Digital Design Lab Manual - SystemVerilog

Table of Contents

1. Introduction to Digital Design Methodology

1.1 Design Flow Overview

The modern digital design flow follows a systematic approach:

1. **Specification** - Define requirements and interface specifications
2. **Architecture** - High-level block diagram and data flow
3. **RTL Design** - Register Transfer Level implementation
4. **Simulation** - Functional verification (covered next week)
5. **Synthesis** - Logic synthesis for target technology
6. **Implementation** - Place and route for FPGA/ASIC
7. **Timing Analysis** - Static timing analysis and closure
8. **Validation** - Hardware validation and testing

1.2 Design Principles

- **Modularity**: Break complex designs into smaller, manageable modules
- **Hierarchy**: Use hierarchical design approach
- **Synchronous Design**: Use single clock domain when possible
- **Reset Strategy**: Implement proper reset methodology
- **Clock Domain Crossing**: Handle CDC carefully
- **Resource Optimization**: Consider FPGA/ASIC resources

1.3 Design Steps for Each Lab

Before writing any code, follow these essential steps:

1. **Read and understand the specification completely**
2. **Draw block diagrams showing all inputs/outputs**
3. **Create state diagrams (for sequential circuits)**
4. **Write truth tables (for combinational logic - if possible)**
5. **Optimize logic on paper using K-maps or Boolean algebra**
6. **Plan your module hierarchy**
7. **Define interfaces and data types**
8. **Code incrementally and test each piece**

1.4 Documentation Requirements

Every design must include:

- Block diagram with interfaces
- State machine diagrams (where applicable)
- Timing diagrams

- Interface specifications
- Synthesis constraints

2. SystemVerilog Fundamentals for Digital Design

2.1 Key SystemVerilog Constructs for Design

Data Types

```
// Logic types - use for all digital signals
logic        single_bit;
logic [7:0]  byte_data;
logic [31:0] word_data;

// Packed arrays - synthesizes to contiguous bits
logic [3:0][7:0] packed_array;  // 4 bytes packed

// Unpacked arrays - used for memories
logic [7:0] memory [0:1023];    // 1K x 8-bit memory

// Enumerations for states - highly recommended
typedef enum logic [2:0] {
    IDLE, START, PROCESS, WAIT_ACK, DONE
} state_t;
```

Always Blocks - Critical for Synthesis

```
// Combinational logic - ALWAYS use always_comb
always_comb begin
    // All outputs must be assigned in all paths
    // Use blocking assignments (=)
end

// Sequential logic - ALWAYS use always_ff
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        // Reset all registers
    end else begin
        // Use non-blocking assignments (<=)
    end
end
```

2.2 Synthesis Guidelines

Golden Rules:

- Use `always_ff` for sequential logic only
- Use `always_comb` for combinational logic only
- Use non-blocking assignments (<=) in sequential blocks
- Use blocking assignments (=) in combinational blocks
- Avoid latches unless specifically required
- Initialize all variables
- Avoid combinational loops

3. Lab 1: Basic Combinational Circuits

3.1 Objective

Master combinational logic design using SystemVerilog synthesis constructs.

3.2 Lab 1A: 8-bit Arithmetic Logic Unit (ALU)

Design Requirements

- 8-bit ALU supporting: ADD, SUB, AND, OR, XOR, NOT, SLL, SRL
- 3-bit operation select
- Status outputs: Zero, Carry, Overflow
- Optimized for FPGA implementation

Design Steps

1. **Create truth table** for all 8 operations
2. **Draw block diagram** showing datapath
3. **Optimize carry/overflow logic**
4. **Consider FPGA resources**

Code Framework

```
module alu_8bit (
    input  logic [7:0] a, b,
    input  logic [2:0] op_sel,
    output logic [7:0] result,
    output logic       zero, carry, overflow
);

    // TODO: Implement operation selection
    always_comb begin
        // Initialize all outputs
        carry = 1'b0;
        overflow = 1'b0;

        case (op_sel)
            // TODO: Implement each operation
            // Consider overflow detection logic
            default: result = 8'b0;
        endcase

        // TODO: Implement flag generation
    end

endmodule
```

3.3 Lab 1B: Priority Encoder with Enable

Design Requirements

- 8-to-3 priority encoder with input enable
- Active-high inputs, MSB has highest priority
- Outputs: 3-bit encoded value, valid signal
- Must handle all-zero input case

Design Steps

1. **Create truth table** for all input combinations
2. **Use K-maps** to optimize the logic equations
3. **Consider using casez** for don't-care optimization

Code Framework

```
module priority_encoder_8to3 (
    input  logic       enable,
    input  logic [7:0] data_in,
    output logic [2:0] encoded_out,
    output logic       valid
);

    // TODO: Implement priority encoding
    // Hint: Consider using casez with don't-care patterns

endmodule
```

4. Lab 2: Advanced Combinational Logic

4.1 Lab 2A: 32-bit Barrel Shifter

Design Requirements

- 32-bit data input/output
- 5-bit shift amount (0-31 positions)
- Direction control (left/right)
- Mode control (shift/rotate)
- Single cycle operation

Design Methodology

1. **Draw the datapath** showing all multiplexer stages
2. **Optimize multiplexer logic** for minimum delay
3. **Consider FPGA routing resources**

Code Framework

```
module barrel_shifter (
    input  logic [31:0] data_in,
    input  logic [4:0]  shift_amt,
    input  logic        left_right,  // 0=left, 1=right
    input  logic        shift_rotate, // 0=shift, 1=rotate
    output logic [31:0] data_out
);

    // TODO: Implement multi-stage shifting
    // Stage signals for intermediate results
    logic [31:0] stage0, stage1, stage2, stage3, stage4;

    // TODO: Implement each stage
    // Consider: How to handle fill bits for shifts vs rotates?

endmodule
```

Design Questions

- How many LUTs will this consume on your FPGA?
- Can you pipeline this for higher frequency?
- What's the critical path through your design?

4.2 Lab 2B: Binary Coded Decimal (BCD) Converter

Design Requirements

- Convert 8-bit binary to 3-digit BCD
- Purely combinational implementation
- Input range: 0-255, Output: 000-255 in BCD

Algorithm Understanding

1. **Study Double-Dabble algorithm**
2. **Trace through examples** on paper

Code Framework

```
module binary_to_bcd (
   input  logic [7:0]  binary_in,
   output logic [11:0] bcd_out    // 3 BCD digits: [11:8][7:4][3:0]
);

   // TODO: Implement Double-Dabble algorithm
   // Consider: Combinational loop approach vs generate loops

endmodule
```

5. Lab 3: Sequential Circuit Fundamentals

5.1 Lab 3A: Programmable Counter

Design Requirements

- 8-bit up/down counter with programmable limits
- Control inputs: load, enable, up/down, reset
- Status outputs: terminal count, zero detect
- Synchronous operation with proper reset

Design Methodology

1. **Draw state diagram** showing all counter states
2. **Define control logic** for each input combination
3. **Plan reset strategy** (synchronous vs asynchronous)
4. **Consider metastability** for control inputs

Code Framework

```
module programmable_counter (
    input  logic        clk,
    input  logic        rst_n,
    input  logic        load,
    input  logic        enable,
    input  logic        up_down,
    input  logic [7:0]  load_value,
    input  logic [7:0]  max_count,
    output logic [7:0]  count,
    output logic        tc,          // Terminal count
    output logic        zero
);

    // TODO: Implement counter logic
    // Consider: What happens when max_count changes during operation?

endmodule
```

6. Lab 4: Finite State Machines

6.1 State Machine Design Methodology

Essential Design Steps

1. **Understand the specification completely**
2. **Identify all states and transitions**
3. **Draw state diagram with all conditions**
4. **Optimize state encoding** (binary, one-hot, gray)
5. **Separate state register, next-state logic, and output logic**
6. **Plan reset state and error handling**
7. **Consider timing and setup/hold requirements**

State Machine Template

```
// Define state enumeration
typedef enum logic [2:0] {
    IDLE = 3'b000,
    START = 3'b001,
    // TODO: Add more states
} state_t;

module fsm_template (
    input  logic clk,
    input  logic rst_n,
    // TODO: Add control inputs
    // TODO: Add status outputs
);

    state_t current_state, next_state;

    // State register - ALWAYS separate this
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            current_state <= IDLE;
        end else begin
            current_state <= next_state;
        end
    end

    // Next state logic - ALWAYS use always_comb
    always_comb begin
        next_state = current_state; // Default assignment prevents latches
        case (current_state)
            // TODO: Implement state transitions
        endcase
    end

    // Output logic - Separate from state logic
    // TODO: Implement Moore or Mealy outputs

endmodule
```

6.2 Lab 4A: Traffic Light Controller

Specification

- 4-way intersection with North-South and East-West directions
- Normal cycle: Green(30s) → Yellow(5s) → Red
- Emergency override: All red with flashing
- Pedestrian crossing request handling
- 1 Hz clock input (students must create timer)

Design Process

1. **Draw complete state diagram** including:
    - Normal operation states
    - Emergency states
    - Pedestrian request states
    - All transition conditions
2. **Design timer module** for time delays
3. **Plan emergency handling -** immediate vs safe transition
4. **Consider pedestrian priority** logic

State Diagram Guidelines

States to consider:

- NS_GREEN_EW_RED
- NS_YELLOW_EW_RED
- NS_RED_EW_GREEN
- NS_RED_EW_YELLOW
- EMERGENCY_ALL_RED
- PEDESTRIAN_CROSSING
- STARTUP_FLASH

Code Framework

```
module traffic_controller (
    input  logic       clk,          // 1 Hz
    input  logic       rst_n,
    input  logic       emergency,
    input  logic       pedestrian_req,
    output logic [1:0] ns_lights,     // [Red, Yellow, Green]
    output logic [1:0] ew_lights,
    output logic       ped_walk,
    output logic       emergency_active
);

    // TODO: Define states and implement FSM
    // Consider: How to handle competing requests?

endmodule
```

6.3 Lab 4B: Vending Machine Controller

Specification

- Accepts 5, 10, 25 cent coins
- Dispenses 30-cent item
- Provides correct change
- Handles coin return request
- LED display for current amount

Design Methodology

1. **List all possible states** based on money inserted (0¢, 5¢, 10¢, 15¢, 20¢, 25¢, 30¢+)
2. **Draw state transitions** for each coin input
3. **Plan change-making logic** (what coins to return?)
4. **Handle error conditions** (coin jam, exact change only)

Code Framework

```
module vending_machine (
    input  logic       clk,
    input  logic       rst_n,
    input  logic       coin_5,      // 5-cent coin inserted
    input  logic       coin_10,     // 10-cent coin inserted
    input  logic       coin_25,     // 25-cent coin inserted
    input  logic       coin_return,
    output logic       dispense_item,
    output logic       return_5,    // Return 5-cent
    output logic       return_10,   // Return 10-cent
    output logic       return_25,   // Return 25-cent
    output logic [5:0] amount_display
);

    // TODO: Implement vending machine FSM
    // Consider: Coin input synchronization and debouncing

endmodule
```

7. Lab 5: Counters and Timers

7.1 Lab 5A: Multi-Mode Timer

Specification

- 32-bit programmable timer with multiple modes:
  - One-shot: Count down once and stop
  - Periodic: Reload and restart automatically
  - PWM: Generate PWM with programmable duty cycle
- 1 MHz input clock, programmable prescaler
- Interrupt generation capability

Design Approach

1. **Design prescaler** for clock division
2. **Plan mode control logic**
3. **Design reload mechanism**
4. **PWM duty cycle calculation**

Code Framework

```
module multi_mode_timer (
    input  logic        clk,         // 1 MHz
    input  logic        rst_n,
    input  logic [1:0]  mode,        // 00=off, 01=one-shot, 10=periodic, 11=PWM
    input  logic [15:0] prescaler,   // Clock divider
    input  logic [31:0] reload_val,
    input  logic [31:0] compare_val, // For PWM duty cycle
    input  logic        start,
    output logic        timeout,
    output logic        pwm_out,
    output logic [31:0] current_count
);

    // TODO: Implement timer with all modes
    // Consider: How to handle mode changes during operation?

endmodule
```

8. Lab 6: Memory Interfaces

8.1 Lab 6A: Synchronous SRAM Controller

Specification

- Interface to 32Kx16 synchronous SRAM
- Single-cycle read/write operation
- Address and data buses with proper timing
- Chip enable and output enable control

Interface Timing Analysis

1. **Study SRAM datasheet** timing requirements
2. **Draw timing diagrams** for read and write cycles
3. **Calculate setup/hold times** relative to clock
4. **Plan address/data multiplexing**

Code Framework

```
module sram_controller (
    input  logic        clk,
    input  logic        rst_n,
    input  logic        read_req,
    input  logic        write_req,
    input  logic [14:0] address,
    input  logic [15:0] write_data,
    output logic [15:0] read_data,
    output logic        ready,

    // SRAM interface
    output logic [14:0] sram_addr,
    inout  wire  [15:0] sram_data,
    output logic        sram_ce_n,
    output logic        sram_oe_n,
    output logic        sram_we_n
);

    // TODO: Implement SRAM control logic
    // Consider: Bidirectional data bus control

endmodule
```

8.2 Lab 6B: DDR Memory Controller (Simplified)

Specification

- Basic DDR memory interface controller
- Initialization sequence handling
- Refresh control logic
- Read/write command scheduling

Design Complexity Note

This is an advanced topic - focus on understanding the concepts and implementing a simplified version.

9. Lab 7: FIFO Design

9.1 Synchronous FIFO

Specification

- Parameterizable width and depth
- Full/empty flag generation
- Almost-full/almost-empty thresholds
- Efficient FPGA block RAM utilization

Design Methodology

1. **Choose pointer width** (binary vs Gray code)
2. **Design flag generation logic**
3. **Plan memory instantiation** (inferred vs explicit)
4. **Optimize for timing** and resource usage

Key Design Decisions

- Binary counters with comparison logic vs Gray code counters
- Registered vs combinational output flags
- Power-of-2 vs arbitrary depth handling

Code Framework

```
module sync_fifo #(
   parameter int DATA_WIDTH = 8,
   parameter int FIFO_DEPTH = 16,
   parameter int ALMOST_FULL_THRESH = 14,
   parameter int ALMOST_EMPTY_THRESH = 2
)(
   input  logic                  clk,
   input  logic                  rst_n,
   input  logic                  wr_en,
   input  logic [DATA_WIDTH-1:0]   wr_data,
   input  logic                  rd_en,
   output logic [DATA_WIDTH-1:0]   rd_data,
   output logic                   full,
   output logic                   empty,
   output logic                   almost_full,
   output logic                   almost_empty,
   output logic [$clog2(FIFO_DEPTH):0] count
);
```

```
    // TODO: Implement FIFO logic
    // Consider: How to generate flags without glitches?

endmodule
```

9.2 Asynchronous FIFO (Clock Domain Crossing)

Specification

- Handles different clock domains for read/write
- Gray code pointers for safe domain crossing
- Metastability protection
- Proper flag synchronization

Critical Design Points

1. **Gray code pointer generation** and comparison
2. **Multi-flop synchronizers** for domain crossing
3. **Flag generation timing** to avoid false flags
4. **Reset handling** across clock domains

10. Lab 8: UART Controller

10.1 UART Transmitter

Specification

- Configurable baud rate (9600, 19200, 38400, 115200)
- 8-bit data, 1 start bit, 1 stop bit, optional parity
- Transmit FIFO with configurable depth
- Status flags: busy, FIFO full/empty

Design Steps

1. **Calculate baud rate generation** - create timing diagram
2. **Draw UART frame format**
3. **Design transmit state machine**
4. **Integrate with FIFO**

State Diagram for TX

States: IDLE → LOAD → START_BIT → DATA_BITS → PARITY → STOP_BIT → IDLE

Code Framework

```
module uart_transmitter #(
    parameter int CLK_FREQ = 50_000_000,
    parameter int BAUD_RATE = 115200,
    parameter int FIFO_DEPTH = 8
)(
    input  logic       clk,
    input  logic       rst_n,
    input  logic [7:0] tx_data,
    input  logic       tx_valid,
    output logic       tx_ready,
    output logic       tx_serial,
    output logic       tx_busy
);

    // TODO: Implement UART transmitter
    // Consider: Baud rate accuracy and jitter

endmodule
```

10.2 UART Receiver

Design Challenges

- Start bit detection and validation
- Data sampling at optimal points
- Frame error detection
- Receive FIFO integration

11. Lab 9: SPI Controller

11.1 SPI Master Controller

Specification

- Configurable clock polarity and phase (CPOL/CPHA)
- Variable clock frequency
- Automatic slave select control
- Bidirectional data transfer

Design Methodology

1. **Understand SPI timing** for all CPOL/CPHA combinations
2. **Draw timing diagrams** for each mode
3. **Design shift register** for data transfer
4. **Plan slave select timing**

Code Framework

```
module spi_master #(
    parameter int NUM_SLAVES = 4,
    parameter int DATA_WIDTH = 8
)(
    input  logic                        clk,
    input  logic                        rst_n,
    input  logic [DATA_WIDTH-1:0]       tx_data,
    input  logic [$clog2(NUM_SLAVES)-1:0] slave_sel,
    input  logic                        start_transfer,
    input  logic                        cpol,
    input  logic                        cpha,
    input  logic [15:0]                 clk_div,

    output logic [DATA_WIDTH-1:0]       rx_data,
    output logic                        transfer_done,
    output logic                        busy,

    // SPI interface
    output logic                        spi_clk,
    output logic                        spi_mosi,
    input  logic                        spi_miso,
    output logic [NUM_SLAVES-1:0]       spi_cs_n
);
```

```
    // TODO: Implement SPI master
    // Consider: How to handle different CPOL/CPHA modes?

endmodule
```

12. Lab 10: AXI4-Lite Interface Design

12.1 AXI4-Lite Protocol Overview

Key Characteristics

- 32-bit address and data buses
- Separate read/write address channels
- Write response channel
- No burst support (single transfers only)
- Simple handshake protocol (VALID/READY)

Channel Structure

```
Write Address Channel: AWADDR, AWVALID, AWREADY
Write Data Channel:    WDATA, WSTRB, WVALID, WREADY
Write Response:        BRESP, BVALID, BREADY
Read Address Channel:  ARADDR, ARVALID, ARREADY
Read Data Channel:     RDATA, RRESP, RVALID, RREADY
```

12.2 AXI4-Lite Slave Design

Specification

- Register bank with 16 x 32-bit registers
- Read/write access to all registers
- Address decode logic
- Proper AXI4-Lite response handling
- Error responses for invalid addresses

Design Process

1. **Study AXI4-Lite specification** - understand handshake protocol
2. **Draw timing diagrams** for read and write transactions
3. **Design address decoder**
4. **Plan register bank implementation**
5. **Design response logic**

Interface Definition

```
interface axi4_lite_if;
    // Write address channel
    logic [31:0] awaddr;
    logic        awvalid;
    logic        awready;
    // Write data channel
    logic [31:0] wdata;
    logic [3:0]  wstrb;
    logic        wvalid;
    logic        wready;

    // Write response channel
    logic [1:0]  bresp;
    logic        bvalid;
    logic        bready;

    // Read address channel
    logic [31:0] araddr;
    logic        arvalid;
    logic        arready;

    // Read data channel
    logic [31:0] rdata;
    logic [1:0]  rresp;
    logic        rvalid;
    logic        rready;

    // Modports for master and slave
    modport master (
        output awaddr, awvalid, wdata, wstrb, wvalid, bready,
               araddr, arvalid, rready,
        input  awready, wready, bresp, bvalid, arready, rdata, rresp, rvalid
    );

    modport slave (
        input  awaddr, awvalid, wdata, wstrb, wvalid, bready,
               araddr, arvalid, rready,
        output awready, wready, bresp, bvalid, arready, rdata, rresp, rvalid
    );
endinterface
```

Code Framework

```
module axi4_lite_slave (
    input  logic        clk,
    input  logic        rst_n,
    axi4_lite_if.slave  axi_if
);

    // Register bank - 16 x 32-bit registers
    logic [31:0] register_bank [0:15];

    // Address decode
    logic [3:0] write_addr_index, read_addr_index;
    logic       addr_valid_write, addr_valid_read;

    // State machines for read and write channels
    typedef enum logic [1:0] {
        W_IDLE, W_ADDR, W_DATA, W_RESP
    } write_state_t;

    typedef enum logic [1:0] {
        R_IDLE, R_ADDR, R_DATA
    } read_state_t;

    write_state_t write_state;
    read_state_t  read_state;

    // TODO: Implement write channel state machine
    // Consider: Outstanding transaction handling

    // TODO: Implement read channel state machine
    // Consider: Read data pipeline timing

    // TODO: Implement address decode logic
    // Consider: What constitutes a valid address?

    // TODO: Implement register bank
    // Consider: Which registers are read-only vs read-write?

endmodule
```

12.3 Design Challenges and Considerations

Protocol Compliance

- Handshake timing: VALID must not depend on READY
- Response requirements: All transactions must receive responses

- Address alignment: Handle unaligned accesses appropriately
- Write strobes: Implement byte-level write enables

Performance Optimization

- Pipeline read data path for back-to-back reads
- Minimize response latency
- Handle simultaneous read/write efficiently

Error Handling

- Invalid address detection
- Timeout mechanisms
- Protocol violation responses

12.4 Integration with Previous Labs

Design a complete system integrating:

- UART controller with AXI4-Lite interface
- Timer/counter modules accessible via AXI4-Lite
- FIFO status and control registers
- System control and status registers

13. FPGA Synthesis Guidelines

13.1 Synthesis-Friendly Coding

Clock Domain Design

- **Use single clock domain** when possible
- **Avoid generated clocks** in design logic
- **Register all outputs** from clock domains
- **Use proper clock enable** instead of clock gating

Reset Strategy

```
// Preferred: Asynchronous reset, synchronous release
always_ff @(posedge clk or negedge rst_n) begin
   if (!rst_n) begin
      // Reset state
   end else begin
      // Normal operation
   end
```

```
end

// Reset synchronizer for reliable release
module reset_sync (
    input  logic clk,
    input  logic async_rst_n,
    output logic sync_rst_n
);
    logic [1:0] reset_sync_reg;

    always_ff @(posedge clk or negedge async_rst_n) begin
        if (!async_rst_n) begin
            reset_sync_reg <= 2'b00;
        end else begin
            reset_sync_reg <= {reset_sync_reg[0], 1'b1};
        end
    end

    assign sync_rst_n = reset_sync_reg[1];
endmodule
```

Memory Inference

```
// Infers Block RAM
logic [7:0] memory [0:1023];

always_ff @(posedge clk) begin
    if (write_enable) begin
        memory[write_addr] <= write_data;
    end
    read_data <= memory[read_addr];  // Registered read
end

// Infers Distributed RAM (LUT-based)
logic [3:0] small_mem [0:15];
assign read_data = small_mem[read_addr];  // Combinational read

always_ff @(posedge clk) begin
    if (write_enable) begin
        small_mem[write_addr] <= write_data;
    end
end
```

DSP Block Utilization

```
// Infers DSP48 on Xilinx FPGAs
module dsp_multiply_accumulate (
    input  logic        clk,
    input  logic        rst_n,
    input  logic [17:0] a,
    input  logic [17:0] b,
    input  logic [47:0] c,
    output logic [47:0] result
);

    always_ff @(posedge clk) begin
        if (!rst_n) begin
            result <= 48'b0;
        end else begin
            result <= (a * b) + c;  // Multiply-accumulate pattern
        end
    end
endmodule
```

13.2 Resource Optimization Techniques

Logic Optimization

- **Use case statements** instead of nested if-else for large multiplexers
- **Balance logic depth** vs resource usage
- **Consider LUT combining** - 6-input LUTs can implement complex functions
- **Use one-hot encoding** for state machines when appropriate

Timing Optimization

```
// Pipeline complex combinational paths
module pipelined_adder (
    input  logic        clk,
    input  logic [31:0] a, b,
    output logic [31:0] sum
);

    logic [31:0] a_reg, b_reg;

    // Pipeline stage 1: Register inputs
    always_ff @(posedge clk) begin
```

```
        a_reg <= a;
        b_reg <= b;
    end

    // Pipeline stage 2: Perform addition
    always_ff @(posedge clk) begin
        sum <= a_reg + b_reg;
    end
endmodule
```

Clock Domain Crossing

```
// Two-flop synchronizer for single-bit signals
module bit_synchronizer (
    input  logic clk_dest,
    input  logic rst_n,
    input  logic data_in,
    output logic data_out
);

    logic [1:0] sync_reg;

    always_ff @(posedge clk_dest or negedge rst_n) begin
        if (!rst_n) begin
            sync_reg <= 2'b00;
        end else begin
            sync_reg <= {sync_reg[0], data_in};
        end
    end

    assign data_out = sync_reg[1];
endmodule
```

13.3 Synthesis Constraints

Timing Constraints Example

```
# Create clocks
create_clock -period 10.0 -name sys_clk [get_ports clk_100mhz]
create_clock -period 40.0 -name uart_clk [get_ports clk_25mhz]

# Set input/output delays
```

```
set_input_delay -clock sys_clk -max 2.0 [get_ports data_in]
set_output_delay -clock sys_clk -max 2.0 [get_ports data_out]

# False paths
set_false_path -from [get_ports reset_n]
set_false_path -from [get_clocks uart_clk] -to [get_clocks sys_clk]

# Multi-cycle paths
set_multicycle_path -setup 2 -from [get_pins slow_logic/*] -to [get_pins reg_bank/*]
```

13.4 Synthesis Reports Analysis

Resource Utilization

Students should analyze:

- LUT utilization and efficiency
- Block RAM usage vs distributed RAM
- DSP block utilization
- I/O buffer usage

Timing Analysis

Key metrics to monitor:

- Worst Negative Slack (WNS)
- Total Negative Slack (TNS)
- Clock skew and uncertainty
- Critical path analysis

14. Design Documentation Standards

14.1 Block Diagrams

Requirements for Every Module

1. **Top-level block** showing all I/O ports
2. **Internal architecture** for complex modules
3. **Interface timing** relationships
4. **Clock domain boundaries** clearly marked
5. **Reset distribution** shown

Example Documentation Structure

Module: uart_controller
Purpose: Full-duplex UART communication with configurable parameters

Block Diagram:
// Complete Block Diagrams of the module

Interface Signals:

- clk: System clock (50 MHz)
- rst_n: Active-low reset
- tx_data[7:0]: Transmit data bus
- rx_data[7:0]: Receive data bus
- tx_valid: Transmit data valid
- rx_ready: Receive data ready
- uart_tx: Serial transmit output
- uart_rx: Serial receive input

14.2 State Machine Documentation

State Diagram Requirements

1. **All states clearly labeled**
2. **All transitions with conditions**
3. **Reset state identified**
4. **Output signals indicated** (Moore vs Mealy)
5. **Timing relationships** specified

State Table Format

```
Current State | Input Conditions | Next State | Outputs
===========================================================
IDLE          | start='1'        | LOAD       | busy='1'
IDLE          | start='0'        | IDLE       | busy='0'
```

```
LOAD        | always        | TRANSMIT  | load_data='1'
TRANSMIT    | bit_count=8   | DONE      | shift_enable='1'
TRANSMIT    | bit_count<8   | TRANSMIT  | shift_enable='1'
DONE        | always        | IDLE      | done_pulse='1'
```

14.3 Timing Diagrams

Required Timing Information

1. **Clock relationships**
2. **Setup and hold times**
3. **Propagation delays**
4. **Interface handshake timing**
5. **Pipeline stage timing**

14.4 Interface Specifications

AXI4-Lite Interface Documentation Template

```
Interface: AXI4-Lite Slave
Address Map:
0x0000: Control Register (R/W)
  [31:16] Reserved
  [15:8]  Configuration bits
  [7:0]   Control bits

0x0004: Status Register (RO)
  [31:16] Error flags
  [15:8]  State information
  [7:0]   Status flags

0x0008: Data Register (R/W)
  [31:0]  Data payload

Transaction Timing:
- Address phase: 1 clock cycle minimum
- Data phase: 1 clock cycle minimum
- Response: 1 clock cycle
- Back-to-back reads: 2 clock latency
- Write-to-read turnaround: 1 clock cycle

Error Responses:
- SLVERR: Invalid address access
- OKAY: Normal completion
```

14.5 Design Review Checklist

Pre-Implementation Review

- [ ] Specification completely understood
- [ ] Block diagrams drawn and reviewed
- [ ] State diagrams complete with all transitions
- [ ] Interface timing analyzed
- [ ] Resource estimation completed
- [ ] Clock domain strategy defined

Post-Implementation Review

- [ ] Synthesis results meet timing requirements
- [ ] Resource utilization reasonable
- [ ] All states reachable and tested
- [ ] Reset behavior verified
- [ ] Clock domain crossings properly handled
- [ ] Documentation matches implementation

Code Quality Checklist

- [ ] Consistent naming conventions
- [ ] Proper module hierarchy
- [ ] All outputs driven in all conditions
- [ ] No combinational loops
- [ ] No unintended latches
- [ ] Reset strategy consistent
- [ ] Comments explain design intent

14.6 Final Project Integration

System-Level Design

For the AXI4-Lite final project, students should integrate multiple previous labs:

1. **Memory Map Design**: Plan register addresses for all modules
2. **Interrupt Handling**: Design interrupt controller for UART, timers
3. **Clock Management**: Multiple clock domains with proper crossing
4. **System Reset**: Hierarchical reset distribution
5. **Performance Analysis**: Meet timing at target frequency

Documentation Deliverables

1. **System Architecture Document**
    - Overall block diagram
    - Memory map specification
    - Clock domain strategy
    - Reset architecture
2. **Module Design Documents** (for each major module)
    - Detailed block diagrams
    - State machine diagrams
    - Interface specifications
    - Timing requirements
3. **Integration Test Plan**
    - System-level test scenarios
    - Performance requirements
    - Error handling verification
4. **Synthesis Report Analysis**
    - Resource utilization summary
    - Timing analysis results
    - Power estimation
    - Recommendations for optimization

Lab Exercise Guidelines

Pre-Lab Preparation

1. **Read the entire lab specification**
2. **Research relevant topics** using textbooks and online resources
3. **Draw all diagrams on paper** before coding
4. **Plan your module hierarchy**
5. **Estimate resource requirements**

During Lab Implementation

1. **Start with simple functionality** and build incrementally
2. **Test each module independently** before integration
3. **Use meaningful signal names** and consistent coding style
4. **Add comments explaining design decisions**
5. **Keep a lab notebook** with problems encountered and solutions

Post-Lab Analysis

1. **Analyze synthesis reports** thoroughly
2. **Compare actual vs estimated resources**
3. **Document any design changes** made during implementation
4. **Identify optimization opportunities**
5. **Prepare for integration** with other modules

Grading Criteria

- **Functionality**: Does the design meet all specifications?
- **Code Quality**: Is the code well-structured and readable?
- **Documentation**: Are diagrams and documentation complete?
- **Synthesis Results**: Does the design synthesize efficiently?
- **Understanding**: Can you explain your design decisions?

Additional Resources

Recommended Reading

- SystemVerilog for Design (Sutherland, Davidmann, Flake)
- Digital Design and Computer Architecture (Harris, Harris)
- FPGA vendor documentation (Xilinx UG901, Intel documentation)
- Industry standards (AXI4 specification, PCIe, USB)

Tools and Software

- Synthesis tools: Vivado, Quartus, Synplify
- Simulation tools: ModelSim, VCS, Xcelium
- Version control: Git for design management
- Documentation: Draw.io for diagrams, Markdown for text

Online Resources

- FPGA vendor forums and application notes
- IEEE standards documents
- Open-source IP repositories (OpenCores, GitHub)
- Industry blogs and technical articles

End of Lab Manual

This manual provides a comprehensive foundation for digital design using SystemVerilog. Each lab builds upon previous concepts while introducing new design challenges. Students should focus on understanding the underlying principles rather than just completing the code, as this knowledge will be essential for advanced digital design projects.