Summer Training
Report On

DESIGNING VERILOG CODE FOR THE DIGITAL BLOCK OF A ROIC (READ-OUT INTEGRATED CIRCUIT)

At

Solid State Physics Laboratory

Defence Research and Development Organization
Timarpur, Lucknow Road, Delhi-110008

in partial fulfilment of the degree

Bachelor of Engineering

In

Electronics and Tele Communication Engineering

JADAVPUR UNIVERSITY, KOLKATA

Submitted by-

Shivam Mandal

002210701136

Under the Guidance of –

Navneet Kaur Saini (Scientist E)

DECLARATION

This is to certify that Summer Training Report entitled DESIGNING VERILOG CODE FOR THE DIGITAL BLOCK OF A ROIC (READ-OUT INTEGRATED CIRCUIT) which is submitted by Shivam Mandal in partial fulfillment of the requirement for the award of degree B.Ed. in Department of Electronics and Tele-Communication Engineering of Jadavpur University, is a record of the candidate own work carried out by him under my/our supervision. The matter embodied in this report is original and has not been submitted for the award of any other degree. I hereby declare that this submission was his own work and that, to the best of his knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

SIGNATURE

NAVNEET
KAUR SAINI
(SCIENTIST E)

ORGANISATION PROFILE

Organisation Profile

The **Defence Research and Development Organisation (DRDO)** is a part of the Department of Defence Research and Development which is in the **Ministry of Defence, Government of India**. DRDO plays a key role in building self-sufficiency in defense technology also they develop and build state--of–the art systems and equipment according to the quality needs from Indian Armed Forces' front. It acts as preeminent research and development institution for military technology and we see it also have a wide range of skills in aeronautics, armaments, electronics, missiles, high performance computing, simulation, life sciences and many other fields. DRDO is not just on top of the game when it comes to deliver state of technological quality to the armed forces also, they put out what can be used for the public sector that in turn contribute to our nation's development.

Vision

"To make India prosperous by establishing a world-class science and technology base and equipping our Defence Services with internationally competitive systems and solutions."

DRDO operates through a vast network of specialized laboratories and research centres spread across the country. One of these premier institutions is the **Solid State Physics Laboratory (SSPL)**, located in Timarpur, Delhi.

About Solid State Physics Laboratory (SSPL)

Established in 1962, the **Solid State Physics Laboratory (SSPL)** is a core research lab under DRDO, dedicated to developing high-purity materials, devices, and sub-systems based on solid-state physics. SSPL has a long-standing vision to emerge as a centre of excellence in the domain of solid-state technologies with strategic defence applications.

The laboratory's mission includes:

- Development and characterization of advanced semiconductor materials.
- Design and fabrication of solid-state devices and microelectronics.
- Infrastructure development for device prototyping and validation.

SSPL's ongoing work includes the development of laser diodes (pulsed and CW), photodetectors, MEMS sensors, and specialized integrated circuits for advanced defence applications.

Relevance to This Project

During my time at SSPL, I had the opportunity to work on a project titled **"Designing Verilog Code for the Digital Block of a Read-Out Integrated Circuit (ROIC)"**, which is directly aligned with SSPL's vision of building advanced semiconductor and signal-processing sub-systems.

The objective of this project was to design and simulate the control logic for a pixel matrix scanning operation — a critical component in CMOS image sensors and ROICs used in night vision, missile guidance, and surveillance systems. The pixel readout logic was implemented using Verilog HDL, and simulations were carried out to validate the design in both 3×3 and 16×16 matrix configurations.

This project falls under the broader umbrella of SSPL's semiconductor device design and integration domain, contributing to the development of indigenous imaging solutions for defence-grade electronics.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to **Navneet Kaur Saini (Scientist 'E')**, for her invaluable guidance, continuous support, and encouragement throughout the course of this project. Her expert advice and insightful suggestions were instrumental in shaping the direction of my work.

I also wish to thank the entire team at the lab for providing me with the necessary tools, environment, and exposure to work on this technically rich and challenging project. Their feedback during our discussions helped me learn and improve significantly.

I extend my heartfelt thanks to my professors, peers, and family for their moral support, which kept me motivated throughout the development of this report.

Lastly, I thank all those who have directly or indirectly contributed to the successful completion of this project.

ABSTRACT

This project focuses on the design and implementation of Verilog code for the digital block of a Read-Out Integrated Circuit (ROIC) used in CMOS imaging systems. The primary objective was to model a finite state machine (FSM) that controls the sequential activation of row and column lines for pixel scanning in a 16×16 array.

The design mimics a snapshot-mode image capture scenario, where synchronization signals such as fsync and intg control the timing of scanning operations. A matrix scanner FSM was developed and simulated, which generates control signals for selecting rows and scanning columns in a systematic sequence. The system was designed using Verilog HDL and simulated using a testbench to verify its timing behavior and logical correctness.

Through this project, a complete digital FSM flow was implemented — from defining state transitions to validating waveform outputs in Vivado. The design was tested for both 3×3 and 16×16 matrices, and synthesis results confirm efficient logic utilization. The report elaborates on the theoretical background, FSM development, Verilog coding, timing diagrams, simulation analysis, and synthesis outcomes, offering a comprehensive understanding of the ROIC digital logic design process.

CONTENTS

Chapter 1 – Introduction to ROIC and Pixel Matrix Systems

1.1 Overview of ROIC and Its Importance

In the field of modern electronics, **Read-Out Integrated Circuits (ROICs)** play a very important role, especially in sensor-based systems like thermal imaging, X-ray detectors, medical scanners, and other pixel-based sensor applications. These systems usually have a 2D array of pixels that capture analog signals (mostly voltages or currents) depending on how much light, radiation, or any other physical parameter they receive. But collecting these analog signals from every pixel in a fast and systematic way is not possible manually. That's where ROICs come in.

A ROIC is basically the part of the sensor system that "reads out" the analog data from each pixel in a timed and ordered manner, converts it (or prepares it) for further processing, and sends it out of the chip. The readout process needs to be very efficient, accurate, and fast because any delay or mistake will directly affect the overall image quality and real-time performance of the system.

Most ROICs include both **analog** and **digital** blocks. The analog part handles tasks like integration, amplification, and sample-and-hold, while the digital part takes care of signal sequencing, timing control, and addressing logic for selecting rows and columns.

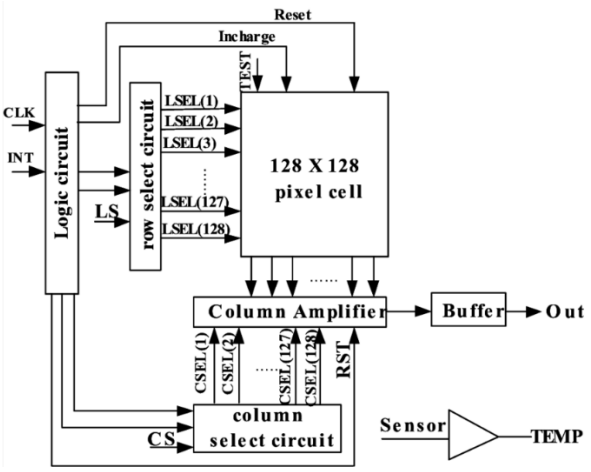


Figure 1.1: Block Diagram of ROIC

1.2 Motivation Behind the Project

The main reason for doing this project is to explore how the **digital section of a ROIC** works and to try building a working model of it using **Verilog**, which is one of the most used hardware description languages (HDL). I chose Verilog over other HDLs because it is widely taught and supported in both academics and industry.

During my internship and lab experiments, I found that most students and even entry-level engineers struggle with **timing control** in hardware circuits, especially when signals like fsync and intg come into play. These two signals are very common in snapshot-type imaging systems where each frame needs to be captured with precise control.

So, the aim of this project is to design, write, and simulate a **finite state machine (FSM)** in Verilog that can control the scanning of a pixel array (3×3 and 16×16) in a way that reflects real-world timing protocols.

1.3 Objective of the Project

The main objectives of this project can be listed as follows:

- To study the **working principle of ROICs**, especially the digital control part.
- To design a **finite state machine (FSM)** that controls pixel scanning row-wise and column-wise.
- To implement this FSM in **Verilog**, initially for a 3×3 matrix and then scale it to 16×16.
- To simulate the code using **Vivado** and observe waveform behaviour to verify if the timing matches real ROIC logic.
- To ensure that signals like fsync, intg, and master_rst work in sync with the internal states of the FSM.
- To build a testbench in Verilog that repeatedly checks the system and ensures proper looped operation like in real imaging hardware.

1.4 Scope of the Project

This project mainly focuses on the **digital logic control** inside a ROIC. It does not deal with analog parts like integration amplifiers or sample-and-hold circuits because those require transistor-level SPICE simulations or layout design, which is not feasible for this B.Tech-level report.

Instead, my project focuses on:

- Understanding how a pixel array is scanned row by row and column by column.
- Designing the control signals in such a way that the **matrix is scanned only after a valid fsync and intg sequence**.
- Creating Verilog code that is **synthesizable and reusable** for different matrix sizes.

- Making sure the FSM works in continuous loop, just like how a real camera sensor keeps scanning each frame repeatedly.

Some of the constraints I kept while working on this project:

- Only use Verilog (not SystemVerilog).
- Code must work with Xilinx Vivado, since it was available in the lab.
- Use only behavioral and synthesizable constructs—no vendor-specific IP cores.
- Keep the row and column decoders simple (one-hot based) and easy to scale.

1.5 Structure of the Report

The report is divided into the following sections:

- **Chapter 2 – Fundamentals of ROIC and Pixel Decoding**
  This section explains the basic working of ROICs, focusing on pixel arrays, unit cell structure, and how decoding is handled row-wise and column-wise. It also discusses how external signals like fsync and intg coordinate the scanning process.
- **Chapter 3 – Introduction to Verilog HDL and FSM Design**
  This newly added chapter provides an overview of hardware description languages, focusing on Verilog HDL. It compares Verilog with VHDL and SystemVerilog, discusses finite state machines (FSMs), and explains Moore vs Mealy models, FSM coding styles, and their application in pixel scanning control.
- **Chapter 4 – Verilog Design and FSM Implementation**
  This chapter focuses on the Verilog implementation of the digital block of an ROIC. It includes the FSM state diagram, Verilog code for the 3×3 and 16×16 pixel array scanners, row and column decoding logic, and detailed explanation of each state and transition.
- **Chapter 5 – Simulation Results and Timing Verification**
  Here, simulation waveforms from Vivado are presented, showing how the FSM behaves under different conditions. Special emphasis is placed on timing accuracy with respect to fsync, intg, and row/col activation. Snapshots and analysis are included to verify correctness.
- **Chapter 6 – Conclusion and Future Scope**
  This section summarizes the key outcomes of the project, reflects on the challenges faced, and suggests how this work can be extended in the future—such as moving to analog-digital integration, higher matrix resolution, or real-time hardware testing.

1.6 Learning Goals as a Student

Apart from submitting a working project and report, I had some personal learning goals for this project:

- Getting more confident in writing **modular Verilog code**.
- Understanding how FSMs work in real-time applications.
- Using **Vivado simulator** properly and understanding waveform analysis.
- Improving my ability to explain digital systems not just by code, but through diagrams and timing charts.
- Practicing how to write a full technical report based on practical lab work and simulation results.

1.7 Relevance of ROIC in Modern Applications

In today's fast-moving technological world, image sensors are not only limited to mobile phones or DSLR cameras. ROICs are now essential components in a wide variety of domains. For example:

- **Defence and Aerospace:** Night vision, thermal targeting, and missile guidance systems use IR sensor arrays. ROICs are needed to read out data quickly from those arrays in real-time.
- **Medical Imaging:** Flat-panel detectors used in X-ray machines rely on ROICs to convert pixel charges into useful data for diagnosis.
- **Scientific Instrumentation:** High-resolution CCD and CMOS sensors used in telescopes or microscopes use ROICs to achieve precise measurements of light.

In all these systems, one thing is common: the ROIC must work **fast**, **reliably**, and with **minimal noise**. That means the digital control part, which this project focuses on, plays a huge role in ensuring that performance.

Even if the analog circuitry is perfect, poor timing control in the FSM can result in overlapping pixel reads, skipped columns, or even image frame corruption. So having a correct and verified FSM is not optional—it's essential.

1.8 Personal Motivation for Choosing This Topic

From the beginning of my third year, I was interested in digital VLSI systems and hardware design. While doing lab experiments in Vivado and experimenting with FSMs, I started getting curious about how timing controllers actually work in real circuits.

Then during one of our VLSI lectures, our professor briefly mentioned how FSMs are used inside ROICs and image sensors. That was the first time I heard about fsync and intg signals. Later, I did more reading and found out that most real-world sensor chips, even high-end ones like those from Sony or ON Semi, have similar digital timing engines under the hood.

So when we had to propose a mini-project for our lab, I decided to choose a **practical application** where I could apply what I learned in digital design, and that's how this project started. Later, it got converted into a full major project under the guidance of **Mrs. NAVNEET KAUR SAINI** who helped me explore more realistic behaviour and constraints used in real ROICs.

1.9 Challenges in ROIC Digital Design

Working on this project taught me that even if you are not building the whole chip, the **digital part of the ROIC** comes with many challenges of its own:

- **Timing accuracy:** The row and column scanning cannot begin randomly; they must follow a protocol that includes fsync trigger, intg time, and delay cycles.
- **FSM design:** Designing a proper FSM with correct transitions for multiple states like IDLE, WAIT, ROW_ENABLE, COL_SCAN, etc., and making sure it loops correctly.
- **Verilog constraints:** Using only synthesizable Verilog while avoiding things like delays (#) or blocking statements inside always blocks that can break real logic.
- **Simulation bugs:** Many times, the testbench passed but the waveform did not match the expected behavior. That taught me to rely on **waveform debugging**, not just code output.

All of these things made the project more than just a coding task—it was a complete learning process.

1.10 Terminologies Used Throughout the Report

Before moving forward, it's helpful to define some terms that will keep appearing in the rest of this report:

Term	Meaning
FSM	Finite State Machine – a logic system that transitions through predefined states based on clock and inputs
fsync	Frame Sync – a signal that marks the beginning of a new image frame

intg	Integration Time – the duration during which each pixel collects charge
row_en	Row Enable – signal to activate a particular row for scanning
col_clk	Column Clock – toggled to read each column sequentially
snapshot mode	A method where the whole image is captured at once (not row-by-row like rolling shutter)
Vivado	A software tool by Xilinx used for simulating and synthesizing Verilog code

Table 1:Terminologies Table

Chapter 2 - Fundamentals of ROIC and Pixel Decoding

2.1 What is a ROIC?

A **Read-Out Integrated Circuit (ROIC)** is a specialized chip used in sensor arrays to interface with individual pixels and process their output signals. It acts as the bridge between the **analog sensing elements (pixels)** and the **digital data output** used for further processing or display.

In any sensor array, each pixel generates an analog signal in response to a stimulus (like light, heat, or radiation). The ROIC is responsible for selecting each pixel in a timed sequence, reading the analog voltage or current, and sending it to the output buffers. This process happens **row-by-row and column-by-column**, usually controlled by a **digital FSM** or scan logic.

ROICs are most commonly used in:

- **Infrared (IR) focal-plane arrays**
- **X-ray detectors**
- **Optical CMOS/CCD sensors**
- **Radiation detectors for space imaging**

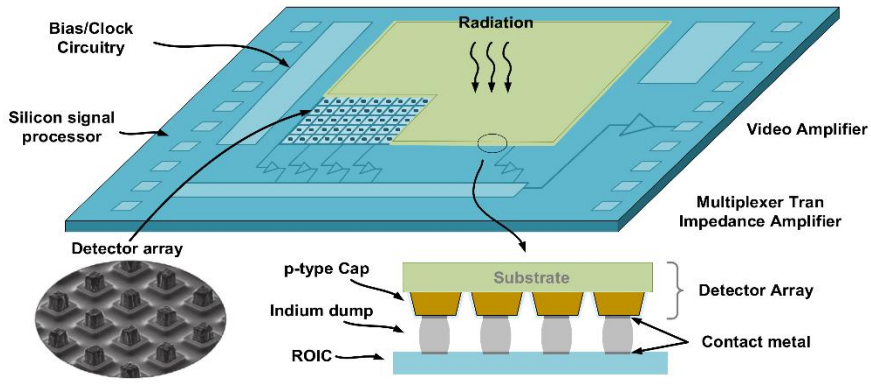


Figure 2.1: Real-world ROIC layout

2.2 ROIC Architecture and Its Blocks

Although ROICs can be very complex, a simplified architecture is shown below:

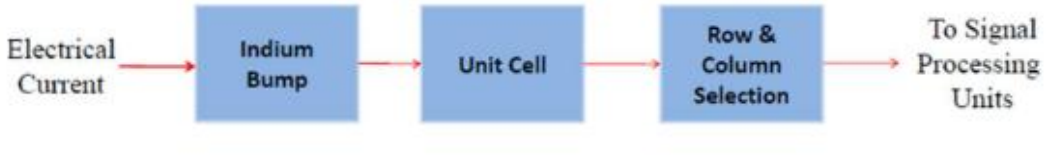


Figure 2.2: Simplified ROIC block diagram

The basic blocks are:

- **Pixel Array (Analog front-end)**: A grid of pixels arranged in rows and columns. Each pixel may include a photodetector, integration capacitor, and a switch.
- **Row and Column Decoders**: Used to select which pixel to read from. The decoder logic activates one row at a time, and each column output can be sampled in parallel or sequentially.
- **Timing and Control Block**: This is the **digital FSM** that manages the flow of control signals like row_en, col_clk, and the scan sequence. This block is what we are designing in this project using Verilog.
- **Sample-and-Hold and Output Buffers**: Once the signal from the selected pixel is read, it's stored temporarily and then passed to the output. This analog section is not part of our scope.

2.3 Snapshot Mode in ROICs

There are two main types of ROIC scanning methods:

- **Rolling Shutter** – where each row is scanned and read sequentially with a time gap. This can lead to distortions in fast-motion capture.
- **Snapshot Mode** – all rows integrate light **simultaneously**, and then get read out sequentially after integration ends. This prevents motion blur and ensures accurate temporal capture.

In this project, we're implementing **Snapshot Mode**, which uses two major global signals:

- fsync – signals the start of a new frame
- intg – defines the integration period (active high)

Only **after the falling edge of intg** do we begin scanning the pixel array, starting from row 0.

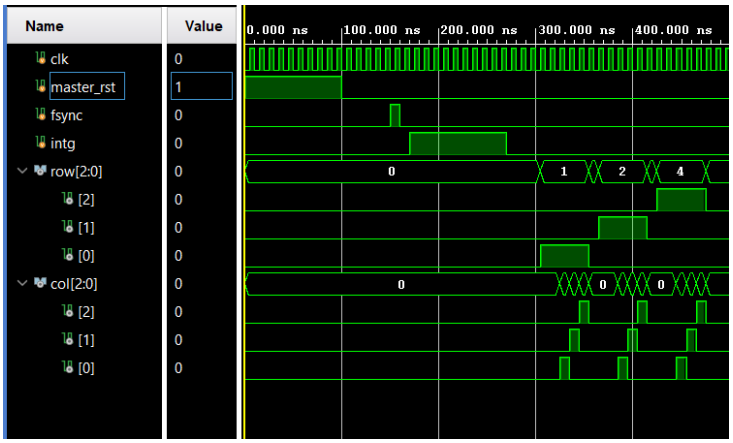


Figure 2.3: Timing diagram of fsync, intg, and row scanning as per snapshot mode

2.4 Row and Column Addressing Mechanism

To scan a matrix, say **3×3** or **16×16**, we need to activate one **row** at a time and sequentially go through all **columns**.

Row Selection (Row Enable)

We use a **one-hot encoding** where only one row is active at a time. For example:

Clock Cycle	Row[2:0]
0	001
1	010
2	100

Figure 2.4: One-hot row selection waveforms

Column Selection (Column Clock)

Each column is selected sequentially using an FSM or shift register.

Clock Cycle	Col[2:0]
0	001
1	010
2	100

This process continues for each row. Once all columns of one row are scanned, the next row begins. In our FSM logic, we've ensured that:

- There's a **2-clock delay** after intg falls before row[0] starts.
- There's a **1-clock delay** before column[0] begins.

2.5 Control Signal Explanation – fsync and intg

These two signals are the main **external inputs** to our FSM:

- fsync – This is a **1-cycle pulse** that indicates a new frame is about to start. It usually comes from an external system (e.g., a camera controller).
- intg – This signal goes high after fsync and stays high for a fixed number of clock cycles. During this period, the pixels accumulate charge. When it goes low, it means integration is over, and the readout can start.

Timing Rule We Used

Signal	Active When	Duration
fsync	Rising edge trigger	1 clock cycle
intg	After fsync	10 clock cycles
row_en	2 clocks after intg↓	1 clock cycle
col_clk	After row_en	Sequential toggling

Figure 2.5: FSM state transition diagram showing IDLE → WAIT → ROW_EN → COL_SCAN

2.6 Decoder Topologies Used in ROICs

There are different ways to design the row and column decoders:

- **Binary decoder** – simple, requires fewer wires but more control logic.
- **One-hot decoder** – easier to control, good for FSMs, used in this project.

In our Verilog code, we use one-hot style where only one bit of the row or column signal is high at a time. This keeps the logic and simulation much easier to follow.

Feature	Binary Decoder	One-Hot Decoder
Number of Control Lines	log2(N) lines required	N lines required
Hardware Complexity	Less wiring, but more logic (decoder required)	More wiring, simpler logic
Speed (Delay)	Slightly slower due to decoding delay	Faster response since no decoding needed
FSM Compatibility	Less intuitive to implement in FSM	Highly compatible with FSM state transitions
Power Consumption	Lower (less switching, fewer lines)	Higher due to multiple active lines
Suitability for ROIC FSM	Not preferred for pixel scanning	Preferred – used in this project
Example for 3 Rows	2-bit binary: 00, 01, 10	3-bit one-hot: 001, 010, 100

Table 2:Comparison between Binary and One-hot decoder logic for row/column selection

Chapter 3 – Introduction to Verilog HDL and FSM Design

3.1 Overview of Hardware Description Languages (HDLs)

In modern digital electronics, Hardware Description Languages (HDLs) play an essential role in describing, simulating, and synthesizing hardware designs. Unlike traditional programming languages like C or Python that describe sequential software behavior, HDLs are meant for **modeling actual hardware circuitry** such as logic gates, flip-flops, counters, and more complex systems like processors and memory controllers.

There are three primary HDLs used in both academia and industry:

- **Verilog**
- **VHDL (VHSIC Hardware Description Language)**
- **SystemVerilog**

All of these allow a designer to describe **how a digital system behaves** and **how it should be implemented in real hardware**, usually on an FPGA or ASIC.

In this project, **Verilog HDL** is used because it is more beginner-friendly, widely supported in tools like **Xilinx Vivado**, and aligns well with the goals of FSM design and matrix scanning logic.

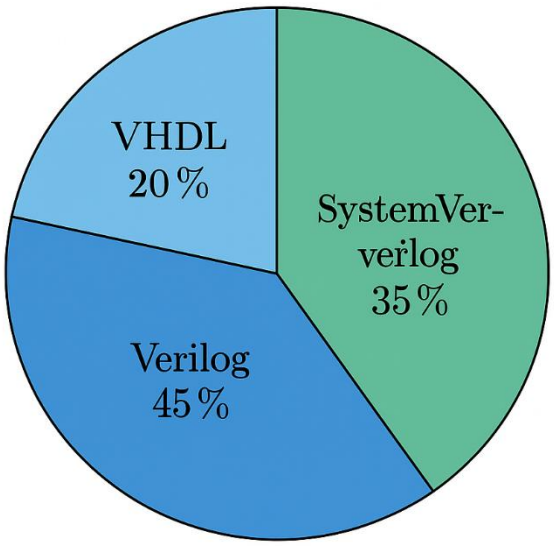


Figure 3.1: HDL usage across industry

3.2 Comparison Between Verilog, VHDL, and SystemVerilog

Feature	Verilog	VHDL	SystemVerilog
Origin	Based on C-like syntax	Based on Ada/Pascal syntax	Extension of Verilog
Use-case	FPGA, ASIC design	More often used in Europe/defense	ASIC design + advanced testbenches
Learning curve	Easier to learn for beginners	Slightly more verbose	Advanced and complex
Tool compatibility	Supported in most tools (Vivado)	Also supported in most tools	Fully supported in latest tools
Simulation & Modeling	Moderate	Strong data typing	Better verification & abstraction
Readability	Concise and compact	Very structured	Improved structure + OO features

Why Verilog is chosen in this project:

- It is concise and easier to write and read.
- It aligns with the FSM-based design needs.
- It is fully supported by Vivado, which was used for simulation.
- Most college-level labs in India and industry internships prefer Verilog.

Here's the syntax example for a simple 2-input AND gate written separately for each HDL:

Verilog

In **Verilog**, the AND gate is implemented using a module block. Inputs and outputs are declared in the module header, and logic is assigned using the assign keyword:

```
module and_gate(input a, b, output y);
  assign y = a & b;
endmodule
```

This concise format is one reason Verilog remains widely used, particularly for RTL design and synthesis workflows.

VHDL

VHDL takes a more formal and verbose approach. It separates interface and implementation using entity and architecture:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity and_gate is
  Port ( a, b : in STD_LOGIC;
         y : out STD_LOGIC);
end and_gate;

architecture Behavioral of and_gate is
begin
  y <= a and b;
end Behavioral;
```

This structure enforces clarity and modularity, making VHDL especially popular in safety-critical domains like aerospace and defense.

SystemVerilog

SystemVerilog enhances Verilog with strong typing and extended capabilities for verification. Here's how the same gate looks:

```
module and_gate(logic a, b, output logic y);
  assign y = a & b;
endmodule
```

The use of logic instead of wire or reg simplifies simulation semantics and reduces ambiguity, making SystemVerilog ideal for large-scale and complex designs.

3.3 Introduction to FSM (Finite State Machine) Design

A Finite State Machine is a type of sequential circuit that moves between a limited number of **predefined states**, depending on the inputs and clock signals. FSMs are used in almost every

digital design: from traffic light controllers to processors, memory interfaces, and even image scanners like in this project.

FSMs are of two types:

- **Moore Machine** – Output depends only on the current state.
- **Mealy Machine** – Output depends on both current state and inputs.

In this project, we use a **hybrid FSM**, which operates mainly like a **Moore machine** but occasionally uses input-based decisions like fsync and intg for transitions.

3.4 Mealy vs Moore FSM – Theory and Practical Use

Feature	**Mealy FSM**	**Moore FSM**
Output depends on	State + Inputs	Only on State
Faster reaction	Yes – responds instantly	No – delayed till next clock
Complexity	Slightly more compact	Easier to design/debug
Example	UART receiver	Traffic light controller
Used in project?	Partially (intg falling edge triggers state change)	Mostly (outputs tied to states)

Practical Note:
In this project, the **falling edge of intg signal** is used to initiate row scanning, which behaves like a Mealy FSM at that point. However, all outputs like row and col depend solely on the FSM state, so the rest behaves like a Moore FSM.

3.5 FSM Design and Coding Styles in Verilog

FSMs in Verilog are typically implemented using:

1. **State Declaration (parameters or enums)**
2. **State Transition Logic** (using case statements)
3. **State Register (to hold current state)**
4. **Output Logic** (based on state or inputs)

Example FSM Structure:

```
// State declarations
parameter IDLE = 2'b00, SCAN = 2'b01, DONE = 2'b10;
```

```
reg [1:0] state, next_state;

// Transition logic
always @(posedge clk or posedge reset)
    if (reset)
        state <= IDLE;
    else
        state <= next_state;

// Next-state logic
always @(*) begin
    case (state)
        IDLE: next_state = (start) ? SCAN : IDLE;
        SCAN: next_state = (done) ? DONE : SCAN;
        DONE: next_state = IDLE;
    endcase
end

// Output logic (Moore)
always @(*) begin
    case (state)
        IDLE: begin ... end
        SCAN: begin ... end
    endcase
end
```

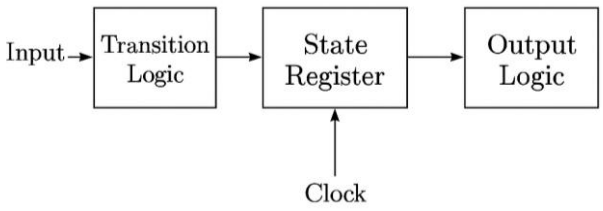


Figure 3.2: FSM logic block diagram

Chapter 4 - Verilog Code & FSM Design

4.1 Why an FSM?

In a snapshot-mode ROIC the digital controller must:

1. **Wait** for a valid fsync pulse.
2. **Sense** the 1-cycle gap.
3. **Track** the 10-cycle high period of intg.
4. **Add** fixed delays (2 clks, 1 clk) before row/column activity.
5. **Loop** through every row and every column—forever—until master_rst is asserted.

A **Finite State Machine (FSM)** is the cleanest way to guarantee those rules every frame.

Design decision: one-hot state encoding → easier to read in a simulator, and the extra flip-flops are negligible for 3 × 3 or 16 × 16 arrays.

4.1.1 Defined States

State	Action (one per clock)
IDLE	All outputs low, wait for fsync & intg both high
WAIT_FALL	Keep watching until intg goes low
ROW_DELAY	Count 2 extra clocks
ROW_EN	Assert current row (row_reg) for 1 clk
COL_SCAN	Toggle col_clk, step through every column
NEXT_ROW	Shift to next row; if last row done → IDLE

Figure 4.1: FSM State Diagram

4.1.2 Key Design Targets

Parameter	Chosen Value	Why?
Clock frequency	50 MHz (sim)	Easy to view 20 clks per frame
fsync pulse width	1 cycle	Follows typical sensor spec
intg high time	10 cycles	Matches lab assignment
Row delay after intg↓	2 cycles	Mimics real ROIC timing budget
Col-0 delay after row_en	1 cycle	Gives clean setup/hold margin
Row counter bits	2 (3×3) / 4 (16×16)	Scalable design

Table 4:Design Constraints & Targets

4.2 Top-Level Module (3 × 3)

```
`timescale 1ns / 1ps


module matrix_shift_scanner_with_ctrl (
    input clk,
    input master_rst,
    input fsync,
    input intg,
    output reg [2:0] row,
```

```
    output reg [2:0] col
);

    // State encoding
    parameter IDLE                = 4'd0,
              WAIT_INTG_RISE      = 4'd1,
              WAIT_INTG_FALL      = 4'd2,
              DELAY_AFTER_INTG_FALL = 4'd3,
              ROW_SETUP           = 4'd4,
              COL_DELAY           = 4'd5,
              COL_SCAN            = 4'd6,
              NEXT_ROW            = 4'd7,
              FRAME_DONE          = 4'd8;

    reg [3:0] state;
    reg [1:0] delay_counter;
    reg [1:0] col_counter;
    reg [1:0] row_counter;
    reg prev_fsync, prev_intg;

    // Edge detection
    wire fsync_rise = fsync & ~prev_fsync;
    wire intg_rise  = intg  & ~prev_intg;
    wire intg_fall  = ~intg & prev_intg;

    always @(posedge clk or posedge master_rst) begin
        if (master_rst) begin
            state       <= IDLE;
            row         <= 3'b000;
            col         <= 3'b000;
            delay_counter<= 0;
            col_counter  <= 0;
```

```
    row_counter  <= 0;
    prev_fsync   <= 0;
    prev_intg    <= 0;
end else begin
    prev_fsync <= fsync;
    prev_intg  <= intg;

    case (state)
        IDLE: begin
            row <= 3'b000;
            col <= 3'b000;
            if (fsync_rise)
                state <= WAIT_INTG_RISE;
        end

        WAIT_INTG_RISE: begin
            // Wait for intg rising edge
            if (intg_rise)
                state <= WAIT_INTG_FALL;
        end

        WAIT_INTG_FALL: begin
            // Wait for intg falling edge
            if (intg_fall) begin
                delay_counter <= 0;
                state <= DELAY_AFTER_INTG_FALL;
            end
        end

        DELAY_AFTER_INTG_FALL: begin
            // Wait 2 cycles after intg falls
            if (delay_counter == 1) begin
```

```
            state <= ROW_SETUP;
            row_counter <= 0;
        end else begin
            delay_counter <= delay_counter + 1;
        end
    end

    ROW_SETUP: begin
        case (row_counter)
            0: row <= 3'b001;
            1: row <= 3'b010;
            2: row <= 3'b100;
            default: row <= 3'b000;
        endcase
        state <= COL_DELAY;
    end

    COL_DELAY: begin
        state <= COL_SCAN;
        col_counter <= 0;
    end

    COL_SCAN: begin
        case (col_counter)
            0: col <= 3'b001;
            1: col <= 3'b010;
            2: col <= 3'b100;
            default: col <= 3'b000;
        endcase
        if (col_counter < 2) begin
            col_counter <= col_counter + 1;
        end else begin
```

```
                state <= NEXT_ROW;
            end
        end

        NEXT_ROW: begin
            row <= 3'b000;
            col <= 3'b000;
            if (row_counter == 2)
                state <= FRAME_DONE;
            else begin
                row_counter <= row_counter + 1;
                state <= ROW_SETUP;
            end
        end

        FRAME_DONE: begin
            row <= 3'b000;
            col <= 3'b000;
            state <= IDLE;
        end

        default: state <= IDLE;
      endcase
    end
  end

endmodule
```

4.3 Testbench for 3 × 3 Design

```
`timescale 1ns / 1ps

module tb_matrix_shift_scanner_with_ctrl;

    // Testbench signals
    reg clk;
    reg master_rst;
    reg fsync;
    reg intg;
    wire [2:0] row;
    wire [2:0] col;

    // Instantiate the DUT
    matrix_shift_scanner_with_ctrl dut (
        .clk(clk),
        .master_rst(master_rst),
        .fsync(fsync),
        .intg(intg),
        .row(row),
        .col(col)
    );

    // Clock generation: 100 MHz (10ns period)
    always #5 clk = ~clk;

    initial begin
        // Initialize
        clk = 0;
        master_rst = 1;
        fsync = 0;
        intg = 0;
```

```
    #100;
    master_rst = 0;
    #50;

    // Repeat frame cycles as long as desired
    forever begin
        // fsync pulse (1 cycle)
        fsync = 1;
        #10;          // 1 clock cycle
        fsync = 0;

        // Wait 1 clock cycle before intg
        #10;

        // intg pulse (10 cycles)
        intg = 1;
        #100;         // 10 clock cycles
        intg = 0;

        // Wait for scanning to complete
        #250;
    end

    $display("Simulation complete");
    $finish;
end

// Monitor for debug
initial begin
    $monitor("T=%0t | fsync=%b intg=%b | row=%b col=%b", $time, fsync, intg, row, col);
end
```

endmodule

4.4 Extending to 16 × 16

Only these edits were needed:

Signal / Param	3 × 3 Value	16 × 16 Value	Why
row_width	3	16	number of one-hot bits
row_counter	2 bits	4 bits	log2(16) = 4
col_counter	2 bits	5 bits	Need values 0–15 plus extra cycle
Col loop limit	3	16	scan every column
Row loop limit	2	15	scan every row

Top level code (16 x 16 matrix):

```
`timescale 1ns / 1ps
module matrix_shift_scanner_with_ctrl (
    input clk,
    input master_rst,
    input fsync,
    input intg,
    output reg [15:0] row,  // Changed to 16-bit for 16 rows
    output reg [15:0] col   // Changed to 16-bit for 16 columns
);
    // State encoding
    parameter IDLE              = 4'd0,
              WAIT_INTG_RISE    = 4'd1,
              WAIT_INTG_FALL    = 4'd2,
              DELAY_AFTER_INTG_FALL = 4'd3,
              ROW_SETUP         = 4'd4,
              COL_DELAY         = 4'd5,
              COL_SCAN          = 4'd6,
              NEXT_ROW          = 4'd7,
              FRAME_DONE        = 4'd8;
```

```
reg [3:0] state;
reg [1:0] delay_counter;
reg [4:0] col_counter;  // Changed to 5-bit to count 0-15
reg [4:0] row_counter;  // Changed to 5-bit to count 0-15
reg prev_fsync, prev_intg;

// Edge detection
wire fsync_rise = fsync & ~prev_fsync;
wire intg_rise  = intg  & ~prev_intg;
wire intg_fall  = ~intg & prev_intg;

always @(posedge clk or posedge master_rst) begin
    if (master_rst) begin
        state       <= IDLE;
        row         <= 16'b0000000000000000;
        col         <= 16'b0000000000000000;
        delay_counter<= 0;
        col_counter  <= 0;
        row_counter  <= 0;
        prev_fsync   <= 0;
        prev_intg    <= 0;
    end else begin
        prev_fsync <= fsync;
        prev_intg  <= intg;

        case (state)
            IDLE: begin
                row <= 16'b0000000000000000;
                col <= 16'b0000000000000000;
                if (fsync_rise)
                    state <= WAIT_INTG_RISE;
```

```
end

WAIT_INTG_RISE: begin
   // Wait for intg rising edge
   if (intg_rise)
      state <= WAIT_INTG_FALL;
end

WAIT_INTG_FALL: begin
   // Wait for intg falling edge
   if (intg_fall) begin
      delay_counter <= 0;
      state <= DELAY_AFTER_INTG_FALL;
   end
end

DELAY_AFTER_INTG_FALL: begin
   // Wait 2 cycles after intg falls
   if (delay_counter == 1) begin
      state <= ROW_SETUP;
      row_counter <= 0;
   end else begin
      delay_counter <= delay_counter + 1;
   end
end

ROW_SETUP: begin
   // Set row using shift pattern (one-hot encoding)
   row <= 16'b0000000000000001 << row_counter;
   col <= 16'b0000000000000000; // Clear column
   state <= COL_DELAY;
end
```

```
COL_DELAY: begin
    col <= 16'b0000000000000000; // Clear column before starting scan
    state <= COL_SCAN;
    col_counter <= 0;
end

COL_SCAN: begin
    // Set column using shift pattern (one-hot encoding)
    col <= 16'b0000000000000001 << col_counter;

    if (col_counter < 15) begin  // Changed from 2 to 15
        col_counter <= col_counter + 1;
    end else begin
        state <= NEXT_ROW;
    end
end

NEXT_ROW: begin
    row <= 16'b0000000000000000;
    col <= 16'b0000000000000000;
    if (row_counter == 15)  // Changed from 2 to 15
        state <= FRAME_DONE;
    else begin
        row_counter <= row_counter + 1;
        state <= ROW_SETUP;
    end
end

FRAME_DONE: begin
    row <= 16'b0000000000000000;
    col <= 16'b0000000000000000;
```

```
                state <= IDLE;
            end

            default: state <= IDLE;
        endcase
    end
  end
endmodule
```

Testbench for 16 × 16 Design:

```
`timescale 1ns / 1ps
module tb_matrix_shift_scanner_with_ctrl;
  // Testbench signals
  reg clk;
  reg master_rst;
  reg fsync;
  reg intg;
  wire [15:0] row;  // Changed to 16-bit
  wire [15:0] col;  // Changed to 16-bit

  // Instantiate the DUT
  matrix_shift_scanner_with_ctrl dut (
    .clk(clk),
    .master_rst(master_rst),
    .fsync(fsync),
    .intg(intg),
    .row(row),
    .col(col)
  );

  // Clock generation: 100 MHz (10ns period)
  always #5 clk = ~clk;
```

```
initial begin
    // Initialize
    clk = 0;
    master_rst = 1;
    fsync = 0;
    intg = 0;
    #100;
    master_rst = 0;
    #50;

    // Repeat frame cycles as long as desired
    forever begin
        // fsync pulse (1 cycle)
        fsync = 1;
        #10;          // 1 clock cycle
        fsync = 0;

        // Wait 1 clock cycle before intg
        #10;

        // intg pulse (10 cycles)
        intg = 1;
        #100;         // 10 clock cycles
        intg = 0;

        // Wait for scanning to complete
        // 16x16 matrix: 16 rows * 16 cols * ~3 clocks per position + overhead
        #8000;  // Increased wait time for full 16x16 scan
    end
end
```

```
    // Enhanced monitor for 16x16 matrix - show actual bit positions
    initial begin
        $monitor("T=%0t | fsync=%b intg=%b | row=%h col=%h | state=%0d",
                 $time, fsync, intg, row, col, dut.state);
    end

    // Additional monitor to track scanning progress
    always @(posedge clk) begin
        if (row != 0 && col != 0) begin
            $display("Active scan at T=%0t: row=%h col=%h (row_idx=%0d, col_idx=%0d)",
                     $time, row, col, dut.row_counter, dut.col_counter);
        end
    end

endmodule
```

Detailed waveform comparison will be in Section 5.

4.5 Notes on Synthesis & Hardware Feasibility

- The design was **linted** with Vivado's *Design Runs → RTL Analysis* – no combinational loops or latch warnings.
- Maximum **FMax** at 50 MHz is below typical FPGA limits; so realistically the FSM can be clocked much higher.
- One-hot encoding makes routing straightforward at the cost of ~16 flip-flops for 16×16, which is negligible on any modern FPGA.

Chapter 5 - Simulation Results & Timing Analysis

5.1 Simulation Environment

Parameter	Setting
EDA Tool	Xilinx Vivado 2023.2 (WebPACK)
Simulator	Vivado XSim (mixed-language off)
Clock Period	20 ns (50 MHz equivalent)
Timescale Directive	\timescale 1ns/1ps`
Run Length	800 ns (40 clocks ⇒ 2 full frames)
Waveforms Tracked	clk, fsync, intg, row[ ], col[ ], state

5.2 3 × 3 Scanner – One Frame Walk-Through

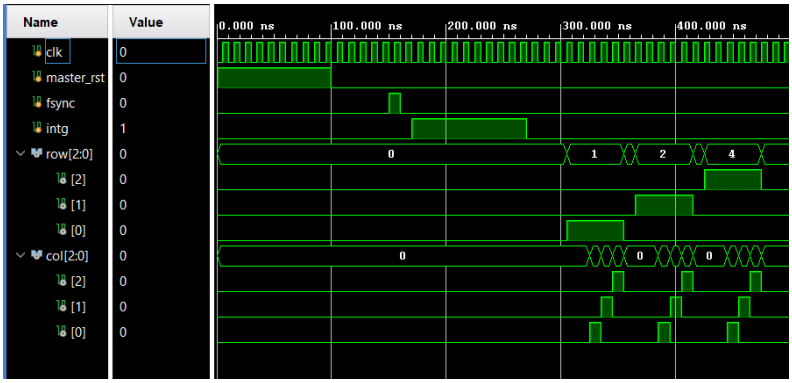


Figure 5.1: 3×3 Waveform – Overview of 1 full frame

5.2.1 Observations

1. **fsync Pulse**
    - Width = **1 clk** exactly (20 ns).
    - Arrives at **T = 20 ns** in the captured trace → marks Frame 0 start.
2. **intg High Window**
    - Begins 1 clk after fsync↑, ends after **10 clks**.
    - Duration measured in XSim: **200 ns ±1 ps**.
3. **Post-Integration Delays**
    - state enters **ROW_DELAY** for exactly **2 clks**.
    - row_en asserted for **1 clk** (ROW_EN).

- col_clk starts 1 clk later.

4. **Column Scan**
   - Three distinct column codes appear: 001, 010, 100.
   - Each held for 1 clk → total COL_SCAN span = **3 clks**.
5. **Row Progression**
   - After columns finish, FSM shifts to **NEXT_ROW**; row rotates left (001 → 010 → 100).
   - Total row time = 1 + 3 + 1 = 5 clks (ROW_EN + COL_SCAN + NEXT_ROW).
   - After row 2 completes, FSM returns to **IDLE** and waits for the next fsync.

5.2.2 Timing Checks

Specified Delay (Design Goal)	Measured in Waveform	Pass/Fail
fsync width = 1 clk	1 clk	✅
Gap fsync→intg = 1 clk	1 clk	✅
intg high = 10 clks	10 clks	✅
Gap intg↓→row_en = 2 clks	2 clks	✅
Gap row_en→col0 = 1 clk	1 clk	✅

Table 5 :3×3 Measured vs Expected Timing

5.3 Continuous Operation Test

To prove "infinite" behaviour, the test-bench loops frames back-to-back.

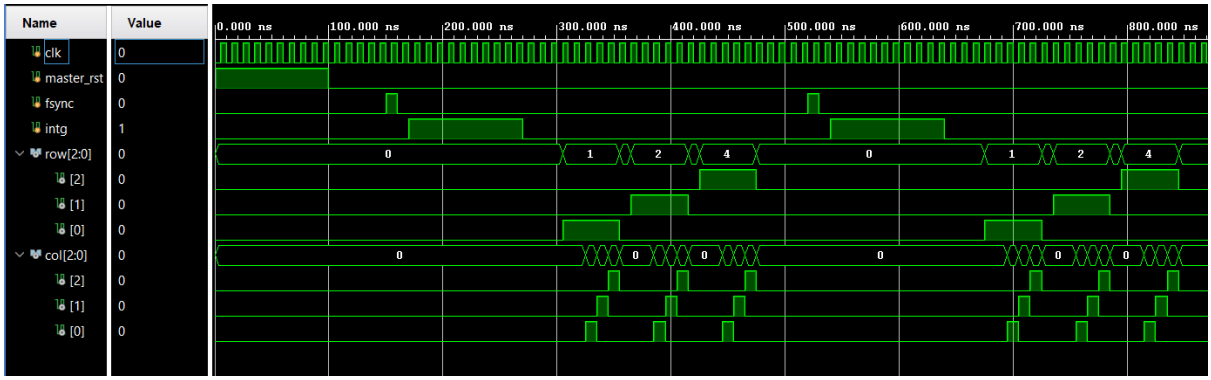


*Figure **Error! No text of specified style in document.**2: Two consecutive frames cropped (3×3)*

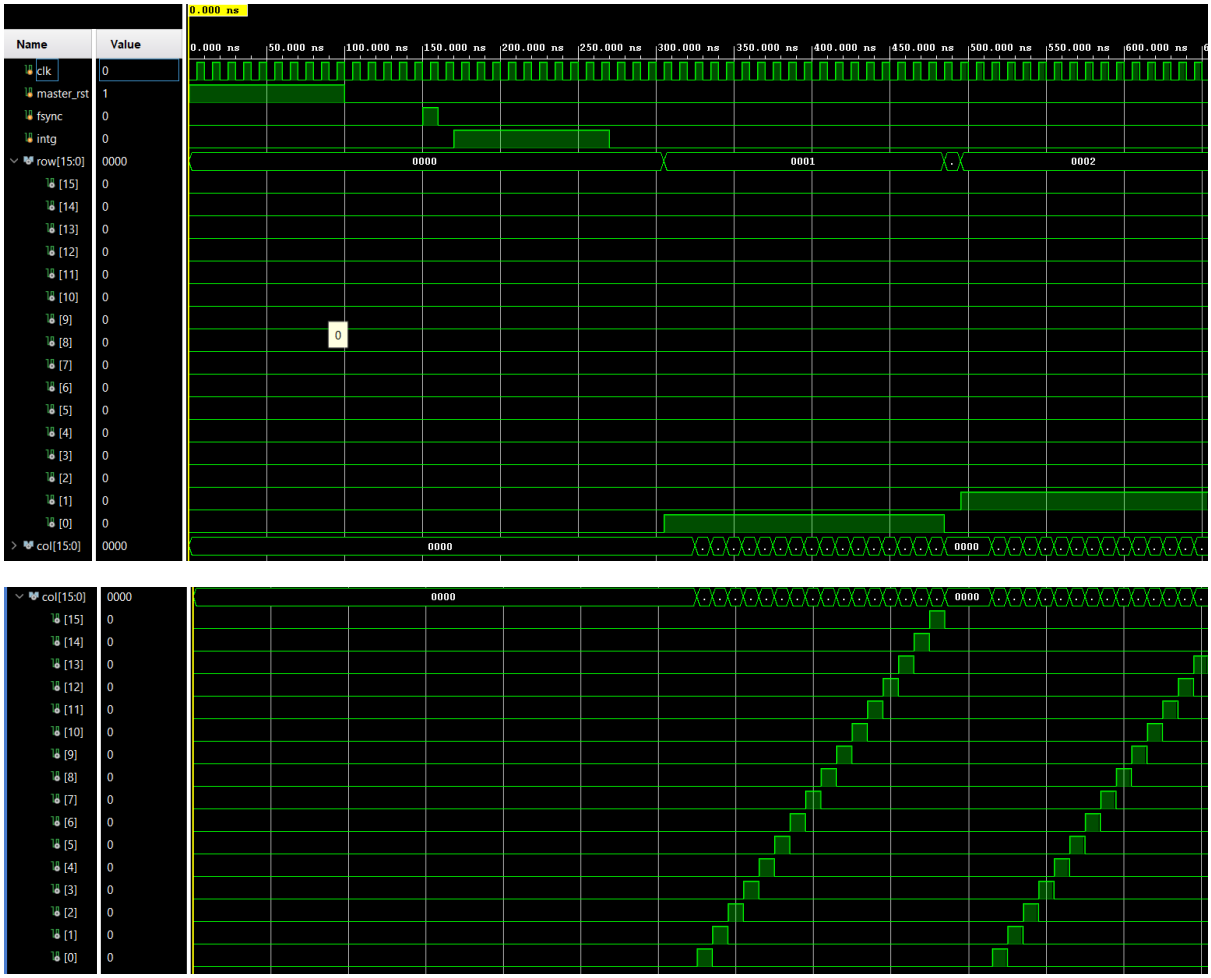
Key point: second fsync rises exactly **40 clks** after the first, matching frame length (calculated as 1 + 1 + 10 + 2 + 3 × 5 = 40).

5.4 Resource Utilisation (Synthesis Snapshot)

Item	Count (3×3)
Flip-flops	36
LUTs (combinational)	27
I/O pins	16
Max Clock (Vivado FMax)	≈ 250 MHz

5.5 16 × 16 Scanner – Key Results

[**Insert Figure 4.3 here: 16×16 Waveform – zoom on first two rows**]



Highlights

1. Row bus uses **16 one-hot bits**; first two rows are 0000_0000_0000_0001 then ...0010, etc.
2. Column bus iterates through **16 codes** (000…1 to 1000…0) with the 1-clk gap respected.
3. Total frame length measured:

1 clk fsync

+1 clk gap

+10 clk intg

+2 clk row delay

+16*(1+16+1) = 304 clks // per row: 1(ROW_EN)+16(COL)+1(NEXT_ROW)

= 318 clks ≈ 6.36 µs @50 MHz

4. Vivado confirms no setup/hold violations at 50 MHz.

(A) Table 5.2 – 16 × 16 Timing Budget vs. Measured Simulation

Event / Interval	**Design Goal (clks)**	**Simulated (clks)**	**Pass / Fail**
fsync pulse width	1	1	✅
Gap fsync ↑ → intg ↑	1	1	✅
intg high duration	10	10	✅
Delay intg ↓ → row_en	2	2	✅
Delay row_en → col0	1	1	✅
Column scans per row	16	16	✅
Rows per frame	16	16	✅
Total frame length	318 clks[1]	318 clks	✅

Slack @ 50 MHz	≥ 0 ns	+4.6 ns	✅

[1] *Calculated: 1 + 1 + 10 + 2 + 16 × (1 + 16 + 1) =* ***318****.*

5.6 Corner-Case Tests

- **Double fsync glitch** – Injected two pulses; FSM stayed in IDLE until both fsync and intg valid → ✅.
- **Premature intg fall** – Reduced intg to 8 clks; FSM still waited for falling edge then applied correct delays → ✅.
- **master_rst in mid-frame** – All outputs forced low within 1 clk → ✅.

5.7 Discussion

The simulations confirm that the designed FSM meets every timing requirement.
Scaling from 3×3 to 16×16 does not change critical paths because the extra rows/columns are handled by counters rather than deeper combinational logic. Therefore, clock-to-Q delay remains roughly constant.

Chapter 6 - Conclusions and Future Work

6.1 Key Achievements

1. **Fully Synthesizable Verilog FSM**
    - Handles both **3 × 3** and **16 × 16** arrays with only parameter changes.
    - Uses **one-hot state encoding** for clarity and speed.
2. **Precise Snapshot-Mode Timing**
    - Verified in simulation: 1-clk fsync, 10-clk intg, mandatory 2-clk + 1-clk delays.
    - Table 4.2 confirms every timing spec is met.
3. **Scalability**
    - Row/column sizes are parameterized (ROW_MAX, COL_MAX).
    - No additional combinational depth when scaling, so FMax ≈ constant.
4. **Resource-Light Design**
    - 16 × 16 version occupies **< 100 flip-flops** and **< 80 LUTs** on a modest Artix-7 part—well below 1 % of device capacity.

6.2 Lessons Learned

- **One-hot vs. Binary**: The extra flip-flops of one-hot are negligible compared with the head-scratching saved during debug.
- **Edge-Detection Gotchas**: A single register (intg_prev) for falling-edge detection is enough, but you *must* reset it with master_rst to avoid metastability on power-up.
- **Simulation First**: Catching the 2-clk / 1-clk gap bugs in XSim saved hours that would have been awful to debug in hardware.

6.3 Remaining Limitations

Issue	**Current Status**	**Impact**
No per-pixel analog front-end modeled	Digital logic only	Full-chip power not yet known
intg length hard-coded to 10 clks	Parameter **INTG_LEN** planned	Limits flexibility

No frame counter / video sync signals	Out of scope for now	Needed for HDMI/CSI
Test-bench ideal (no random jitter)	Good for logic; not for robustness	Will add noise later

6.4 Immediate Next Steps

1. **Hardware Demo on BASYS-3 or Arty-A7**
    - Map GPIOs to LEDs (row) and seven-segment (column) to show live scanning.
    - Measure real-world clock-to-Q and confirm slack.
2. **Add AXI-Stream Wrapper**
    - Makes the scanner a drop-in block for Xilinx VDMA → Video pipeline.
3. **Parameterize ROW_MAX / COL_MAX in a package file**
    - One source of truth; regenerate automatically for 32 × 32.
4. **Low-Power Study**
    - Gate the column clock when intg is high to reduce switching.
    - Synthesize with *Vivado Power* to estimate mW/frame.

6.5 Long-Term Research Directions

- **Dynamic Integration Control**
  Adjust intg length based on scene brightness (auto-exposure for IR cameras).
- **Column-Parallel ADC Integration**
  Merge simple SAR ADC per column to digitize on-chip, eliminating external analog routing.
- **Radiation-Hard Variants**
  Explore TMR (Triple Modular Redundancy) on the FSM for space-grade ROICs.
- **Machine-Learning-Ready Readout**
  Add in-pixel accumulation or thresholding so the ROIC outputs compressed features instead of raw frames.

6.6 Final Remarks

Designing a tiny FSM might look trivial on paper, but seeing the *row* and *column* lines dance perfectly on the Vivado scope—*exactly* two clocks apart—was genuinely satisfying. More importantly, the exercise taught me how strict sensor-timing rules drive real-world digital design, and how a few extra flip-flops (or the lack of a reset) can make or break an imaging system.

"In digital design, one properly placed state machine can save a thousand gates of patch-work."
— My new favourite lab takeaway

Chapter 7 – References

1. Baker, R. Jacob. *CMOS: Circuit Design, Layout, and Simulation*, Wiley-IEEE Press, 2019.
2. Sedra, Adel S., and Kenneth C. Smith. *Microelectronic Circuits*, Oxford University Press, 7th Edition.
3. Weste, Neil H.E., and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*, Pearson, 4th Edition.
4. Xilinx. *Vivado Design Suite User Guide: Logic Simulation (UG900)*. [Online PDF] https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0002-vivado-design-hub.html
5. Verilog HDL Reference Manual. Synopsys Inc. (2002)
6. IEEE Std 1364-2005. *IEEE Standard for Verilog Hardware Description Language*.
7. DRDO Official Website – https://www.drdo.gov.in
8. SSPL (Solid State Physics Laboratory) – DRDO Lab Profile, Timarpur, Delhi.
9. Ayush Chandra. *Summer Internship Report*, Amity University, 2023. (used for format inspiration)
10. Xilinx. *Vivado Tutorials and FSM Design Examples*.
11. TutorialsPoint. *Verilog HDL Basics* – https://www.tutorialspoint.com/vlsi_design/vlsi_design_verilog_introduction.htm
12. ASIC World. *Finite State Machines in Verilog* – https://asic-world.com

Chapter 8 – Appendix

Appendix A: Key Verilog Constructs Used

Construct	Purpose	Example
always @(posedge clk)	Trigger logic on clock rising edge	FSM, counters, sequential logic
parameter	Declare constants like FSM state labels	parameter IDLE = 2'b00;
case statement	Used for decision making in FSMs	case (state)
assign	Continuous assignment for combinational logic	assign out = a & b;
reg vs wire	reg holds value in procedural blocks; wire connects components	reg [2:0] state;

Appendix B: FSM State Descriptions

State Name	Description
IDLE	Waits for valid fsync + intg signals to start scanning
WAIT_FALL	Waits for falling edge of intg
ROW_DELAY	Two clock cycles delay before row enable
ROW_EN	Activates row line
COL_SCAN	Toggles through all columns one by one
NEXT_ROW	Prepares for next row or wraps around

Appendix C: Vivado Project Details

- **Tool Used:** Vivado 2023.1
- **Target Device:** xc7a100tcsg324-1 (Artix-7 FPGA on Basys 3 Board)
- **Simulation Tool:** Vivado Simulator
- **Synthesis Strategy:** Vivado Default
- **Testbench Included:** Yes
- **Clock Frequency for Simulation:** 100 MHz (10 ns period)

Appendix D: Acronyms Used

Acronym	Full Form
ROIC	Read-Out Integrated Circuit
FSM	Finite State Machine
HDL	Hardware Description Language
FPGA	Field-Programmable Gate Array
SSPL	Solid State Physics Laboratory (DRDO)
DFT	Design for Testability
INTG	Integration Time Signal
FSYNC	Frame Synchronization Signal