Digital Design Lab Manual - SystemVerilog

**Table of Contents**

[1. Introduction to Digital Design Methodology 5](#_9qsllaiw7vu4)

[1.1 Design Flow Overview 5](#_rqxrg2t4vdsi)

[1.2 Design Principles 5](#_fcui8tdzumvj)

[1.3 Design Steps for Each Lab 5](#_hcfxw4n86iaf)

[1.4 Documentation Requirements 5](#_9qg7ddqho5lq)

[2. SystemVerilog Fundamentals for Digital Design 7](#_s43d7shgq7q7)

[2.1 Key SystemVerilog Constructs for Design 7](#_u4bivuxvyox7)

[Data Types 7](#_ig91gcca8gk4)

[Always Blocks - Critical for Synthesis 7](#_62usaqtgdgm7)

[2.2 Synthesis Guidelines 7](#_o8ftr9f02h2o)

[3. Lab 1: Basic Combinational Circuits 9](#_y1zzo0gehdna)

[3.1 Objective 9](#_4gps47mwcfeu)

[3.2 Lab 1A: 8-bit Arithmetic Logic Unit (ALU) 9](#_uvvxbjjl2upw)

[Design Requirements 9](#_jr7xm1upelnw)

[Design Steps 9](#_d45uzgze2mv5)

[Code Framework 9](#_8hbghz5vmlmq)

[3.3 Lab 1B: Priority Encoder with Enable 10](#_iyculbu7f0b5)

[Design Requirements 10](#_lxq0ajlw7sq3)

[Design Steps 10](#_ry7uvb2znl1e)

[Code Framework 10](#_hefuos84yamk)

[4. Lab 2: Advanced Combinational Logic 11](#_vznvqb6pahv2)

[4.1 Lab 2A: 32-bit Barrel Shifter 11](#_buipx3z1ea5)

[Design Requirements 11](#_337x4g2bb94d)

[Design Methodology 11](#_8wugkh46oqx4)

[Code Framework 11](#_ppbzzz5rbrds)

[Design Questions 11](#_5kd9qkkmlth6)

[4.2 Lab 2B: Binary Coded Decimal (BCD) Converter 12](#_xif8a7aha50c)

[Design Requirements 12](#_s5jnlwa1qp0j)

[Algorithm Understanding 12](#_aw7m5xlwy1sn)

[Code Framework 12](#_fb4tbv4fyo2e)

[5. Lab 3: Sequential Circuit Fundamentals 13](#_vx23z0lc3yyf)

[5.1 Lab 3A: Programmable Counter 13](#_73sy22kl084)

[Design Requirements 13](#_mzikjm5ccy9l)

[Design Methodology 13](#_2qdwrbe3q34e)

[Code Framework 13](#_m40g3pc3eugc)

[6. Lab 4: Finite State Machines 14](#_y688cq91zi20)

[6.1 State Machine Design Methodology 14](#_2o1ifkl8odgv)

[Essential Design Steps 14](#_vva53vfen5kn)

[State Machine Template 14](#_15vk7473rpfg)

[6.2 Lab 4A: Traffic Light Controller 15](#_e40dwqg5vo3o)

[Specification 15](#_2sdgz531bkid)

[Design Process 15](#_gqznn01k5da6)

[State Diagram Guidelines 15](#_8kjrmgoc0a1d)

[Code Framework 16](#_a9j2zzt3udbe)

[6.3 Lab 4B: Vending Machine Controller 16](#_d5sr8yra5ok7)

[Specification 16](#_i0269bhmt78o)

[Design Methodology 16](#_2sae5lohkz5g)

[Code Framework 16](#_vbjfilss0s)

[7. Lab 5: Counters and Timers 18](#_tok9hc4y2cz2)

[7.1 Lab 5A: Multi-Mode Timer 18](#_89mmexqteqyr)

[Specification 18](#_owqkql8v9itz)

[Design Approach 18](#_az97yipivznx)

[Code Framework 18](#_ir67gejy2eld)

[8. Lab 6: Memory Interfaces 19](#_r8x83r6uoj9)

[8.1 Lab 6A: Synchronous SRAM Controller 19](#_ytepe0jfcswz)

[Specification 19](#_lit4nti7z4g4)

[Interface Timing Analysis 19](#_b858zfxr5kt8)

[Code Framework 19](#_pcbo7k4lhv6m)

[9. Lab 7: FIFO Design 20](#_5poxixrsqb2y)

[9.1 Synchronous FIFO 20](#_xjy78kshj8kh)

[Specification 20](#_mmoeqz2ktv3d)

[Design Methodology 20](#_937omjjwn231)

[Key Design Decisions 20](#_bjji7dt5el2q)

[Code Framework 20](#_1driqsgrrohw)

[9.2 Asynchronous FIFO (Clock Domain Crossing) 21](#_9044xxblb6pm)

[Specification 21](#_tcy19wbh27lt)

[Critical Design Points 21](#_r2nvnd2v89vr)

[10. Lab 8: UART Controller 22](#_8l7eu06fika8)

[10.1 UART Transmitter 22](#_7pnu34iwz4u9)

[Specification 22](#_wytwtiw93719)

[Design Steps 22](#_o5ah9p9osrdx)

[State Diagram for TX 22](#_dxt23ih6z36l)

[Code Framework 22](#_d2iaxqnqlbr)

[10.2 UART Receiver 23](#_8yyg8mpgbv21)

[Design Challenges 23](#_wbl6p75n7bg9)

[11. Lab 9: SPI Controller 24](#_i6408bmw1m6j)

[11.1 SPI Master Controller 24](#_4zchf5bolpum)

[Specification 24](#_v2n7oc8c9u9m)

[Design Methodology 24](#_s97e4tp93208)

[Code Framework 24](#_2u122a2ki3r9)

[12. Lab 10: AXI4-Lite Interface Design 26](#_g38ablue03g7)

[12.1 AXI4-Lite Protocol Overview 26](#_nbnl846z8qz2)

[Key Characteristics 26](#_gnn45yehr7cf)

[Channel Structure 26](#_lgr34nh2d3rf)

[12.2 AXI4-Lite Slave Design 26](#_jvh6qi33h6wa)

[Specification 26](#_xmd3kvin82wa)

[Design Process 26](#_2bseb3kmssew)

[Interface Definition 26](#_5yctfuqj5egl)

[Code Framework 27](#_h2xv4qh9adma)

[12.3 Design Challenges and Considerations 28](#_ssss1ojhb27r)

[Protocol Compliance 28](#_4fdgfc7iyp2m)

[Performance Optimization 29](#_2at0ixcn0w9c)

[Error Handling 29](#_fdborvbrbsqf)

[12.4 Integration with Previous Labs 29](#_45gf47mku741)

[13. FPGA Synthesis Guidelines 29](#_98j8f1l5k15o)

[13.1 Synthesis-Friendly Coding 29](#_46u7lc3d4z8b)

[Clock Domain Design 29](#_etle1kmoi61f)

[Reset Strategy 29](#_1uh0z9aljrwy)

[Memory Inference 30](#_cbcc31a88jvk)

[DSP Block Utilization 31](#_tmjcvkfs0oc0)

[13.2 Resource Optimization Techniques 31](#_alhof0mowzfh)

[Logic Optimization 31](#_5c1q7x7q086p)

[Timing Optimization 31](#_8tre0ecej52b)

[Clock Domain Crossing 32](#_b6sb2auqbuhi)

[13.3 Synthesis Constraints 32](#_cmig2qpfvib)

[Timing Constraints Example 32](#_f7d0xzvsjpzh)

[13.4 Synthesis Reports Analysis 33](#_kltuegz17y31)

[Resource Utilization 33](#_biufwvl87a48)

[Timing Analysis 33](#_244bpansqye4)

[14. Design Documentation Standards 34](#_9892zcw2vi9g)

[14.1 Block Diagrams 34](#_bwlfpgirlwgb)

[Requirements for Every Module 34](#_xkgnx3fb55t4)

[Example Documentation Structure 34](#_etxrbegle6hh)

[14.2 State Machine Documentation 34](#_yiq6dtw1vobb)

[State Diagram Requirements 34](#_tb6n089c8csq)

[State Table Format 34](#_uzk89232a0d9)

[14.3 Timing Diagrams 35](#_wagwlxjhre8x)

[Required Timing Information 35](#_kyjge69tmqjl)

[14.4 Interface Specifications 35](#_re3orrmvxrba)

[AXI4-Lite Interface Documentation Template 35](#_b371uig4kk6d)

[14.5 Design Review Checklist 36](#_u5mdhy44dfyk)

[Pre-Implementation Review 36](#_77ohqbz780vd)

[Post-Implementation Review 36](#_kpkk62rb577b)

[Code Quality Checklist 36](#_36mhta1zgnft)

[14.6 Final Project Integration 36](#_7jf9cf11mwoa)

[System-Level Design 36](#_koqcggkw0blq)

[Documentation Deliverables 36](#_b25ruyvy87bq)

[Lab Exercise Guidelines 37](#_e1fe1qunx54o)

[Pre-Lab Preparation 37](#_bz79i2vnu318)

[During Lab Implementation 37](#_d3b9sa7cj34n)

[Post-Lab Analysis 37](#_g9nppthfct5r)

[Grading Criteria 38](#_ffuxdwwx7juy)

[Additional Resources 38](#_7ijio5pemwqo)

[Recommended Reading 38](#_l89jcct8jfwt)

[Tools and Software 38](#_h9ptdois0v0i)

[Online Resources 38](#_y0akkjxy19g8)

## 

## **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 |
| --- |

## 

## **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 rearite
* 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.*