clock and alarm.

c_min0,a_min0 : The least significant minute digit of the temp clock and alarm.

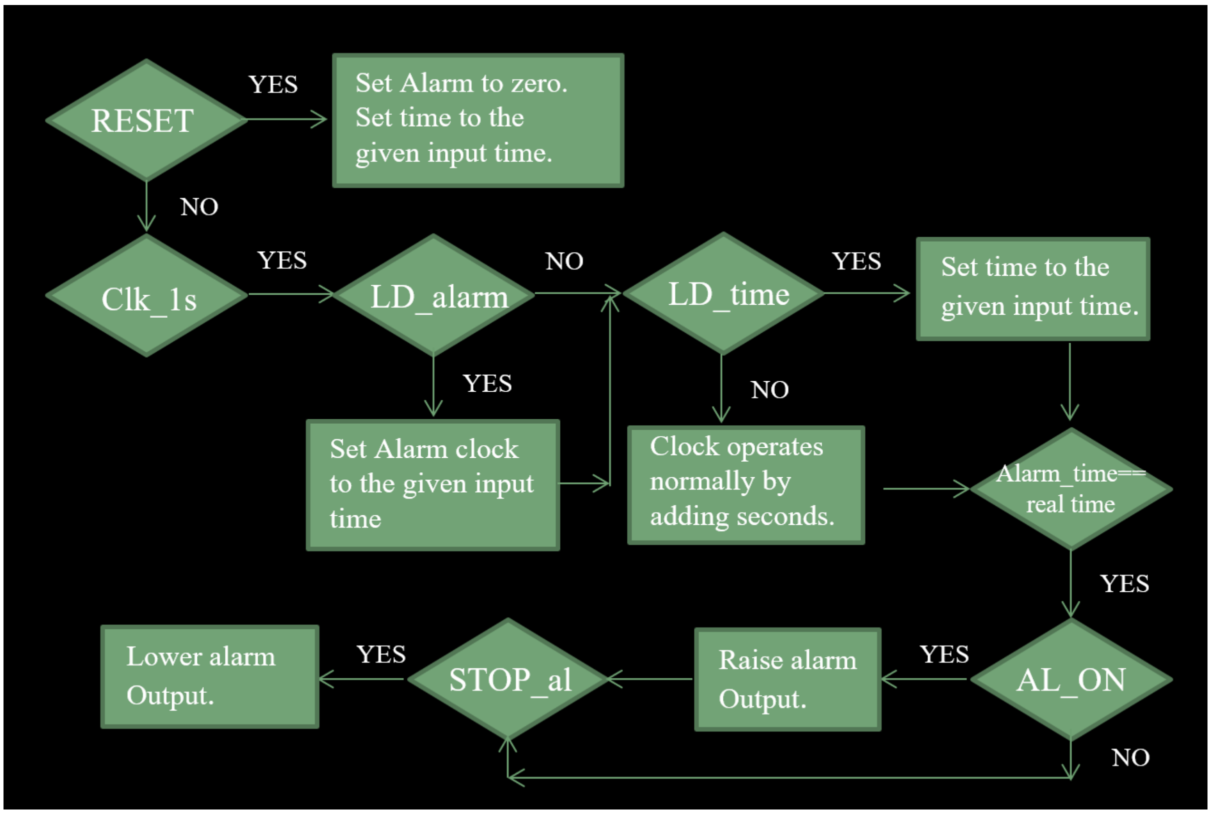
c_sec1,a_sec1 : The most significant second digit of the temp clock and alarm.

c_sec0,a_sec0 : The least significant minute digit of the temp clock and alarm.

BLOCK DIAGRAM:



FLOWCHART:


RESET
YES
Set Alarm to zero. Set time to the given input time.
NO
Clk_1s
YES
LD_alarm
NO
LD_time
YES
Set time to the given input time.
YES
Set Alarm clock to the given input time
NO
Clock operates normally by adding seconds.
Alarm_time== real time
YES
AL_ON
YES
Raise alarm Output.
STOP_al
YES
Lower alarm Output.
NO

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);

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

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

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

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;
endmodule
```

TESTBENCH:

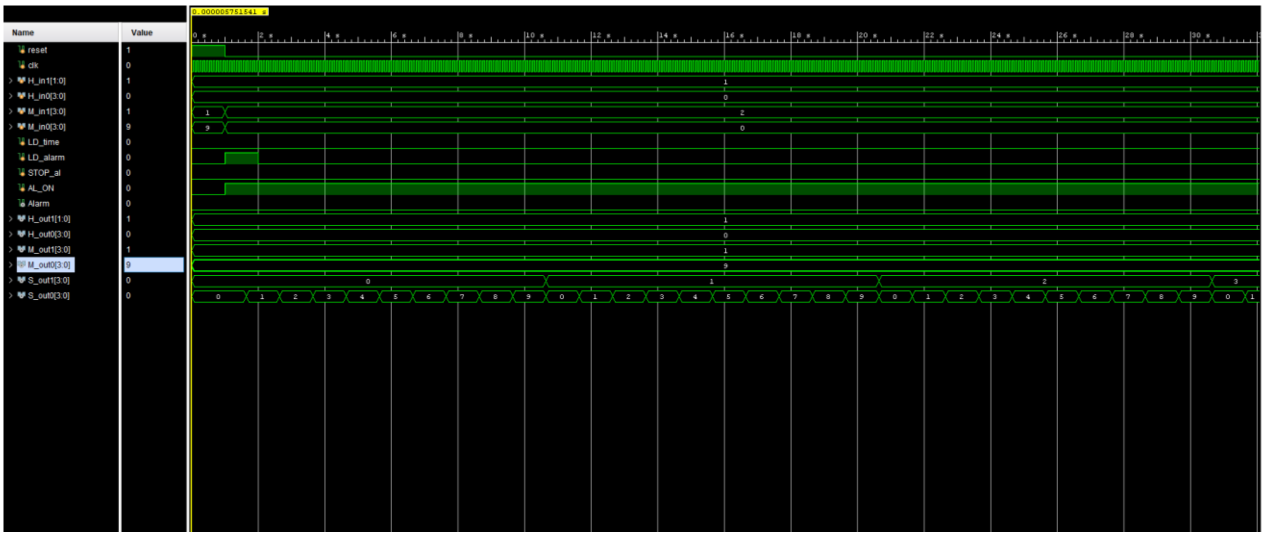
```
module Testbench;
 reg reset;
 reg clk;
 reg [1:0] H_in1;
 reg [3:0] H_in0;
 reg [3:0] M_in1;
 reg [3:0] M_in0;
 reg LD_time;
 reg LD_alarm;
 reg STOP_al;
 reg AL_ON;
 // Outputs
 wire Alarm;
 wire [1:0] H_out1;
 wire [3:0] H_out0;
 wire [3:0] M_out1;
 wire [3:0] M_out0;
 wire [3:0] S_out1;
 wire [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)
 );
 initial begin
  clk = 0;
  forever #50000000 clk = ~clk;
 end
 initial begin
 // Initialize Inputs
 reset = 1;
 H_in1 = 1;
 H_in0 = 0;
 M_in1 = 1;
 M_in0 = 9;
 LD_time = 0;
 LD_alarm = 0;
 STOP_al = 0;
 AL_ON = 0;
 #1000000000;
 reset = 0;
 H_in1 = 1;
 H_in0 = 0;
 M_in1 = 2;
 M_in0 = 0;
 LD_time = 0;
 LD_alarm = 1;
 STOP_al = 0;
 AL_ON = 1;
#1000000000;
 reset = 0;
 H_in1 = 1;
 H_in0 = 0;
 M_in1 = 2;
 M_in0 = 0;
 LD_time = 0;
 LD_alarm = 0;
 STOP_al = 0;
 AL_ON = 1;
wait(Alarm);
#1000000000;
#1000000000;
#1000000000;
#1000000000;
```

```
#1000000000;

#1000000000;

STOP_al = 1;

 end

endmodule
```

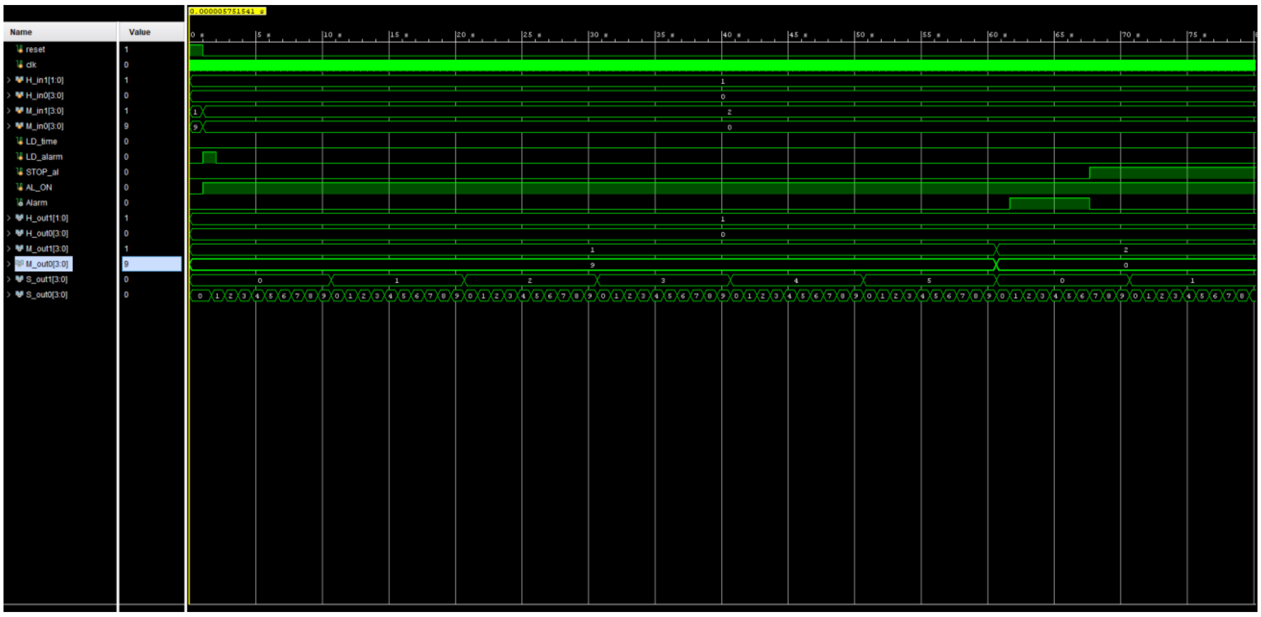
SIMULATION & SYSTHESIS RESULTS

SIMULATION:

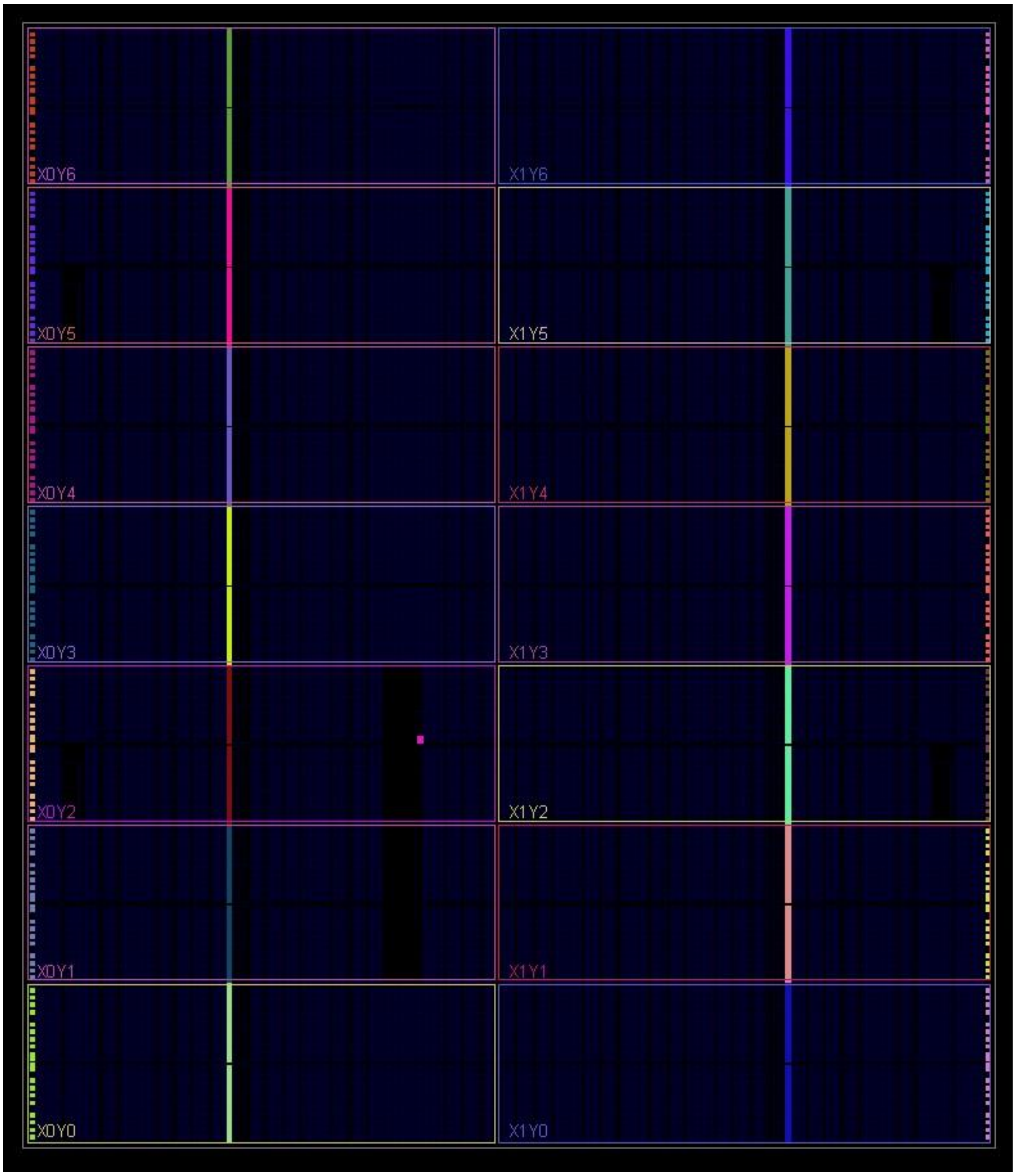
- **30 Seconds Timeframe**

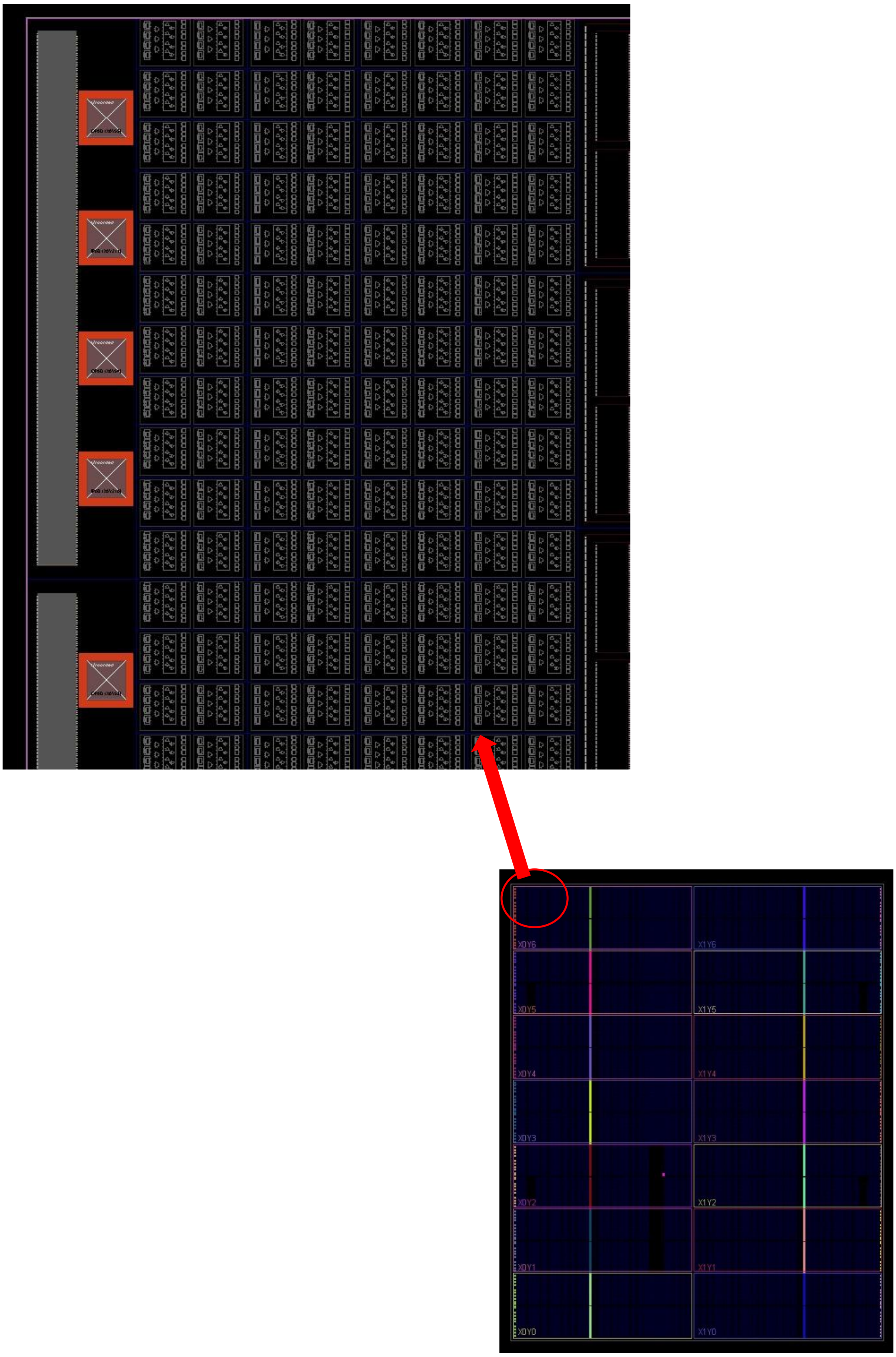


- **75 Seconds Timeframe**

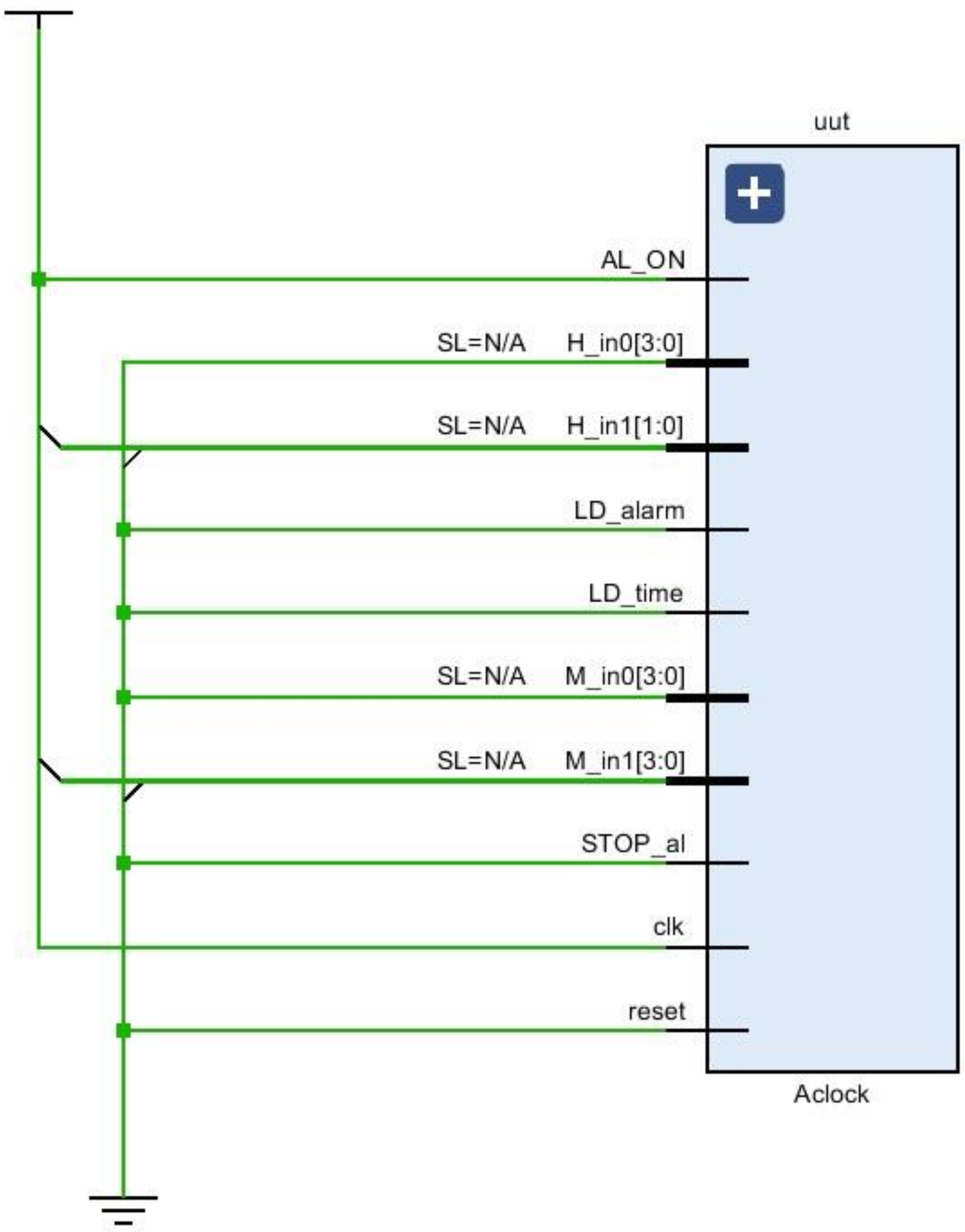


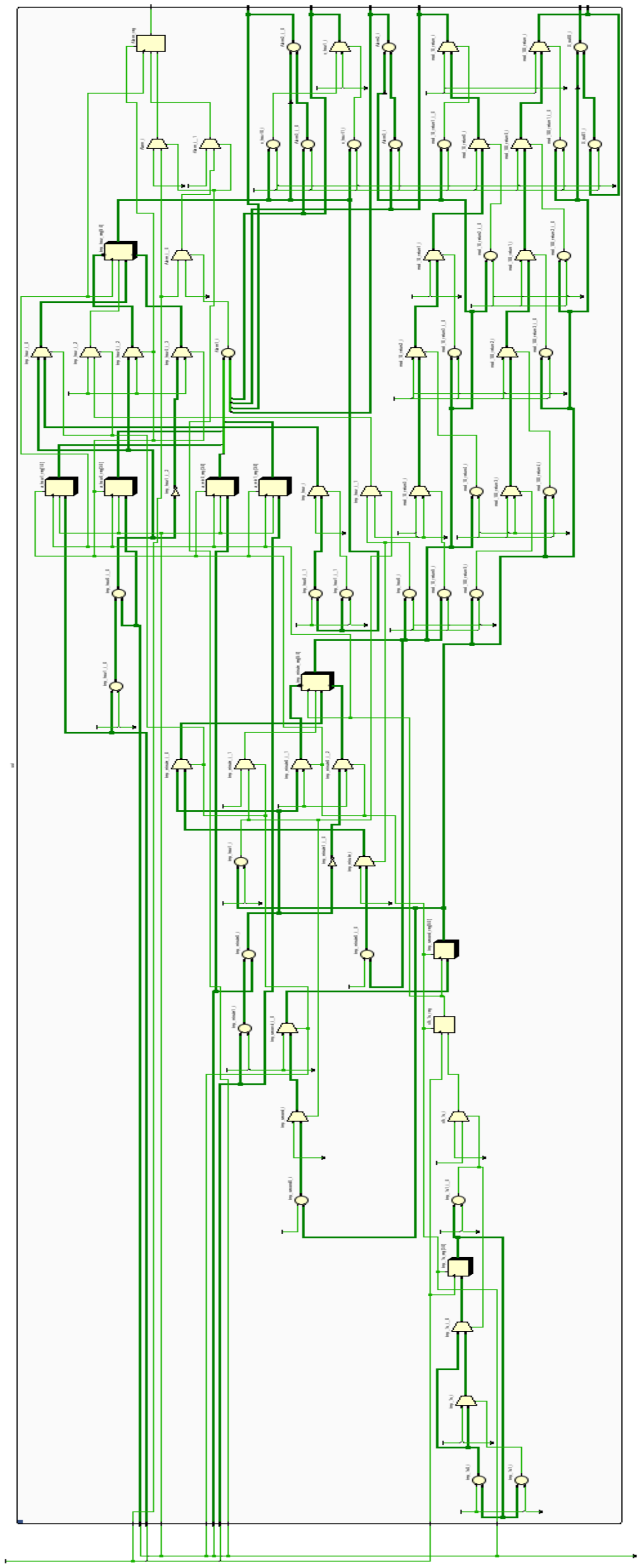
DEVICE LAYOUT AFTER SYNTHESIS:



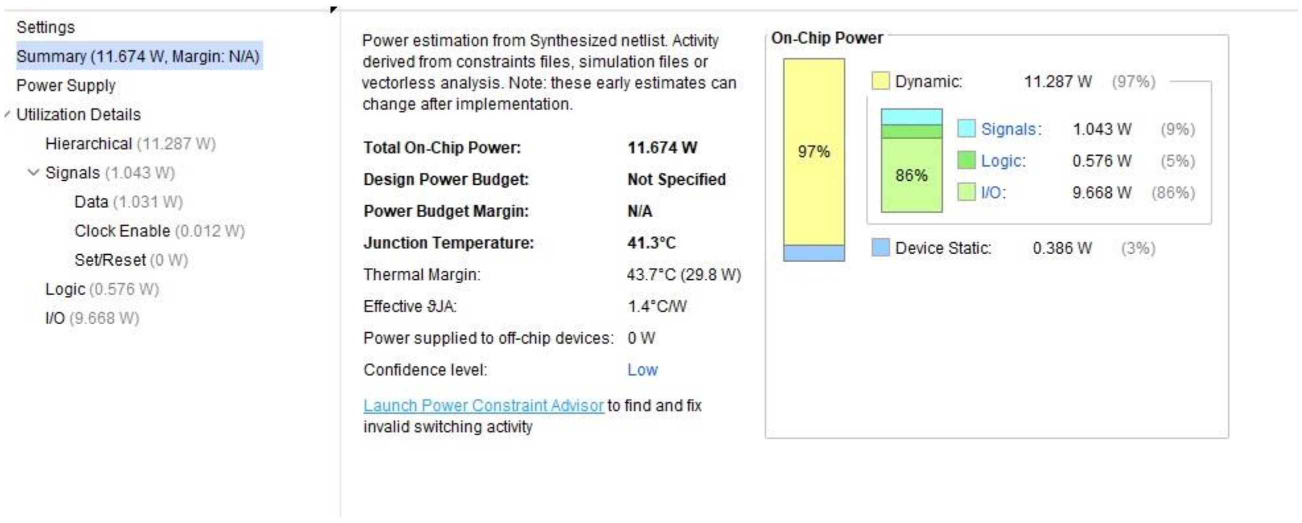

X0Y6
X1Y6
X0Y5
X1Y5
X0Y4
X1Y4
X0Y3
X1Y3
X0Y2
X1Y2
X0Y1
X1Y1
X0Y0
X1Y0

SCHEMATIC AFTER SYNTHESIS:





POWER REPORT:



Settings
Summary (11.674 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (11.287 W)
Signals (1.043 W)
Data (1.031 W)
Clock Enable (0.012 W)
Set/Reset (0 W)
Logic (0.576 W)
I/O (9.668 W)

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

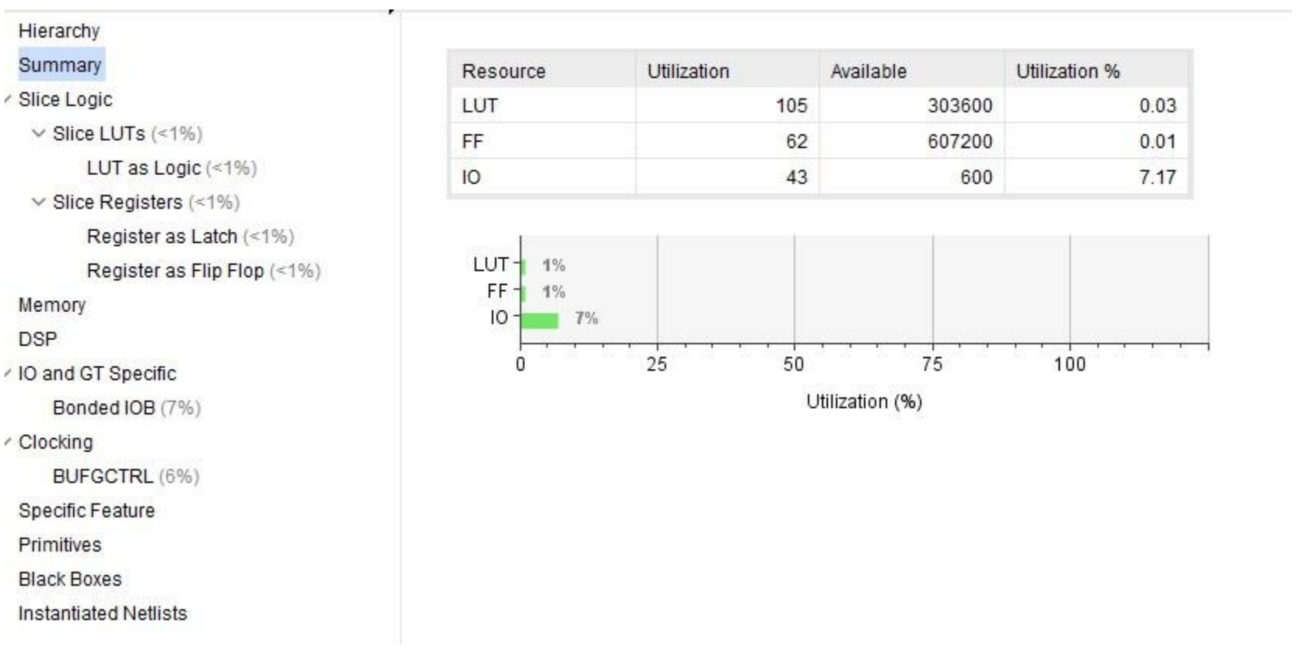
Total On-Chip Power:	**11.674 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**41.3°C**
Thermal Margin:	43.7°C (29.8 W)
Effective ϑJA:	1.4°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	11.287 W	(97%)
Signals:	1.043 W	(9%)
Logic:	0.576 W	(5%)
I/O:	9.668 W	(86%)
Device Static:	0.386 W	(3%)

UTILIZATION REPORT:



Hierarchy
Summary
Slice Logic
Slice LUTs (<1%)
LUT as Logic (<1%)
Slice Registers (<1%)
Register as Latch (<1%)
Register as Flip Flop (<1%)
Memory
DSP
IO and GT Specific
Bonded IOB (7%)
Clocking
BUFGCTRL (6%)
Specific Feature
Primitives
Black Boxes
Instantiated Netlists

Resource	Utilization	Available	Utilization %
LUT	105	303600	0.03
FF	62	607200	0.01
IO	43	600	7.17

REFERENCES

- https://www.xilinx.com/products/design-tools/vivado.html
- https://github.com/Arjun-Narula/Clock-with-Alarm
- https://www.youtube.com/watch?v=pTk1H50e8bI&t=7s
- https://www.youtube.com/watch?v=yy2ojvC6ssw
- https://www.fpga4student.com/2016/11/verilog-code-for-alarm-clock-on-fpga.html