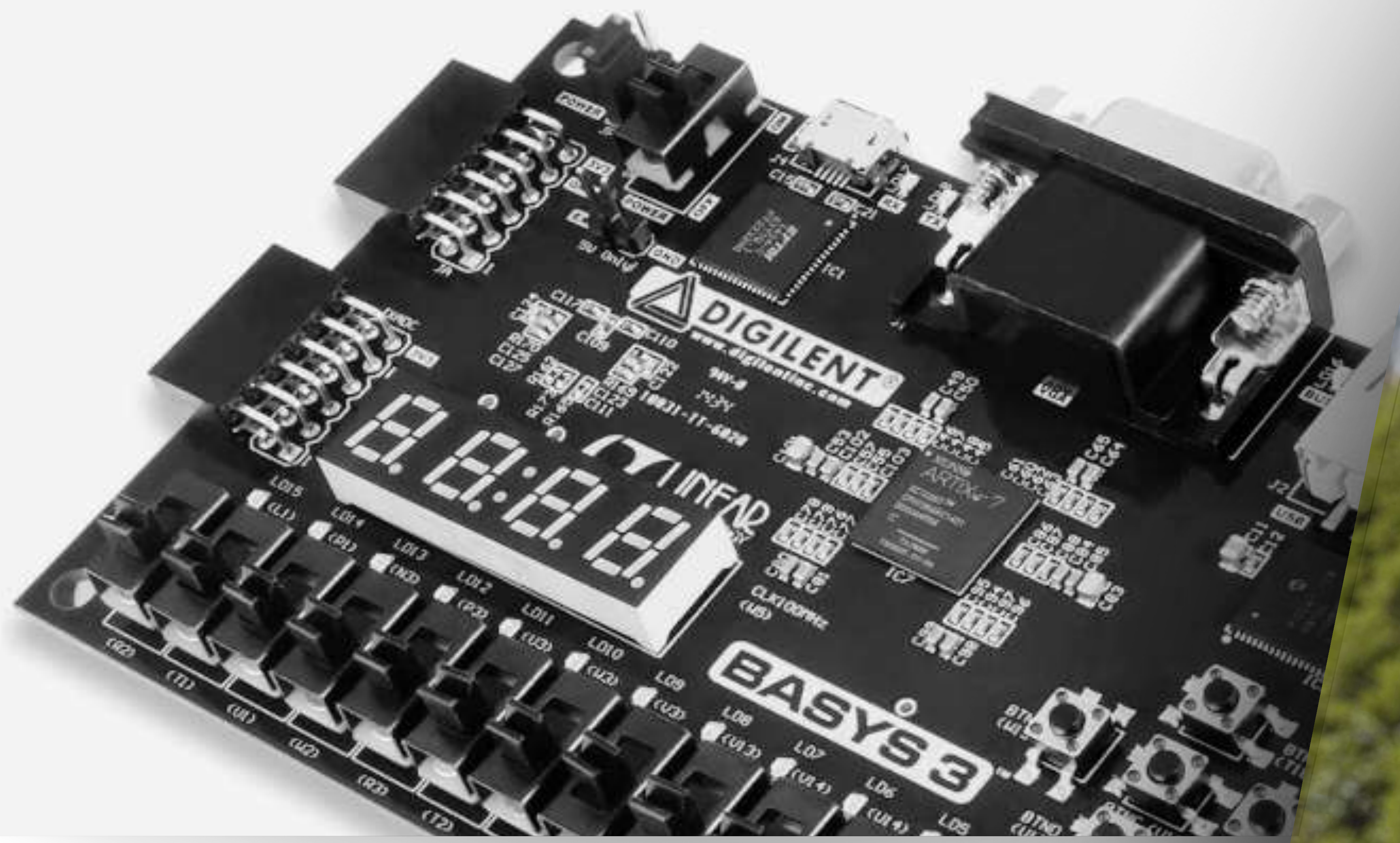


SPI PROJECT

Silicon Service

Our team

Andrei Popa

Student

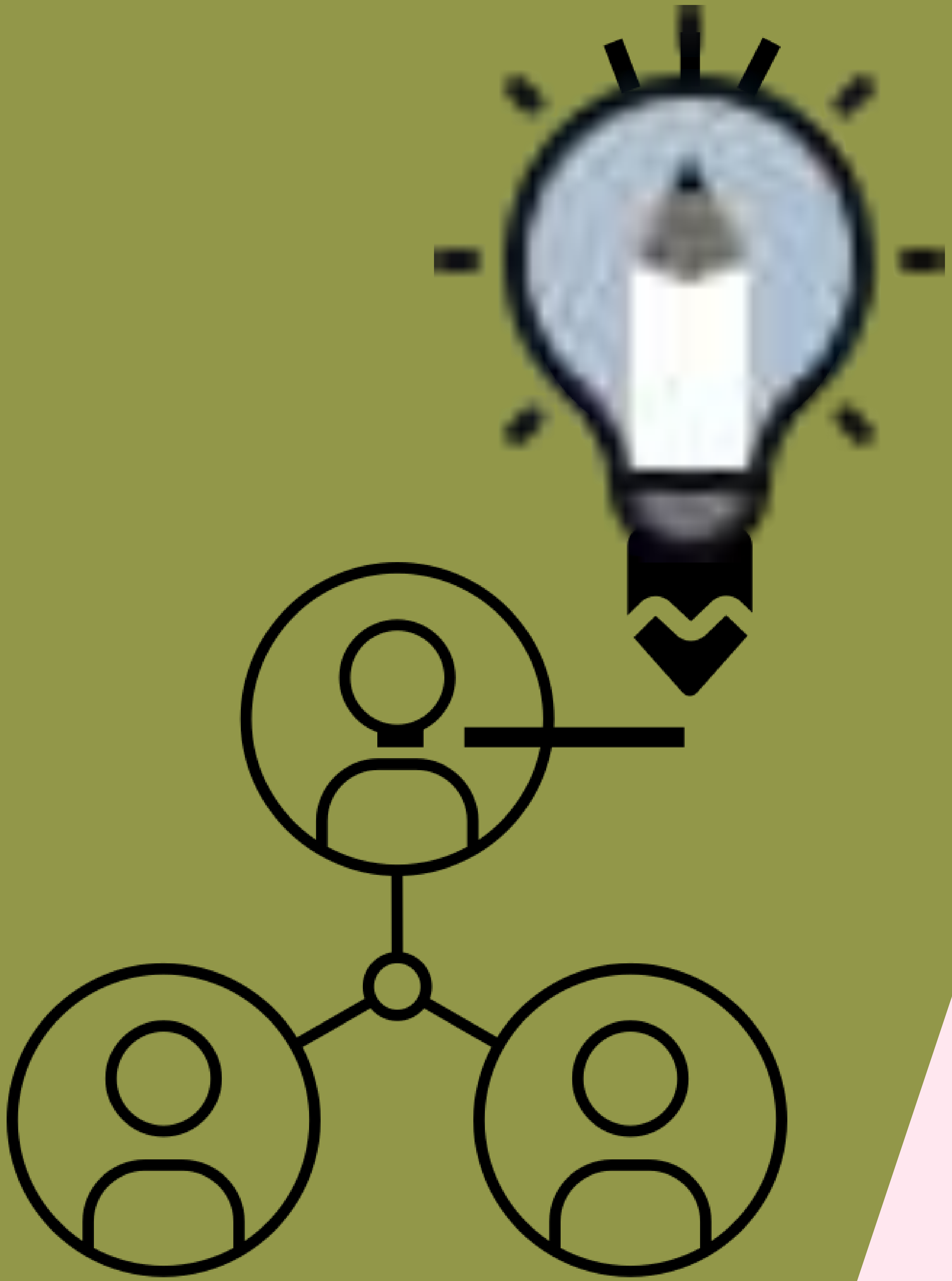
Mădălina Mihălucă

Student

Alexandru Antoci

Student

Table of Contents

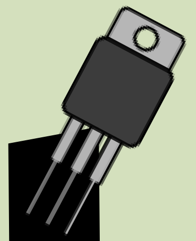


WHAT IS A SPI?

SPI, or Serial Peripheral Interface, is a communication protocol used to transfer data between a microcontroller and peripheral devices. It enables fast, synchronous data exchange using four main signals:

- **MOSI (Master Out Slave In): Sends data from the master device to the slave.**
- **MISO (Master In Slave Out): Sends data from the slave device back to the master.**
- **SCK (Serial Clock): Provides the clock signal from the master to synchronize data transfer.**
- **SS/CS (Slave Select/Chip Select): Selects the slave device that the master communicates with.**

Real World Applications of SPI in Engineering



Smartphones

SPI is commonly used to connect the main processor with various peripherals such as touchscreens, fingerprint sensors, and memory modules.

Gaming consoles

The PlayStation 5's DualSense controller utilizes SPI to relay data from the controller to the console, enabling responsive haptic feedback and adaptive trigger functionalities.

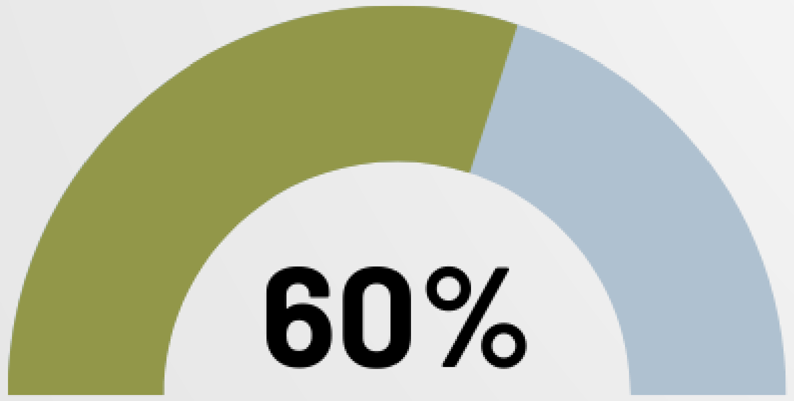
Smart home Devices

Smart home devices also benefit from SPI's robust communication capabilities. Smart thermostats, security cameras, and home automation hubs often use SPI to facilitate communication between the central processor and various peripheral components.

About Vivado

Vivado is a software program developed by Xilinx for designing and programming programmable logic circuits, such as FPGAs. It is used by engineers to create, simulate, verify, and implement complex digital designs on hardware.

The main languages supported by Vivado are: VHDL, Verilog, SystemVerilog, C/C++.



🟩 Used for FPGA Design and Implementation

🟦 Used for Embeded Systems

VIVADO WORKFLOW

Simulation

Simulation means testing and verifying the digital design, without actually building the physical circuit, to ensure it functions as expected before its real-world implementation.

Synthesis

Synthesis identifies synthesis errors and transforms the design described in a hardware description language into a concrete physical circuit. .

Implementation

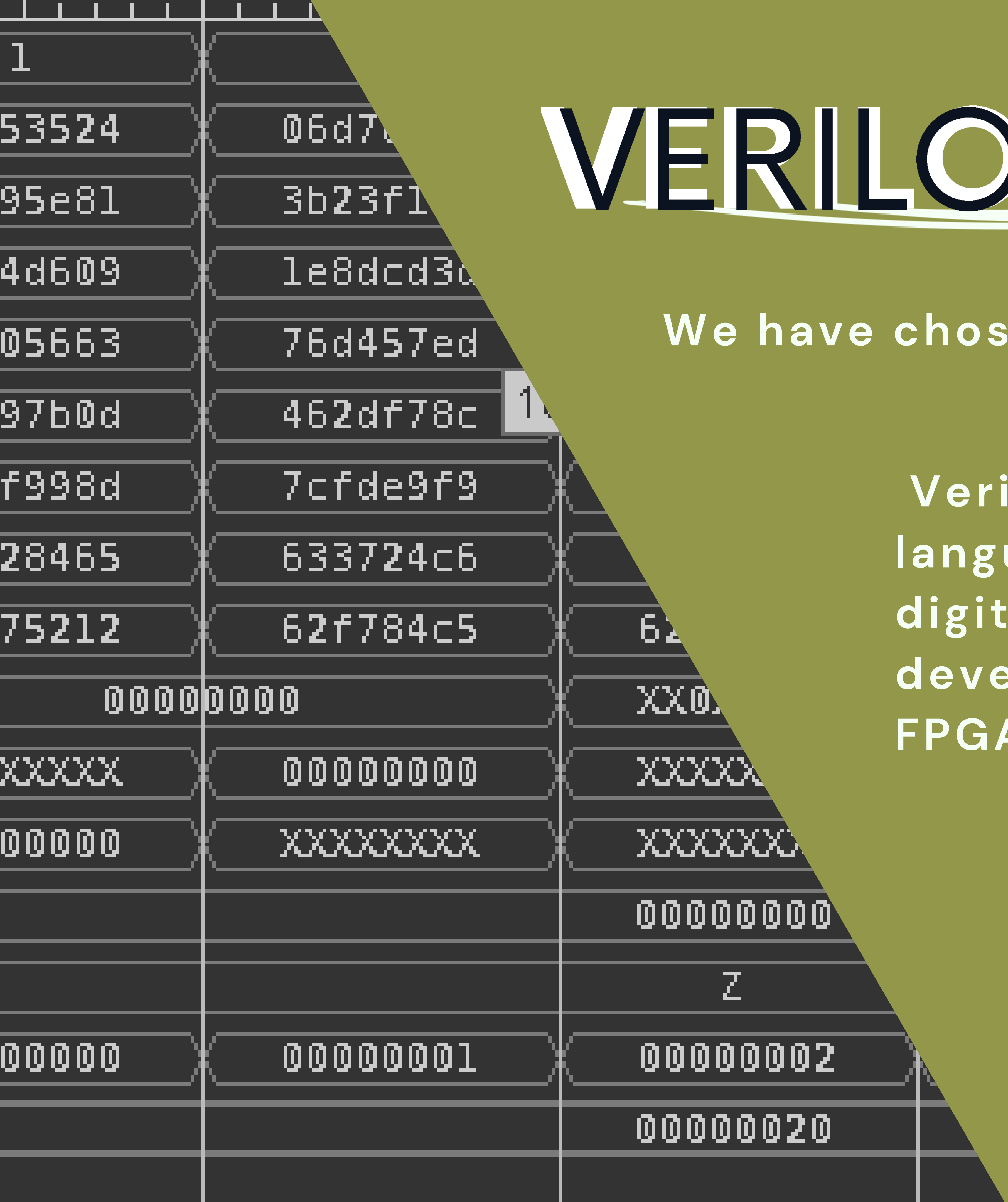
This step includes placing and routing the circuit generated by synthesis, ensuring that the design aligns with the available hardware resources.

Generate Bitstream

Generate Bitstream refers to the process of creating a bitstream file from a hardware design, which is then used to program a programmable logic device such as an FPGA.

Open Hardware manager

This step establishes the connection to the hardware resource, in our case, the Basys 3 FPGA.



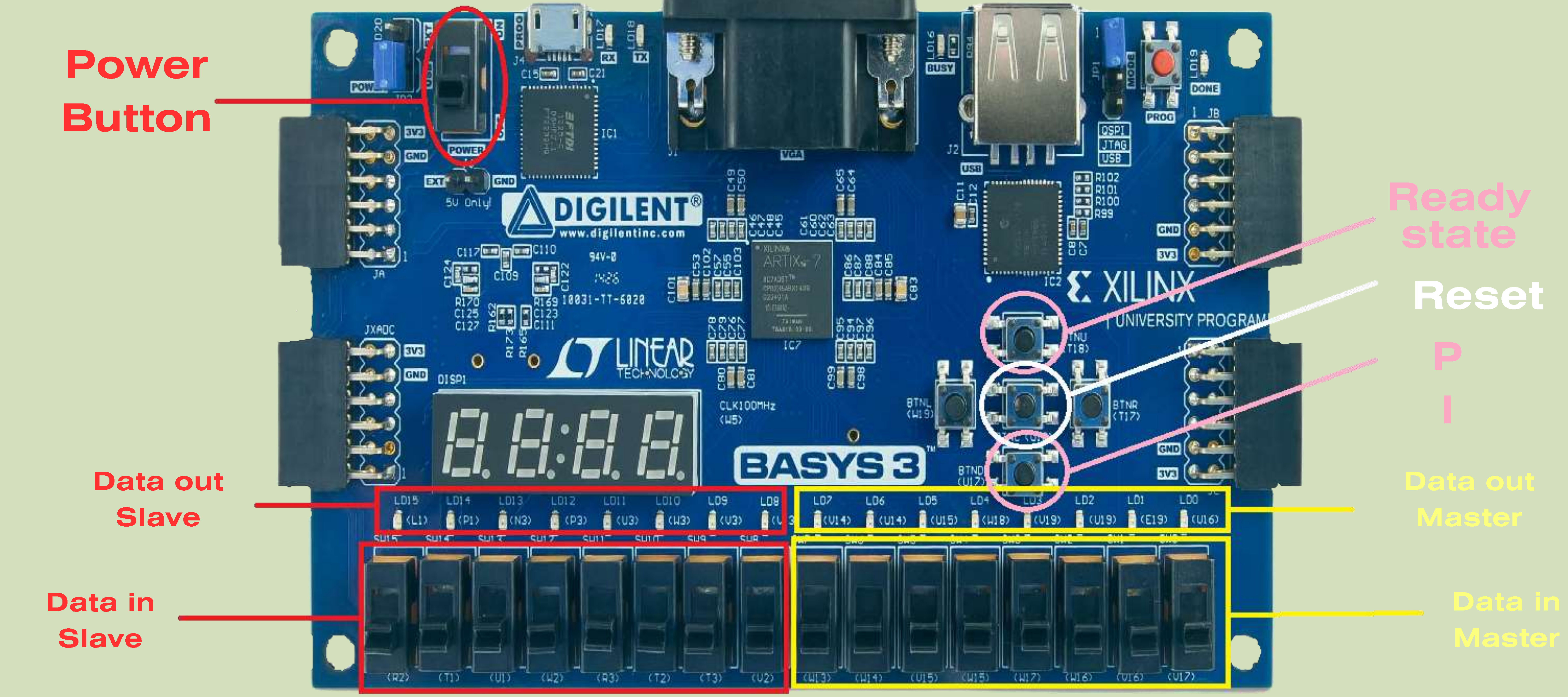
VERILOG

We have chosen to implement the project using the Verilog.

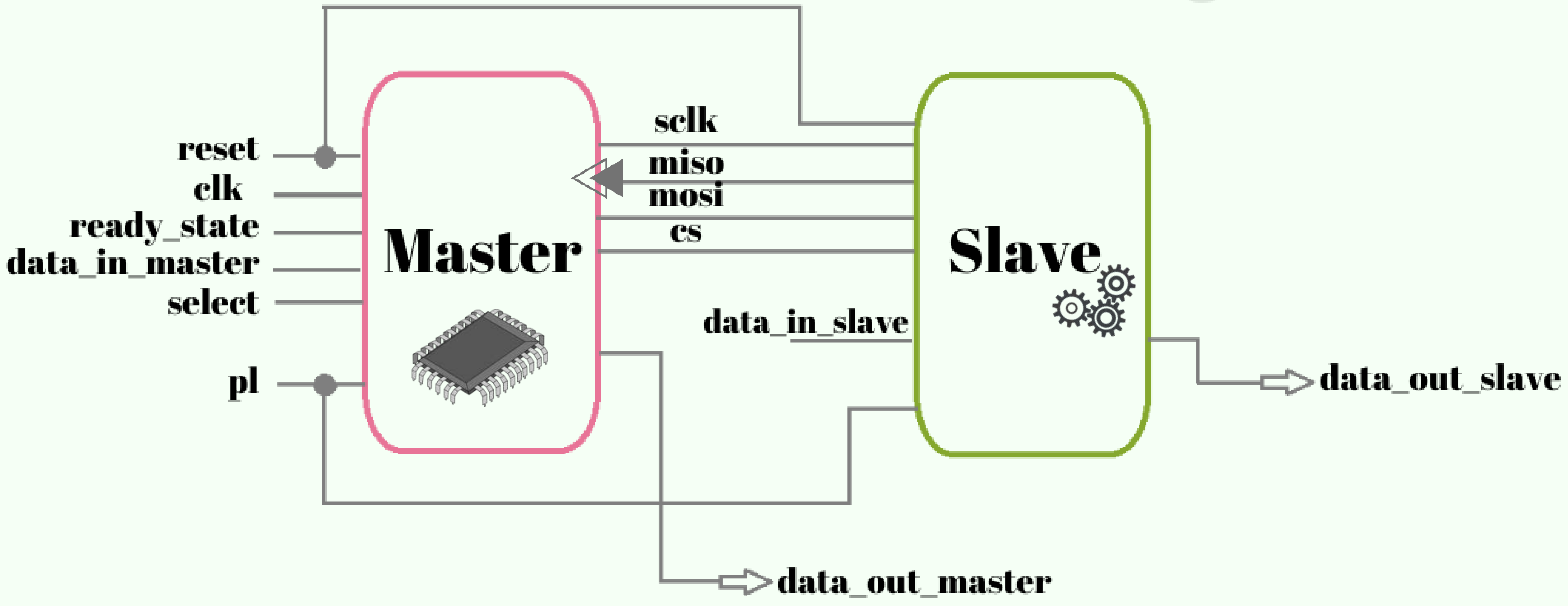
Verilog is a hardware description language used to model and simulate digital circuits. It is widely used in the development of integrated systems and FPGA design.

Verilog allows for the description of both the behavior and the structure of circuits, providing a way to describe and simulate the circuit before physically implementing it.

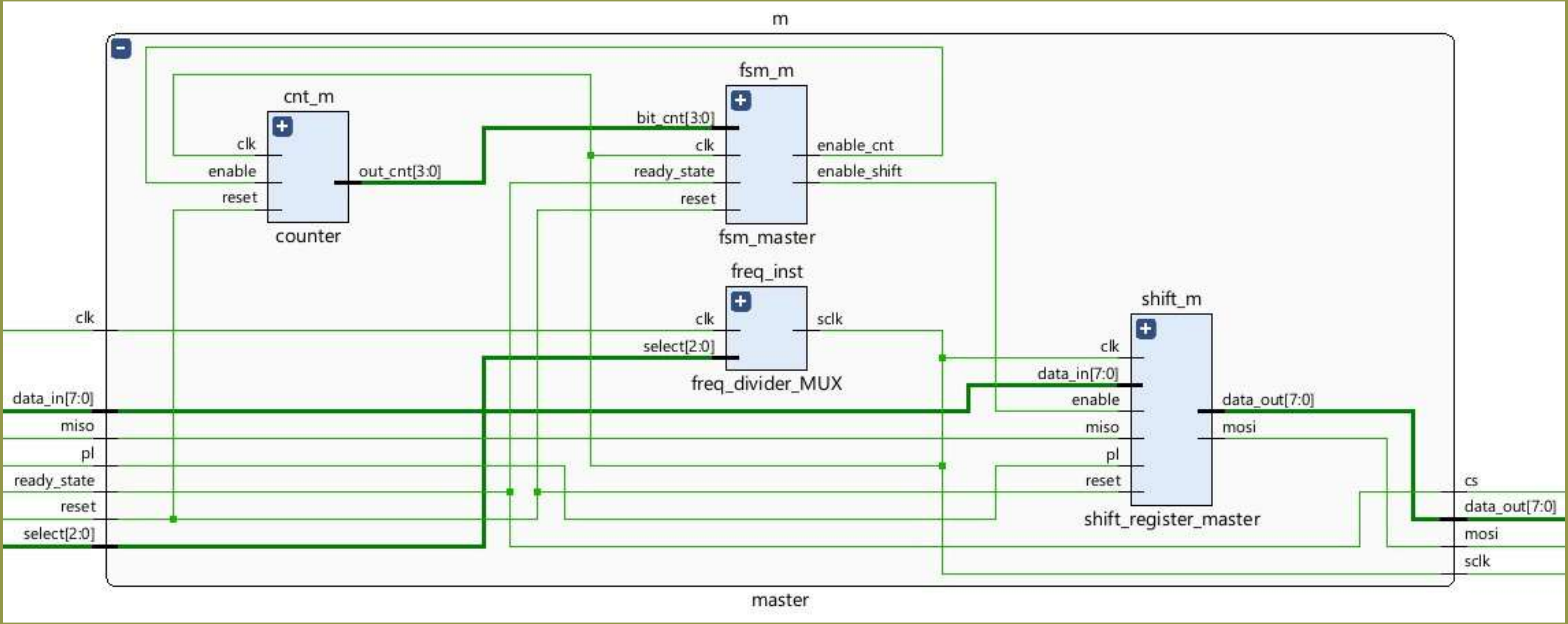
BASYS 3 BOARD



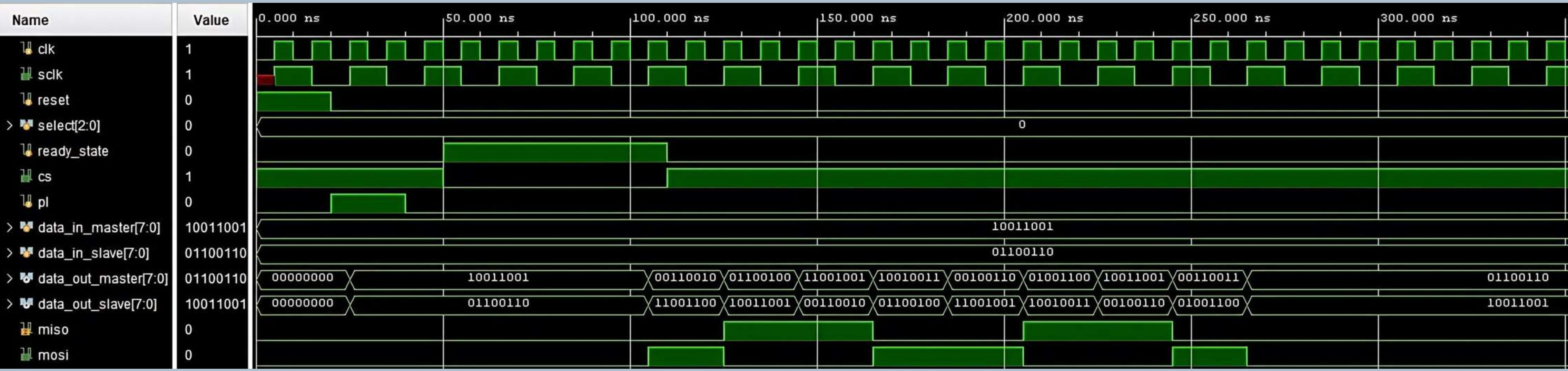
BLOCK DIAGRAM



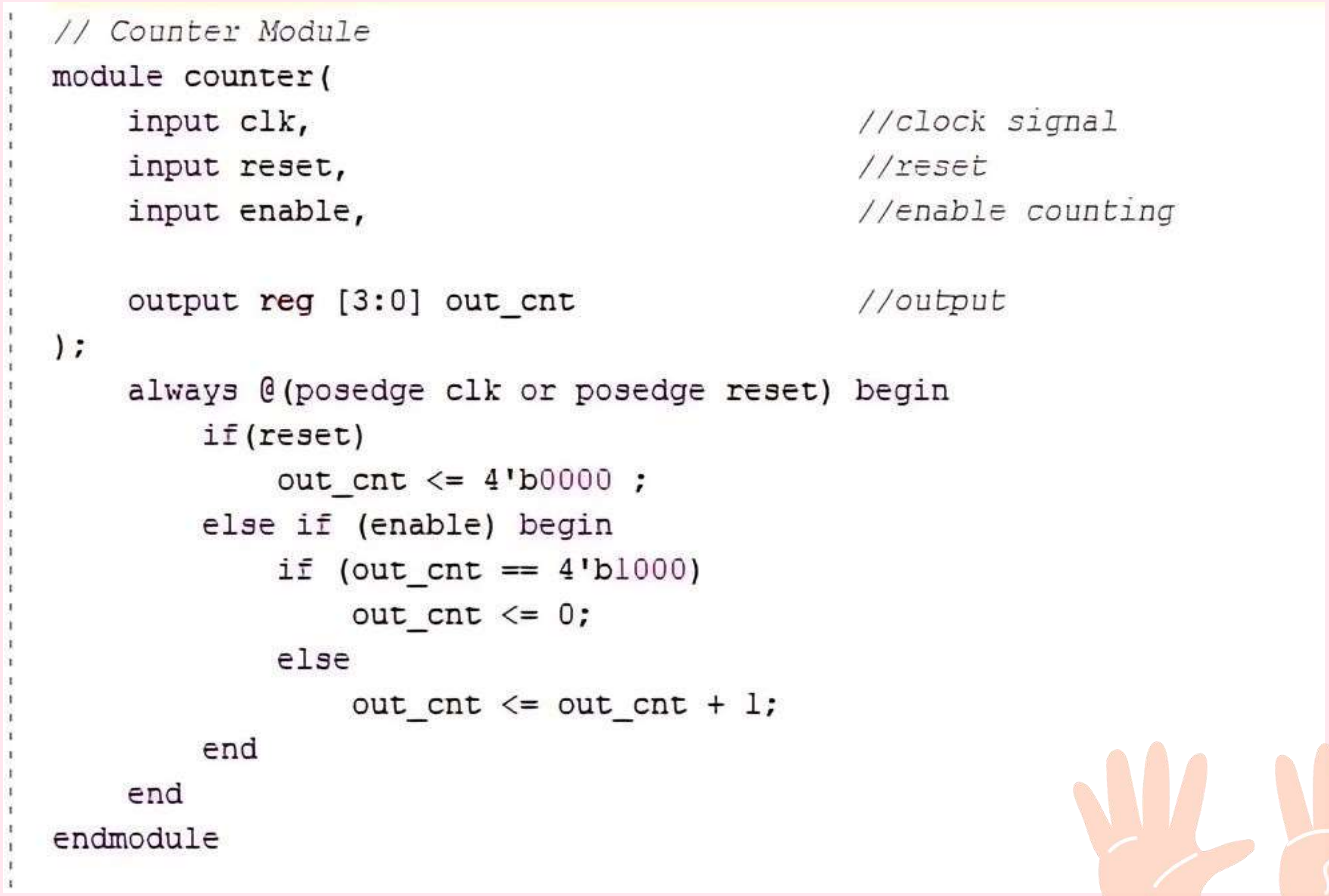
MASTER INTERNAL STRUCTURE



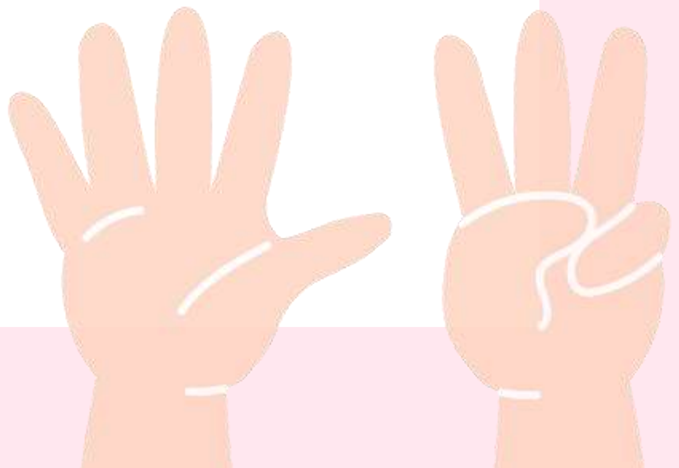
SIMULATION WAVEFORM



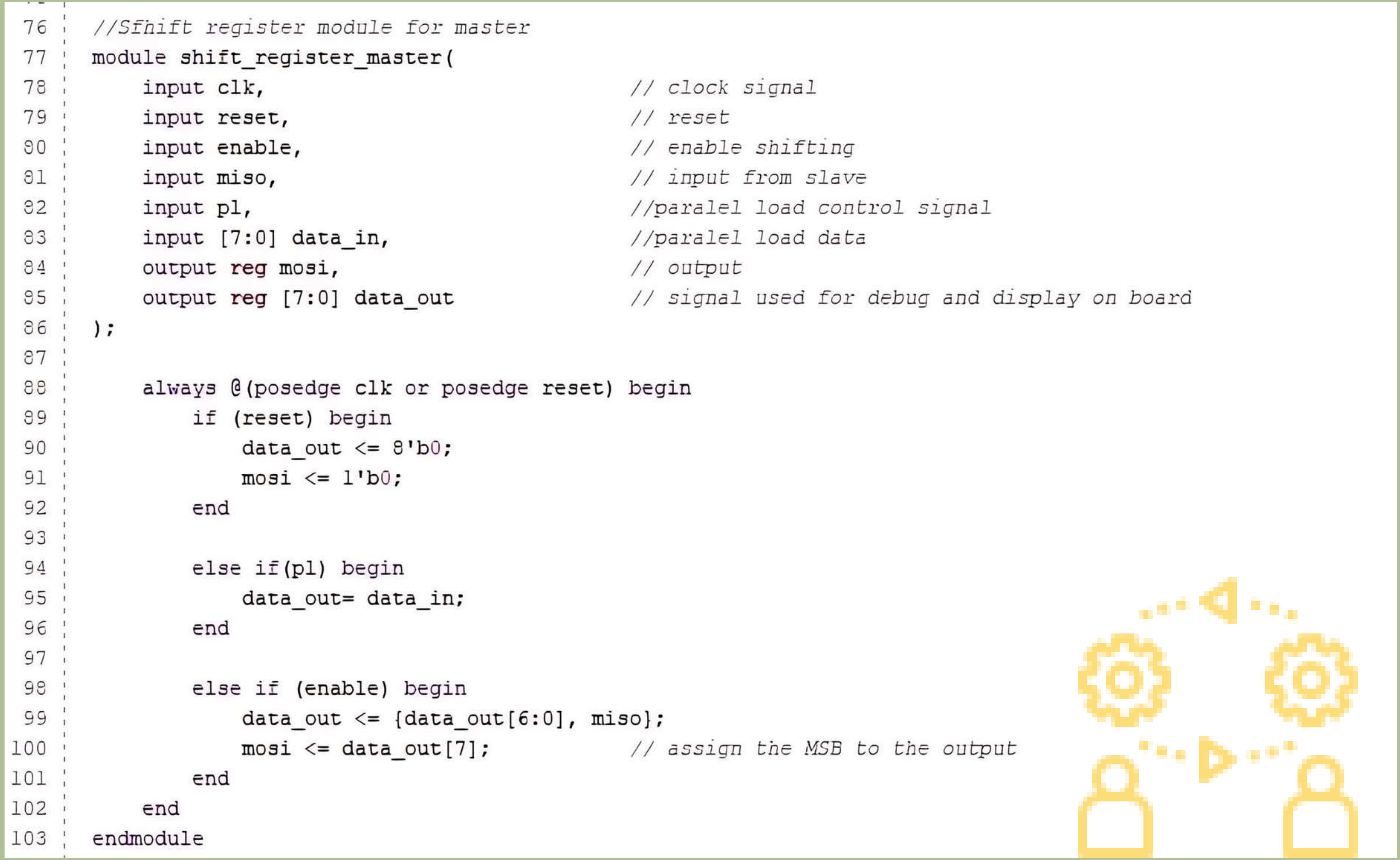
Counter



```
// Counter Module
module counter(
    input clk,                               //clock signal
    input reset,                             //reset
    input enable,                            //enable counting

    output reg [3:0] out_cnt                 //output
);
    always @(posedge clk or posedge reset) begin
        if(reset)
            out_cnt <= 4'b0000 ;
        else if (enable) begin
            if (out_cnt == 4'b1000)
                out_cnt <= 0;
            else
                out_cnt <= out_cnt + 1;
        end
    end
endmodule
```

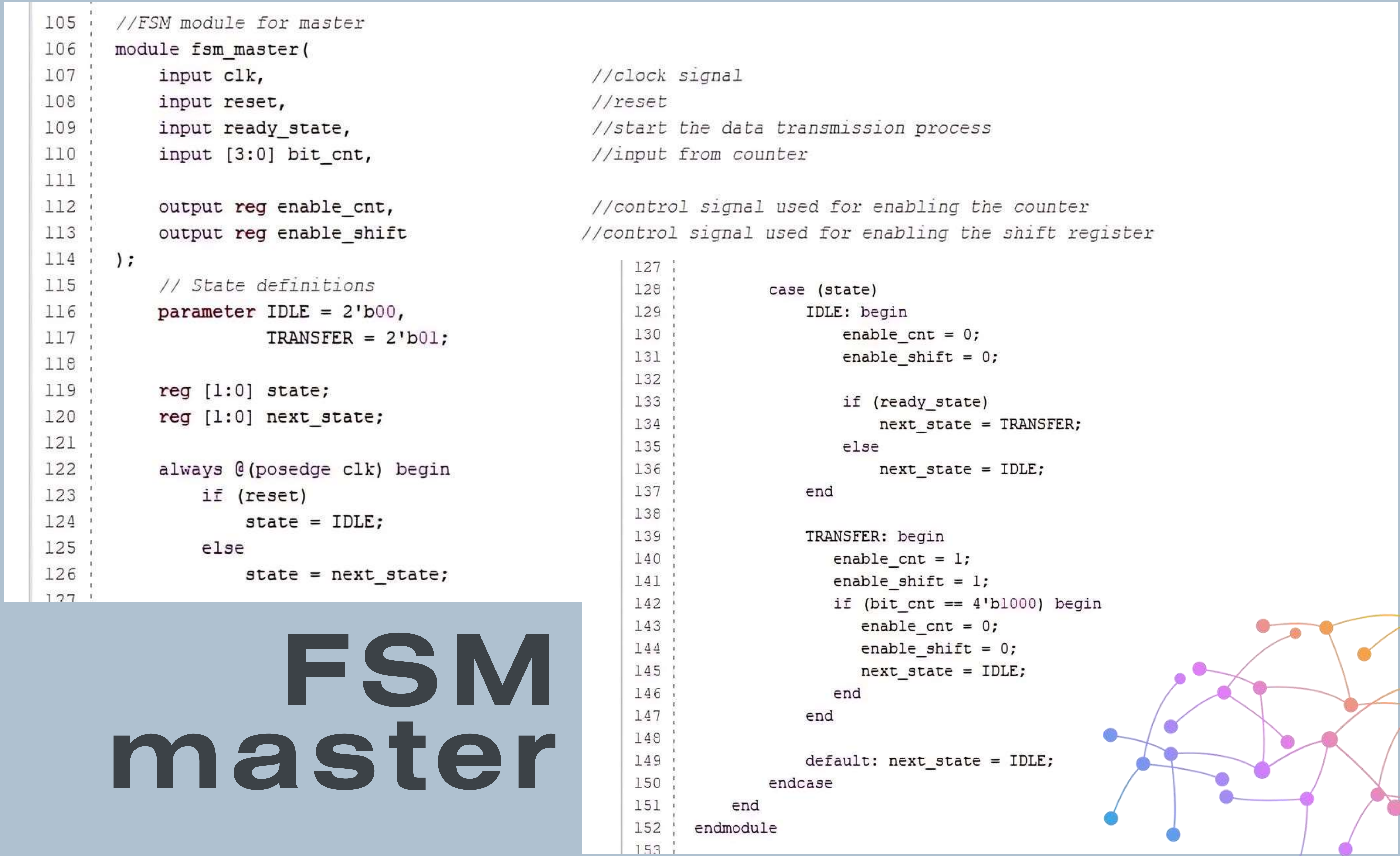


Shift Register



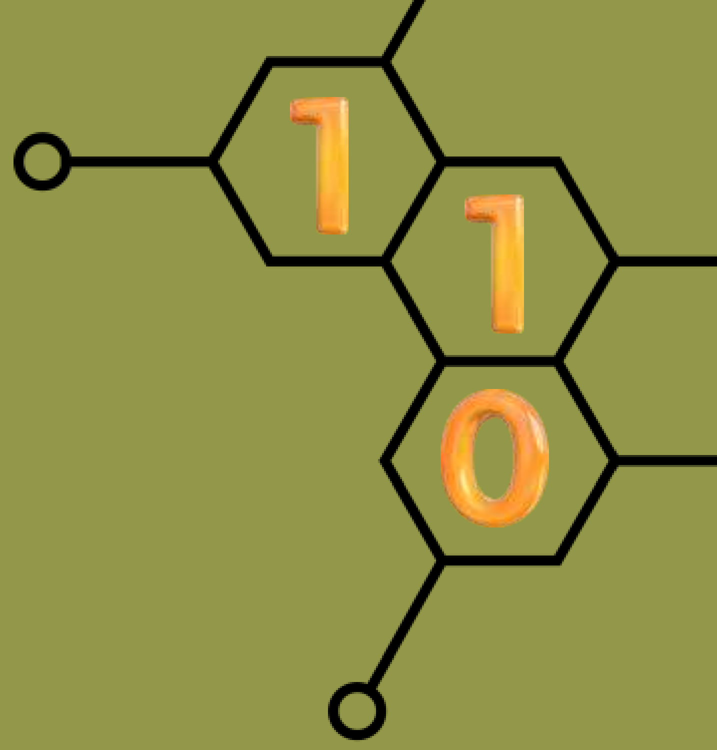
```verilog
//Sfhift register module for master
module shift_register_master(
    input clk,                             // clock signal
    input reset,                           // reset
    input enable,                          // enable shifting
    input miso,                            // input from slave
    input pl,                              //paralel load control signal
    input [7:0] data_in,                   //paralel load data
    output reg mosi,                       // output
    output reg [7:0] data_out              // signal used for debug and display on board
);

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            data_out <= 8'b0;
            mosi <= 1'b0;
        end

        else if(pl) begin
            data_out= data_in;
        end

        else if (enable) begin
            data_out <= {data_out[6:0], miso};
            mosi <= data_out[7];           // assign the MSB to the output
        end
    end
endmodule
```

FSM master



```
//FSM module for master
module fsm_master(
    input clk,                                  //clock signal
    input reset,                                //reset
    input ready_state,                          //start the data transmission process
    input [3:0] bit_cnt,                        //input from counter

    output reg enable_cnt,                      //control signal used for enabling the counter
    output reg enable_shift                    //control signal used for enabling the shift register
);
    // State definitions
    parameter IDLE = 2'b00,
              TRANSFER = 2'b01;

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

    always @(posedge clk) begin
        if (reset)
            state = IDLE;
        else
            state = next_state;

            case (state)
                IDLE: begin
                    enable_cnt = 0;
                    enable_shift = 0;

                    if (ready_state)
                        next_state = TRANSFER;
                    else
                        next_state = IDLE;
                end

                TRANSFER: begin
                   enable_cnt = 1;
                   enable_shift = 1;
                   if (bit_cnt == 4'b1000) begin
                      enable_cnt = 0;
                      enable_shift = 0;
                      next_state = IDLE;
                   end
                end

                default: next_state = IDLE;
            endcase
        end
endmodule
```

BIBLIOGRAPHY



https://forum.digikey.com/t/implementing-a-robust-microcontroller-to-fpga-spi-interface-part-2-protocol-definition

https://www.fpga4student.com/p/fpga-projects.html

https://www.xilinx.com/products/boards-and-kits/1-54wqge.html

https://chatgpt.com

THANK YOU