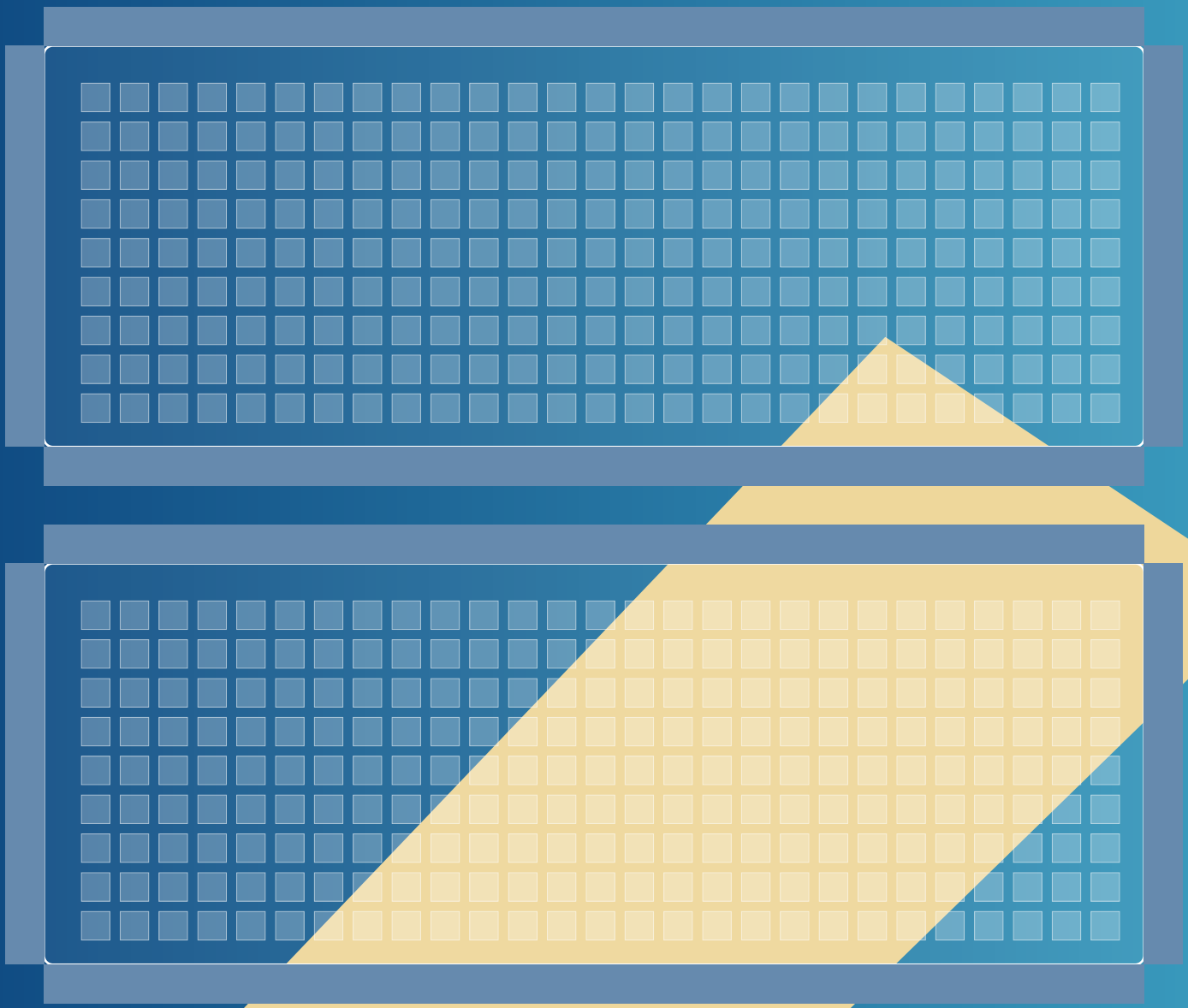
Verilog HDL and FPGA:
A Practical Approach to Digital and Logic Design

Komsan Kanjanasit

College of Computing, Prince of Songkla University



First Edition • 2025 | ISBN (e-book): 978-616-271-830-4

Verilog HDL and FPGA

A Practical Approach to Digital and Logic Design

Komsan Kanjanasit, Ph.D.

First Edition — 2025

ISBN (e-book): 978-616-271-830-4

Published by

College of Computing
Prince of Songkla University
Thailand

Published by College of Computing Prince of Songkla University

National Library of Thailand Cataloging in Publication Data

Komsan Kanjanasit.

Verilog HDL and FPGA: A Practical Approach to Digital Logic and Design,–

Phuket : College of Computing Prince of Songkla University, 2025.

1. System design. I. Title.

004.21

ISBN 978-616-271-830-4

Declaration of Generative AI Assistance

This textbook was prepared with the assistance of generative artificial intelligence (AI) tools, including OpenAI's ChatGPT. These tools were utilized to:

- Improve the clarity, coherence, and overall readability of the English text,
- Detect and correct grammatical, syntactic, and stylistic issues,
- Assist in generating structural templates and formatting suggestions.

All AI-assisted content has been critically reviewed, revised, and approved by the author to ensure technical accuracy, originality, and academic integrity. The author retains full responsibility for the final content presented in this work.

Preface

The growing demand for digital systems in today's technology-driven world has reshaped engineering education. From consumer electronics to embedded systems, digital hardware is central to intelligent devices. This textbook, *Verilog HDL and FPGA: A Practical Approach to Digital Logic and Design*, introduces undergraduate students to the fundamentals and practical aspects of digital system design using Verilog HDL and Field-Programmable Gate Arrays (FPGAs).

This book supports the **977-114 Digital Logic and Design** course, a foundational component of the undergraduate Digital Engineering curriculum (international program) at the College of Computing, Prince of Songkla University. It also contributes to the Electronics and Internet of Things module. The book strikes a balance between theoretical concepts and hands-on practice, guiding students through the design, simulation, testing, and debugging of digital circuits using industry-standard tools.

Verilog HDL is used as the primary language due to its widespread industry and academic adoption. Through structured examples—from logic gates to state machines and system-level projects—students learn synthesizable coding, design abstraction, and complete FPGA workflows using tools like Xilinx Vivado.

Each chapter offers examples, simulation guidance, synthesis tips, and exercises to strengthen understanding. Mini-projects provide opportunities to build practical FPGA-based systems.

Ideal for both classroom use and self-study, this book aims to equip students with the skills and insight needed for modern digital design.

Komsan Kanjanasit

Acknowledgment

This book reflects years of teaching, research, and collaboration in digital system design and intelligent systems. Its completion owes much to the support and contributions of many individuals and institutions.

I am especially grateful to my students, whose curiosity and dedication have inspired new perspectives and strengthened the link between theory, simulation, and real-world applications.

The author sincerely acknowledges the valuable contributions of the academic reviewers in evaluating and improving the textbook "Verilog HDL and FPGA: A Practical Approach to Digital Logic and Design."

Special appreciation is extended to the Internal Reviewer, the **Subcommittee for the Review of Textbooks for Academic Positions** under the **College of Computing, Prince of Songkla University**, for their constructive feedback and insightful suggestions that have significantly enhanced the academic and pedagogical quality of this book.

The author also wishes to express heartfelt thanks to the External Reviewers:

- **Assoc. Prof. Dr. Kritsanaphong Pansri**, Department of Electronics and Telecommunications Engineering, Faculty of Engineering, Rajamangala University of Technology Isan, Khon Kaen Campus
- **Asst. Prof. Dr. Nithizethe Mhuadthongon**, School of Science and Technology, Sukhothai Thammathirat Open University

for their invaluable comments, thoughtful critiques, and guidance in refining the content and improving its clarity and instructional effectiveness.

Their contributions have greatly helped to make this textbook a more comprehensive and accessible resource for students and educators alike.

Komsan Kanjanasit

Additional Resources

- Chapter Exercise Solutions – Suggested solutions for end-of-chapter questions.
- Hands-on Lab Practical – A collection of FPGA-based Verilog projects for practice.
- Lab Solutions – Step-by-step answers to lab assignments.

Contents

List of Figures

List of Tables

List of Abbreviations

ADC	Analog-to-Digital Converter
AI	Artificial Intelligence
ALM	Adaptive Logic Module
ALU	Arithmetic Logic Unit
AMC	artificial Magnetic Conductor
APB	Advanced Peripheral Bus
ASIC	Application-Specific Integrated Circuits
ASM	Algorithmic State Machine
AXI	Advanced eXtensible Interface
BCD	Binary Coded Decimal
BRAM	Block RAM
CDC	Clock Domain Crossing
CLB	Configurable Logic Blocks
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DDR	Double Data Rate
DFT	Design for Testability
DRC	Design Rule Check
DSP	Digital Signal Processing
DUT	Design Under Test
EDA	Electronic Design Automation
ECC	Error Correction Code
ECU	Electronic Control Unit
EDIF	Electronic Design Interchange Format
FEC	Forward Error Correction
FF	Flip-Flop
FIFO	First In, First Out
FIR	Finite Impulse Response
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
FSM	Finite State Machine

HDL	Hardware Description Languages
HDMI	High-Definition Multimedia Interface
HLS	High-Level Synthesis
I/O	Input/Output
ILA	Integrated Logic Analyzers
IP	Intellectual Property
IOB	Input/Output Blocks
IoT	Internet of Things
JTAG	Joint Test Action Group
LCD	Liquid Crystal Display
LFSR	Linear Feedback Shift Register
LBIST	Logic Built-In Self-Test
LCD	Liquid Crystal Display
LDPC	Low-Density Parity-Check
LED	Light Emitting Diode
LSB	Least Significant Bit
LUT	Look-Up Tables
NRE	Non-Recurring Engineering
MAC	Multiply-Accumulate
MCU	Microcontroller
MIMO	Multiple Input Multiple Output
MISR	Multiple Input Signature Register
ML	Machine Learning
MMCM	Mixed-Mode Clock Managers
MSB	Most Significant Bit
MUX	Multiplexer
OFDM	Orthogonal Frequency Division Multiplexing
PAR	Place-and-Route
PID	Proportional–Integral–Derivative
PLL	Phase-Locked Loops
PMOD	Peripheral Module
PWM	Pulse Width Modulation
RAM	Random Access Memory
RISC-V	Reduced Instruction Set Computer Version Five
RTL	Register Transfer Level
SDC	Synopsys Design Constraints
SDN	Software-Defined Networking
SDR	Software-Defined Radios
SoC	System-on-Chip

SPI	Serial Peripheral Interface
STA	Static Timing Analysis
TNS	Total Negative Slack
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VCD	Value Change Dump
VGA	Video Graphics Array
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
WNS	Worst Negative Slack

Chapter 1

Digital Design and FPGA

Chapter Objectives

- Understand the basics of digital logic and system design.
- Describe FPGA architecture and the standard design flow.
- Describe Verilog HDL for simulation and FPGA prototyping.

1.1 Overview of Digital Systems

Digital systems form the foundation of modern electronics, enabling functionality in everything from household appliances and smartphones to automotive control units and aerospace systems. Unlike analog systems that represent information continuously, digital systems use discrete binary values—typically `0` and `1`—to represent data and control operations. This binary representation underpins the reliability and versatility of digital computing and communication technologies.

1.1.1 Advantages of Digital Systems

Digital systems offer several benefits that make them preferable in a wide range of applications:

- **Noise Immunity:** Digital signals are more resistant to noise and degradation, ensuring better signal integrity over long distances or in noisy environments.
- **Scalability:** Digital designs can be easily scaled by adding more components or functionality without a complete redesign.
- **Reproducibility:** Once a digital system is designed and verified, it can be reproduced consistently across different platforms or manufacturing runs.

- **Integration:** Advances in VLSI (Very Large Scale Integration) allow millions of logic gates and memory elements to be packed into a single chip, enabling powerful and compact systems.

1.1.2 Applications

Digital systems are used in nearly every technological domain. Common applications include:

- **Microprocessors and Microcontrollers:** Central to computing and control systems in embedded applications.
- **Digital Signal Processing (DSP):** Real-time processing of audio, video, and sensor signals using digital computation.
- **Communication Systems:** Handling modulation, error correction, and data framing in wired and wireless communication protocols.
- **Embedded and IoT Devices:** Powering smart devices and sensors in applications such as home automation, wearables, and industrial control.
- **Consumer Electronics:** TVs, gaming consoles, smartphones, and appliances are all driven by digital systems.

1.2 Basic Concepts in Digital and Logic

Digital logic is the foundation of all digital electronic systems, from simple circuits like LED blinkers to complex processors and memory controllers. Understanding its core concepts is essential before moving into hardware description languages like Verilog.

1.2.1 Switching Theory

Switching theory is the mathematical foundation for analyzing and designing digital systems. It focuses on the study of switching devices, such as transistors, relays, or logic gates, that function in two discrete states:

$$0 \quad \text{(OFF, low, false)} \quad \text{and} \quad 1 \quad \text{(ON, high, true)}.$$

Since digital circuits rely exclusively on binary signals, switching theory establishes the rules and systematic methods for constructing, analyzing, and simplifying the logical operations that support modern digital and logic design.

Example: In switching theory, a 2-input AND gate produces an output of 1 only when both inputs are 1, whereas a 2-input OR gate produces an output of 1 if at least one input is 1.

1.2.2 Binary Number System

The binary number system uses only two digits: `0` and `1`. Each digit is referred to as a **bit**. Like the decimal system, the binary system is a positional system, but it is based on powers of 2.

In general, a binary number with digits $b_{n-1}b_{n-2}\ldots b_1b_0$ can be expressed as:

$$(b_{n-1}b_{n-2}\ldots b_1b_0)_2 = \sum_{i=0}^{n-1} b_i \times 2^i \tag{1.1}$$

where each $b_i \in \{0, 1\}$. For example, the binary number 1011 represents:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11_{10}$$

1.2.3 Boolean Algebra

Boolean algebra provides the mathematical framework for analyzing and designing digital logic circuits. It uses binary variables and logical operations.

Basic Boolean operations:

- **AND** ($A \cdot B$ or AB) — true if both inputs are 1.
- **OR** ($A + B$) — true if at least one input is 1.
- **NOT** ($\overline{A}$) — inverts the input.

These operations follow specific laws (commutative, distributive, DeMorgan's theorem, etc.) and form the basis of logic simplification and optimization.

1.2.4 Logic Gates

Logic gates are the physical realization of Boolean operations. They are the fundamental building blocks of digital hardware.

- **Basic Gates:** AND, OR, NOT
- **Derived Gates:** NAND, NOR, XOR, XNOR

Each gate has a defined truth table that maps input combinations to output values.

1.2.5 Combinational vs. Sequential Logic

Digital systems are broadly classified into:

- **Combinational Logic:** The output depends only on the current inputs. There is no memory element. Examples include:
  - Adders
  - Multiplexers
  - Encoders/Decoders
- **Sequential Logic:** The output depends on current inputs and the history of inputs (i.e., previous states). These systems require memory elements like flip-flops. Examples include:
  - Counters
  - Registers
  - Finite State Machines

Understanding these basic concepts equips you to design and analyze digital circuits and prepares you for modeling and simulating them using Verilog HDL in later chapters.

1.3 Introduction to Hardware Description Languages

HDLs (Hardware Description Languages) are specialized computer languages used to describe, model, and simulate digital electronic systems. As digital designs grow in complexity, manual circuit implementation using logic gates becomes inefficient and error-prone. HDLs provide a higher level of abstraction, enabling designers to develop, verify, and synthesize complex systems effectively.

1.3.1 Why HDLs?

Designing large-scale digital systems directly at the gate level is impractical due to the vast number of components and interconnections. HDLs simplify this process by allowing designers to:

- Describe behavior and structure of hardware at various abstraction levels.
- Simulate and test digital logic before fabrication.
- Automatically generate gate-level implementations using synthesis tools.

HDLs bridge the gap between design intent and physical implementation, streamlining both the design and verification processes.

1.3.2 Types of HDLs

The two most widely used HDLs in industry and academia are:

- **Verilog HDL:** Known for its concise syntax and C-like structure, Verilog is widely used in ASIC and FPGA development. It supports structural, dataflow, and behavioral modeling styles.

- **VHDL (VHSIC Hardware Description Language):** Originally developed under a U.S. Department of Defense program, VHDL is known for its strong typing and verbose syntax. It is popular in safety-critical and formally verified designs.

Both languages are IEEE standardized and supported by major EDA (Electronic Design Automation) tools.

1.3.3 Applications of HDLs

HDLs are central to modern digital design workflows. Key applications include:

- **RTL (Register Transfer Level) Modeling:** Describing the flow of data and control signals at the register level, which is the basis for synthesis.

- **Simulation and Verification:** Validating functional behavior of designs through testbenches, waveform analysis, and assertions.

- **Logic Synthesis:** Translating HDL code into gate-level netlists suitable for hardware implementation.

- **ASIC and FPGA Design:** Using HDLs to create designs that target Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs), enabling both prototyping and production.

HDLs form the backbone of digital system design, enabling abstraction, automation, and optimization across the development lifecycle.

Table 1.1 compares Verilog and VHDL across syntax, usage, typing discipline, and application domains—highlighting their strengths in different industrial and academic contexts.

Table 1.1: Comparison between Verilog and VHDL

Verilog HDL	**VHDL**
C-like syntax, concise and easier for beginners	Ada-like syntax, verbose but strong in structure
Popular in the U.S. and industry for ASIC/FPGA design	Widely used in Europe and defense/aerospace sectors
Case-sensitive	Not case-sensitive
Loosely typed language	Strongly typed language
Designed for simulation and synthesis	Originally for documentation, later evolved for synthesis
Supports multiple modeling styles: behavioral, dataflow, structural	Supports multiple abstraction levels including behavioral and structural
Easier to write and debug due to simpler syntax	Better for large and safety-critical designs due to strict rules
IEEE Standard: IEEE 1364	IEEE Standard: IEEE 1076
Faster learning curve for students	Better suited for complex, strongly-typed designs

1.4 What is an FPGA?

1.4.1 Definition

An FPGA is a type of integrated circuit that allows users to configure and program its logic functionality after manufacturing. Unlike fixed-function chips, FPGAs contain an array of CLBs, programmable interconnects, and I/O blocks, all of which can be reconfigured to implement various digital circuits and systems.

This flexibility makes FPGAs ideal for designing custom hardware accelerators, control systems, signal processing units, and many other digital applications without the time and cost of manufacturing custom silicon.

1.4.2 FPGA vs. ASIC

- **FPGA:** Reprogrammable hardware that enables rapid design iteration, debugging, and prototyping. FPGAs are suitable for applications requiring flexibility, such as low- to medium-volume production, proof-of-concept designs, and field upgrades.

- **ASIC:** Custom-fabricated silicon tailored for a specific function. ASICs offer high performance and low power consumption but require a long development time and high non-recurring engineering (NRE) costs, making them suitable for high-volume production.

Table 1.2: Comparison of FPGA, ASIC, and microcontroller technologies

Feature	**FPGA**	**ASIC**	**Microcontroller (MCU)**
Configurability	Field-programmable	Fixed hardware	Software programmable
Performance	High (parallelism)	Very high (custom)	Moderate
Cost	Low for low volume	High NRE, low per-unit at volume	Very low
Power Efficiency	Moderate	Excellent	Good
Development Time	Short to moderate	Long (design + fab)	Very short
Reusability	Reprogrammable	Not reusable	Programmable via firmware
Best Use Cases	Prototyping, acceleration, reconfigurable logic	Mass production, performance-critical apps	Control systems, low-power embedded tasks
Non-Volatility	Volatile (needs bitstream)	Non-volatile	Non-volatile (with flash/ROM)

1.4.3 Why Use FPGAs?

FPGAs are favored in many applications due to their reconfigurability and performance benefits. Key advantages include:

- **Hardware Acceleration:** FPGAs can perform computations faster than software by executing tasks directly in parallel hardware logic.
- **Customizable Parallel Processing:** Designers can implement parallel pipelines and data paths tailored to the application, maximizing throughput and efficiency.
- **Real-Time Control:** FPGAs respond deterministically to inputs, making them ideal for control systems and time-critical applications such as robotics or communications.

- **Rapid Prototyping:** FPGAs allow iterative testing and validation of hardware designs before committing to ASIC fabrication, reducing risk and time-to-market.

FPGAs serve as a bridge between flexibility and performance, enabling designers to explore, verify, and deploy complex digital systems efficiently. Table 1.2 compares key characteristics of FPGAs, ASICs, and microcontrollers, outlining their trade-offs in performance, flexibility, cost, and ideal application domains.

1.5 FPGA Architecture

An FPGA consists of a reconfigurable grid of digital components that can be customized to implement any logic function. The internal architecture of a modern FPGA typically includes the following key components:

- **Configurable Logic Blocks (CLBs):** The fundamental building blocks of logic implementation in an FPGA. Each CLB contains Look-Up Tables (LUTs), flip-flops, and multiplexers that can be configured to realize combinational and sequential circuits.

- **Interconnect Routing Fabric:** A programmable network of wires and switches that connects the CLBs, I/O blocks, memory elements, and DSP slices. This fabric determines the path of data signals and enables flexible, high-speed communication between logic elements.

- **Input/Output Blocks (IOBs):** Interface circuits located at the periphery of the FPGA. IOBs connect the internal logic to external pins and support multiple voltage standards, bidirectional communication, and configurable drive strength.

- **Block RAM (BRAM):** Dedicated embedded memory blocks within the FPGA that provide on-chip storage for buffering, caching, or data processing. These RAM blocks are dual-port and support various access widths and depths.

- **DSP Slices:** Specialized blocks for fast arithmetic operations such as multiplication, accumulation, and filtering. They are optimized for signal processing, video/image processing, and machine learning workloads.

- **Clocking and PLL Units:** FPGAs include global and regional clock networks along with Phase-Locked Loops (PLLs) and Mixed-Mode Clock Managers (MMCMs) to provide clock synthesis, distribution, and management for synchronous designs.

Understanding these architectural components is essential for optimizing FPGA resource utilization, improving performance, and ensuring scalable design implementation. Table 1.3 outlines the major architectural blocks in an FPGA.

Table 1.3: Summary of FPGA architecture components

Component	**Function**
CLBs	Contain LUTs and flip-flops for implementing logic and registers
Interconnect	Programmable routing network for connecting logic blocks and resources
IOBs	Handle communication between FPGA and external pins/signals
BRAM	On-chip memory for data buffering and storage
DSP Slices	Optimized units for multiplication, MACs, and signal processing
Clocking/PLLs	Provide clock management and frequency synthesis

1.6 Design Flow for FPGA-Based Systems

Designing digital systems with FPGAs involves a systematic sequence of steps that transform high-level functional requirements into a working hardware prototype. The standard FPGA design flow is outlined below:

1. **Design Specification:** Define the system's functional requirements, interfaces, performance goals, and resource constraints. This step guides architecture decisions and coding strategies.

2. **Verilog HDL Coding:** Develop the hardware logic using Verilog HDL. The design may include modules for combinational and sequential logic, arithmetic units, state machines, and data paths.

3. **Functional Simulation:** Verify the logical correctness of the design using a testbench and simulation tools (e.g., ModelSim, Vivado Simulator). Identify and fix functional bugs before synthesis.

4. **Synthesis:** Convert the Verilog RTL into a gate-level netlist using a synthesis tool (e.g., Vivado, Quartus, Synplify). This step maps the design into logic elements available on the FPGA.

5. **Place and Route (P&R):** Physically map the synthesized logic onto the FPGA resources. This includes placing logic blocks and routing interconnections to meet timing and resource constraints.

6. **Bitstream Generation:** Generate a configuration file (bitstream) that programs the FPGA. This file contains all the information needed to implement the design on the target device.

7. **FPGA Programming:** Use an FPGA programming tool (e.g., Vivado Hardware Manager or Quartus Programmer) to load the bitstream into the device via JTAG, SPI, or other interfaces.

8. **Testing and Verification:** Validate the design in real hardware using test instruments or embedded testbenches. Ensure it behaves as expected under all operating conditions.

This structured design flow ensures a reliable transition from HDL code to working hardware while enabling iterative debugging and optimization.

1.7 Simulation and Verification

Simulation is a critical step in the FPGA design flow that allows designers to test the behavior of their HDL code before synthesis and hardware implementation. It helps ensure the correctness of the logic design by running it in a virtual environment under various conditions.

Verification enhances the design process by systematically checking whether the circuit behaves according to its specification. It includes various techniques that increase the confidence that the design is free from functional errors.

Key Verification Techniques

- **Testbenches:** A testbench is a non-synthesizable piece of HDL code written to stimulate the Design Under Test (DUT). It provides input vectors, generates clocks, and monitors outputs to verify functionality.

- **Assertions:** Assertions are statements embedded in HDL code that automatically check for expected conditions during simulation. They help detect design bugs and enforce protocol compliance.

- **Coverage Analysis:** Coverage metrics help quantify how thoroughly the design has been tested. Common types include code coverage (e.g., line, condition, branch) and functional coverage (user-defined scenarios).

Simulation Tools

Common tools used for simulation and verification include:

- `ModelSim`, `Vivado Simulator`, `Icarus Verilog`, `VCS`, `QuestaSim`

Simulation and verification are essential to catching design issues early, reducing iteration time, and ensuring a successful synthesis and hardware implementation.

1.8 Tools for Verilog and FPGA Development

Effective Verilog HDL development and FPGA deployment rely on a range of tools for coding, simulation, synthesis, and device programming. This section introduces commonly used toolchains and simulators.

1.8.1 Popular FPGA Toolchains

- **Xilinx Vivado:** A comprehensive design suite for Xilinx FPGAs that supports Verilog/VHDL synthesis, simulation, implementation, and device programming. Includes IP integrator, clocking wizards, and timing analysis.
- **Intel Quartus Prime:** Formerly Altera Quartus, this toolchain supports development for Intel FPGAs. It includes tools for HDL compilation, pin assignments, timing simulation, and bitstream generation.
- **Lattice Diamond:** A design environment tailored for Lattice Semiconductor FPGAs. It offers synthesis, place-and-route, power analysis, and support for low-power and small-footprint devices.
- **Microsemi Libero SoC:** Used for Microchip (formerly Microsemi) FPGAs such as SmartFusion and IGLOO series. It integrates synthesis, simulation, and debugging tools for secure, low-power designs.

1.8.2 Simulation Tools

- **ModelSim:** An industry-standard simulator from Siemens EDA (Mentor Graphics). It supports mixed-language simulation and is widely used for both educational and professional projects.
- **Icarus Verilog + GTKWave:** An open-source toolchain. Icarus Verilog is a compiler/simulator for Verilog, while GTKWave is used to view simulation waveforms. Ideal for lightweight educational or scripting environments.

- **Vivado Simulator:** Integrated within Xilinx Vivado, it supports behavioral and timing simulation for Xilinx-based Verilog/VHDL designs with native waveform viewing and testbench support.

These tools support the full FPGA design workflow from RTL entry to hardware programming, enabling efficient and reliable development of digital systems.

1.9 Design Abstractions

Digital design is typically developed across multiple levels of abstraction. Each abstraction level provides a different view of the system, offering clarity, simplification, and tool compatibility for tasks such as modeling, verification, and synthesis.

- **Behavioral Level:** Describes the functionality of a system without specifying its structure. Designers write what the circuit should do (e.g., using 'if', 'case', 'for', or 'always' blocks) without detailing how it is implemented. This is commonly used for algorithmic modeling and testbenches.
- **Register-transfer level:** The most widely used abstraction for hardware synthesis. It models the system in terms of data flow between registers controlled by finite state machines and clocks. Designers define registers, multiplexers, and logic between them using Verilog constructs like 'always @(posedge clk)'.
- **Gate Level:** Specifies logic in terms of gates (AND, OR, NOT, etc.) and flip-flops. It is usually the result of synthesizing RTL code. Gate-level designs are used for timing analysis, equivalence checking, and low-level optimizations.
- **Layout Level:** Represents the physical design on silicon, including placement of standard cells, routing of wires, power distribution, and I/O pads. This level is used in ASIC workflows and not directly accessible in FPGA development.

Understanding and utilizing different design abstractions helps bridge the gap between conceptual functionality and implementable hardware.

1.10 Building FPGA Designs with Verilog

Verilog HDL serves as a cornerstone in modern FPGA development. It enables engineers to describe, simulate, and synthesize complex digital logic systems with efficiency and precision. Its combination of structural and behavioral modeling capabilities makes it versatile for both low-level gate descriptions and high-level algorithmic designs.

Verilog is essential in FPGA workflows for the following purposes:

- **RTL Modeling of Digital Logic:** Verilog allows designers to express the functional behavior of digital systems using constructs that represent registers, combinational logic, and control logic. RTL modeling captures how data moves between registers under control of clocks and conditions.
- **Module Design and Instantiation:** Hierarchical and reusable design is achieved through modules. Designers can define, connect, and instantiate hardware blocks, which promotes modularity and scalability in system architecture.
- **Simulation and Debugging:** Testbenches written in Verilog simulate the functionality of designs before synthesis. System tasks like `$display`, `$monitor`, and waveform generation are used to verify correctness and catch logic errors early in the design process.
- **Synthesis and Optimization:** Synthesizable Verilog code is transformed into a netlist of logic gates and flip-flops that FPGA tools can map to physical hardware. Synthesis tools also optimize logic to improve area, speed, and power efficiency.

Verilog HDL bridges the gap between abstract design ideas and tangible hardware implementations, making it an indispensable tool for FPGA-based digital system development.

1.11 FPGA Boards and Development Kits

FPGA development boards are essential platforms that allow students, engineers, and researchers to prototype and test digital designs. These kits provide access to real hardware, supporting everything from basic logic design to embedded systems development.

Popular FPGA Development Boards

- **Basys 3 (Xilinx Artix-7):** Entry-level FPGA board designed for students and hobbyists. It includes a rich set of I/O components for learning digital logic.
- **Nexys A7:** A more advanced board also based on the Xilinx Artix-7. It offers more switches, memory, and peripherals, making it suitable for system-level design.
- **DE10-Lite (Intel MAX10):** A compact and affordable FPGA board from Intel (Altera), often used in academic courses and digital labs.
- **Zybo Z7 (Zynq SoC):** A powerful board featuring a Xilinx Zynq SoC, which combines FPGA logic with an ARM processor. Ideal for embedded systems and hardware-software co-design.

Common Onboard Features

Most FPGA kits include the following built-in components for interfacing and experimentation:

- **Input/Output Devices:** Switches, push-buttons, LEDs, and 7-segment displays for basic digital I/O.
- **Display Interfaces:** VGA or HDMI ports for outputting video signals from FPGA logic.
- **Expansion Connectors:** PMOD (Peripheral Module) headers allow users to add sensors, motor controllers, or communication modules.

These boards help bridge the gap between Verilog code and working digital systems, providing a hands-on environment for learning and innovation. Table 1.4 provides a comparison of widely used FPGA development boards, highlighting their core FPGA chips, logic capacities, and integrated peripherals to help users select the appropriate platform for learning, prototyping, or system design.

Table 1.4: Comparison of popular FPGA development boards

Board	**FPGA Chip**	**Logic Cells**	**Key Peripherals**
Basys 3	Xilinx Artix-7 XC7A35T	33,280	16 switches, 5 buttons, 16 LEDs, 4-digit 7-segment, PMODs, USB-UART
Nexys A7	Xilinx Artix-7 XC7A100T	101,440	16 switches, 16 LEDs, 7-segment, VGA, Audio out, USB-UART, PMODs, 10/100 Ethernet
DE10-Lite	Intel MAX10 10M50DAF484C7G	50,000 (LEs)	10 switches, 10 LEDs, 7-segment, USB-Blaster II, Arduino header, ADC inputs
Zybo Z7	Xilinx Zynq-7000 XC7Z010/20	28,000 / 85,000	ARM Cortex-A9, HDMI I/O, USB OTG, Ethernet, Audio codec, DDR3 RAM, PMODs

1.12 Applications of FPGA-Based Designs

FPGAs offer high flexibility, parallelism, and performance, making them ideal for a wide range of applications across industries. Their reconfigurable nature allows rapid prototyping, hardware acceleration, and real-time responsiveness.

- **Digital Communication Systems:** FPGAs are widely used in wireless base stations, software-defined radios (SDRs), and high-speed networking equipment for tasks such as encoding, decoding, modulation, and packet processing.
- **Audio and Video Processing:** With their high-speed parallel architecture, FPGAs efficiently handle real-time video filtering, scaling, encoding/decoding (e.g., H.264), and audio effects in multimedia applications.
- **Real-Time Control Systems:** Used in industrial automation, automotive ECUs, robotics, and aerospace, FPGAs provide deterministic control and low-latency decision-making, often interfacing directly with sensors and actuators.
- **AI and ML Acceleration:** FPGAs are increasingly deployed for accelerating inference tasks in deep learning models. They offer customizable compute engines optimized for convolution, matrix multiplication, and logic-intensive operations.
- **Custom Processor Implementations:** Designers can implement RISC-V cores, DSP engines, or domain-specific accelerators directly on FPGAs to meet application-specific performance, power, or integration goals.

These capabilities make FPGAs an essential platform in domains where performance, flexibility, and rapid development are crucial.

1.13 Challenges in Digital Design

Designing complex digital systems involves several technical and practical challenges that engineers must address to ensure functional correctness, performance, and efficiency. As systems grow in complexity, the following issues become more prominent:

- **Meeting Timing Constraints:** Ensuring that all signals propagate and settle within specified clock periods is critical. Violations can cause data corruption, setup/hold time errors, and unpredictable behavior.
- **Power Consumption:** As devices integrate more functionality, managing dynamic and static power becomes essential—especially for battery-powered and embedded systems. Designers must balance performance with energy efficiency.
- **Debugging Complex Systems:** Observability is often limited in hardware, making bugs harder to detect and reproduce. Debugging tools such as logic analyzers, on-chip trace units, and simulation probes are often required.
- **Verification Complexity:** With growing design size and concurrency, verifying all possible states, transitions, and behaviors becomes difficult. Simulation, formal verification, and coverage metrics must be applied to ensure correctness.

Overcoming these challenges requires experience, robust toolchains, and good design practices to ensure that digital systems are reliable, efficient, and ready for deployment.

1.14 Future Trends in Digital Systems and FPGAs

As digital design technologies evolve, FPGAs continue to expand their role across emerging computing paradigms. Their reconfigurable nature, parallel processing capability, and adaptability position them at the forefront of future computing trends.

- **Increasing Use of HLS (High-Level Synthesis):** High-Level Synthesis tools allow designers to write algorithms in C/C++ or Python and automatically convert them to synthesizable RTL code. This trend accelerates design cycles, promotes software-hardware co-design, and lowers the barrier to FPGA adoption.
- **Integration with AI and DSP Systems:** FPGAs are increasingly deployed for AI inference and DSP tasks due to their low-latency and parallelism advantages. Many vendors now provide AI/ML IP blocks and frameworks for FPGA-based acceleration.
- **Reconfigurable Computing:** Dynamic Partial Reconfiguration enables parts of an FPGA to be reprogrammed on-the-fly while other regions continue operation. This allows flexible, multi-function designs optimized for time-multiplexed workloads.
- **FPGA in Edge and Embedded Computing:** With growing demand for real-time, low-power computation at the edge (e.g., in autonomous vehicles, IoT, and robotics), FPGAs provide a programmable, efficient solution for running custom logic close to the data source.

These trends illustrate how FPGAs are not only keeping pace with but actively driving innovation in the fields of AI, embedded systems, and future computing architectures.

1.15 Summary

This chapter provided a comprehensive overview of digital design principles and the foundational role of FPGAs in modern hardware systems. Key topics covered include the binary number system, Boolean algebra, and the distinction between combinational and sequential logic.

HDLs, particularly Verilog HDL, are introduced with emphasis on their syntax, modeling styles, and practical applications in digital system design. A comparison with VHDL helped contextualize Verilog's role in the industry.

The architecture of FPGAs was discussed, emphasizing their key components such as CLBs, routing fabric, I/O blocks, block RAM, and DSP slices. The standard design flow—from HDL coding to bitstream generation and FPGA programming—is also outlined.

In addition, the chapter reviewed simulation and verification tools, popular FPGA development boards, abstraction layers, and real-world applications. The chapter concludes with a discussion of challenges in digital design and emerging trends such as HLS, AI integration, and edge computing.

Together, these foundational concepts set the stage for deeper exploration of Verilog-based design and synthesis workflows in the chapters that follow.

The laboratory exercises for Chapter 1: *Digital Design and FPGA* consist of Lab 1: Introduction to Xilinx Vivado and Lab 2: Getting Started with the Xilinx Nexys A7, which are provided in the Appendix section.

1.16 Exercises

1. What is the difference between combinational and sequential logic? Provide examples of each.
2. Explain the role of HDLs in digital design. Why are they important?
3. Compare Verilog and VHDL in terms of syntax, typing, and typical use cases.
4. What is an FPGA, and how does it differ from an ASIC and a microcontroller?
5. List and briefly describe the major components of an FPGA architecture.
6. Outline the typical FPGA design flow from specification to programming.
7. What are some challenges commonly faced in digital design?
8. Describe three real-world applications of FPGA-based designs.
9. How is HLS changing the way hardware is developed?
10. What trends are shaping the future of FPGA usage in embedded and AI systems?

Bibliography

[1] C. Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes, 2004.

[2] M. Manohar and J. Bhasker, *Digital System Design Using FPGA: Implementation with Verilog and VHDL*. New York, NY, USA: McGraw-Hill, 2017.

[3] R. Merrick, *Getting Started with FPGAs: Digital Circuit Design, Verilog, and VHDL for Beginners*. San Francisco, CA: No Starch Press, 2023.

[4] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, Pearson Education, 2003.

[5] Digilent Inc., "What is an FPGA?," 2019. [Online]. Available: https://digilent.com/blog/what-is-an-fpga/

[6] Xilinx Inc., *Vivado Design Suite User Guide*, 2023. [Online]. Available at: https://www.xilinx.com/products/design-tools/vivado.html

[7] Intel Corporation, *Intel Quartus Prime Pro Edition User Guide*, 2023. [Online]. Available at: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/download.html

[8] Siemens EDA (Mentor Graphics), *ModelSim User's Manual*, 2021. [Online]. Available at: https://eda.sw.siemens.com/en-US/ic/model/

[9] Stephen Williams and contributors, *Icarus Verilog and GTKWave Open Source Tools*, 2024. [Online]. Available online: http://iverilog.icarus.com and http://gtkwave.sourceforge.net

[10] Digilent Inc., *Basys 3 and Nexys A7 FPGA Boards*, [Online]. Available: https://digilent.com

[11] Terasic Technologies, *DE10-Lite User Manual*, [Online]. Available: https://www.terasic.com.tw

Chapter 2

Fundamentals of Verilog HDL

Chapter Objectives

- Understand the syntax and structure of Verilog HDL.
- Apply gate-level, dataflow, and behavioral modeling styles.
- Simulate and verify Verilog designs using testbenches.

2.1 Introduction to Verilog HDL

Verilog HDL is a standardized and widely adopted language for modeling, designing, and verifying digital electronic systems. It enables designers to describe hardware at various abstraction levels, from simple logic gates to complex integrated circuits and processor architectures.

Originally developed to simulate digital systems, Verilog has evolved into a critical tool for designing hardware for both ASIC and FPGA platforms. Its concise syntax, structural clarity, and simulation support make it a preferred choice in both industry and academia.

This chapter introduces the fundamental aspects of Verilog HDL, including its lexical structure, data types, modeling styles (gate-level, dataflow, and behavioral), and simulation constructs. Understanding these foundational elements is essential for writing efficient, synthesizable code and building reliable digital systems.

2.2 History and Motivation

Verilog HDL was originally developed in 1984 by Gateway Design Automation as a proprietary language for modeling digital circuits. Its main purpose was to allow designers to simulate and verify hardware behavior before physically implementing the circuits.

The language quickly gained popularity due to its compact syntax, its similarity to the C programming language, and its support for both behavioral and structural modeling.

As digital systems became increasingly complex, there was a need for standardized hardware description practices. In response, Verilog was standardized by the IEEE in 1995 under the designation IEEE 1364. This formalization made Verilog an industry-accepted tool for design and verification, enhancing portability, tool support, and design reusability.

The shift from gate-level design to RTL design further emphasized the importance of HDLs like Verilog. RTL design enabled engineers to work at a higher level of abstraction, focusing on data movement between registers and functional blocks rather than manually wiring individual gates. This transition significantly improved productivity, scalability, and the ability to design complex, modular digital systems.

Verilog's widespread adoption in industry stems from its versatility—it supports simulation, synthesis, testbench creation, and hardware verification. It is used extensively in the design of both ASICs and FPGAs. From simple logic gates to sophisticated system-on-chip (SoC) designs, Verilog remains a foundational tool in modern digital electronics.

In recent years, Verilog has been extended through SystemVerilog (IEEE 1800), which adds advanced verification and object-oriented features. Despite these advancements, traditional Verilog remains a core component of digital design education and practice.

2.3 Structure of a Verilog Design

A Verilog design is composed of modular units that describe hardware behavior and interconnections. These units work together to form a complete digital system. Understanding the basic structure of a Verilog design is essential for writing readable, synthesizable, and reusable code.

Figure 2.1 illustrates the hierarchical structure of a Verilog design, showing a top-level module containing two interconnected submodules, with clearly defined input and output ports.

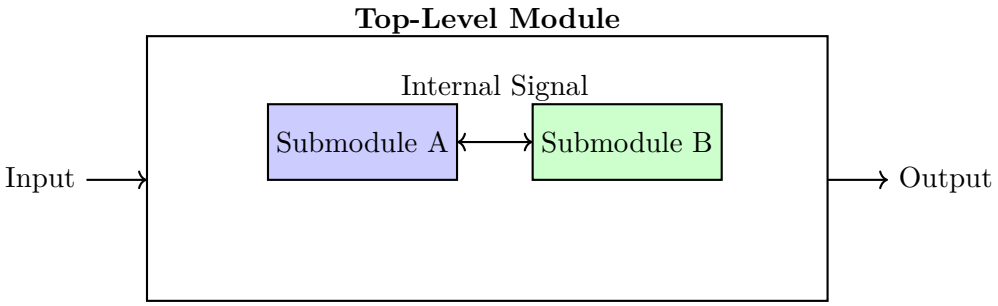


Figure 2.1: Verilog module hierarchy and port directions

A typical Verilog design consists of the following elements:

- **Modules:** These are the fundamental building blocks in Verilog. A module defines a circuit or subcircuit, encapsulating its functionality and internal implementation. Modules can be instantiated hierarchically, allowing complex designs to be built from simpler components.

- **Ports:** Ports define the interface of a module. They declare the signals that connect the module to the outside world. Ports can be inputs, outputs, or bidirectional (inouts).

- **Data Types:** Verilog provides several data types to describe digital values. Common types include `wire` for connections and `reg` for storage elements. Verilog supports 4-state logic: `0` (low), `1` (high), `z` (high impedance), and `x` (unknown).

- **Behavioral Constructs:** Verilog includes high-level programming constructs such as `if-else`, `case`, `for` loops, and `always` blocks. These are used to describe how the logic behaves over time, especially for sequential and conditional operations.

2.3.1 Basic Module Structure

The following is a simple example of a Verilog module implementing a 2-input AND gate. It illustrates the core elements of a Verilog design: the module declaration, port list, and internal logic definition.

```
1 module AND_gate(input a, b, output y);
2   assign y = a & b;
3 endmodule
```

In this example:

- `module AND_gate` declares the module name.

- `input a, b` and `output y` define the interface ports.

- `assign y = a & b;` is a continuous assignment using dataflow modeling to express the logical AND operation.

This basic structure can be extended to more complex modules by including internal wires, registers, conditional logic, hierarchical submodules, and procedural blocks. As designs grow in complexity, maintaining a clean and modular structure becomes critical for verification, synthesis, and debugging.

2.4 Lexical Elements and Syntax

Understanding the lexical elements and basic syntax of Verilog is essential for writing valid and effective hardware descriptions. These elements define how the language is structured and how various components are named, commented, and expressed.

- **Identifiers:**
  Identifiers are names used to label modules, variables, ports, wires, registers, and other entities in a Verilog design. An identifier must begin with a letter (a–z or A–Z) or an underscore (_) and can contain letters, digits (0–9), underscores, and the dollar sign ($). Identifiers are case-sensitive.

```
module ALU;           // Valid identifier
wire result_line;     // Valid identifier
reg temp1;            // Valid identifier
```

- **Comments:**
  Verilog supports two styles of comments:
  - `//` for single-line comments
  - `/* ... */` for multi-line or block comments

  Comments are ignored during synthesis and simulation but are crucial for documentation and readability.

```
// This is a single-line comment
/*
  This is a
  multi-line comment
*/
```

- **Numbers:**
  Verilog allows constants to be expressed in multiple bases with optional size specifications. The general format is: `[size]'[base][value]`
  - `b` or `B` – binary
  - `d` or `D` – decimal
  - `o` or `O` – octal
  - `h` or `H` – hexadecimal

```
4'b1010      // 4-bit binary value
8'd255       // 8-bit decimal value
12'o77       // 12-bit octal value
16'h1A3F     // 16-bit hexadecimal value
```

The underscore (_) can also be used to visually separate digits (e.g., `32'hDEAD_BEEF`).

- **Operators:**
  Verilog provides a wide range of operators for arithmetic, logic, comparison, bit manipulation, and shifting:
  - **Arithmetic:** `+, -, *, /, %`
  - **Relational:** `==, !=, >, <, >=, <=`
  - **Logical:** `&&, ||, !`
  - **Bitwise:** `&, |, ^, ~`
  - **Shift:** `«, »`

  Example:

```
assign y = (a & b) | (~c);
assign equal = (x == y);
```

Table 2.1: Summary of common Verilog operators

Category	**Operators**
Arithmetic	+, -, *, /, %
Relational	==, !=, >, <, >=, <=
Logical (Boolean)	&&, |, !
Bitwise	&, |, ^, ~
Shift	<<, >>
Equality (4-state)	===, !==
Reduction	&, |, ^, ~&, ~|, ~^
Concatenation	{}
Replication	{n{...}}
Conditional	? : (ternary operator)

These lexical components form the basic building blocks of Verilog syntax. Mastery of identifiers, comments, numbers, and operators allows designers to write clear, compact, and functional Verilog code that can be synthesized into reliable hardware. Table 2.1

summarizes the most commonly used Verilog operators, grouped by category, to assist designers in writing expressions for arithmetic, logical decisions, bit manipulation, and conditional assignments.

2.5 Verilog Data Types

Verilog provides a set of built-in data types to describe and model digital systems. These data types help define how signals behave, how values are stored, and how components are interconnected. Two of the most fundamental data classifications in Verilog are nets and registers, which differ in how they represent and maintain signal values.

2.5.1 Nets vs. Registers

Verilog separates signal types based on how they are driven and where they are used. The two most commonly used types are `wire` and `reg`.

- **wire:**
  A `wire` is used to represent physical connections between structural elements like gates, modules, or continuous assignments. It does not hold a value by itself but reflects the value being driven onto it. `wire` must be assigned by continuous assignment (e.g., using `assign`) or by outputs of module instances or gate primitives.

```
wire a, b, y;
assign y = a & b;  // y is continuously driven by AND of a
   and b
```

- **reg:**
  A `reg` holds its value until explicitly changed in a procedural block (e.g., `always` or `initial`). Despite the name "reg", it does not always represent a hardware register unless used in sequential logic with clocking. It is often used in behavioral modeling to store values during simulation cycles.

```
reg flag;
always @(posedge clk)
  flag <= ~flag; // flag is updated on clock edge
```

2.5.2 Vectors and Arrays

Verilog allows multi-bit signals to be grouped as vectors, and also supports arrays of vectors or scalars for representing structured data.

- **Vectors:** A vector is a single multi-bit signal. Vectors are useful for buses and data paths. The range is specified in square brackets, with either little-endian (e.g., `[0:7]`) or big-endian (e.g., `[7:0]`) indexing.

```
reg [7:0] byte_data; // 8-bit wide register
wire [3:0] select_lines; // 4-bit wire
```

- **Arrays:** Arrays are collections of scalars or vectors, indexed by an additional dimension. Arrays are useful for modeling memories, register files, or multiple data channels.

```
reg [3:0][7:0] memory; // 4 elements, each 8 bits wide
```

  In SystemVerilog, support for multidimensional arrays and advanced indexing is extended further, but basic Verilog supports 1D packed arrays using this syntax.

In Verilog, data types play a crucial role in modeling hardware correctly and efficiently. The `wire` type is used for physical interconnections and continuous assignments, making it suitable for connecting gates, modules, and simple combinational logic. The `reg` type is used within procedural blocks to store values across simulation steps and is essential in sequential logic designs. Vectors allow designers to group multiple bits into a single signal, such as a data bus or control word, while arrays enable the definition of multiple instances of similar signals, useful for modeling memory blocks or sets of registers. Understanding when and how to use each of these data types ensures that Verilog code remains both synthesizable and logically sound.

2.6 Modeling Styles

Verilog HDL supports multiple abstraction levels for describing digital systems. These abstraction levels are reflected through three primary modeling styles: Gate-Level, Dataflow, and Behavioral. Each modeling style provides different capabilities and is suitable for different phases of the design process—from low-level implementation to high-level algorithmic description.

2.6.1 Gate-Level Modeling

Gate-level modeling is the most detailed and lowest level of abstraction in Verilog. It involves instantiating primitive logic gates directly to build circuits. Verilog provides built-in gate primitives such as `and`, `or`, `not`, `nand`, `nor`, `xor`, and `xnor`. This style is closest to the actual physical implementation of digital logic.

```
and (out1, a, b);    // out1 = a AND b
or  (out2, x, y);    // out2 = x OR y
not (inv, a);        // inv = NOT a
```

Gate-level modeling is primarily used for structural descriptions, simulation of existing netlists, or when fine-grained control of logic is necessary. However, for large or complex designs, this style becomes cumbersome and is often avoided in favor of higher-level modeling.

2.6.2 Dataflow Modeling

Dataflow modeling describes how data flows through a circuit using logical or arithmetic expressions. It typically uses the `assign` keyword for continuous assignments and is useful for modeling combinational logic.

```
assign y = a & b;    // Bitwise AND using dataflow
assign sum = a + b; // Arithmetic addition
```

Dataflow modeling strikes a balance between readability and low-level control. It is especially useful for implementing logic equations, multiplexers, arithmetic units, and other purely combinational circuits without explicitly instantiating gates.

2.6.3 Behavioral Modeling

Behavioral modeling is the highest level of abstraction in Verilog. It allows designers to describe what a system does, rather than how it is implemented. This style uses procedural constructs such as `always` and `initial` blocks, along with control statements like `if-else`, `case`, and loops.

```
always @(posedge clk) begin
  count <= count + 1;    // Count on rising edge
end
```

Behavioral modeling is ideal for describing complex control logic, finite state machines (FSMs), and sequential circuits. It allows for easy simulation, debugging, and early-stage algorithm development. However, designers must ensure that behavioral descriptions are synthesizable if the target is hardware implementation.

Choosing the Right Modeling Style

Designers often use a combination of modeling styles in the same project. For example, arithmetic operations might be implemented using dataflow modeling, control logic with behavioral modeling, and lower-level hardware components using structural (gate-level)

modeling. Choosing the appropriate style depends on the design requirements, desired abstraction, simulation needs, and synthesis targets.

Note

The following summarizes the key characteristics of the three primary Verilog modeling styles:

- **Gate-Level Modeling:** Describes circuits using built-in logic gate primitives (e.g., `and`, `or`, `not`). It is best suited for low-level, structural designs and netlist representations.
- **Dataflow Modeling:** Uses continuous assignment statements (`assign`) to describe how data moves through the system. This style is ideal for implementing Boolean expressions and arithmetic operations in combinational logic.
- **Behavioral Modeling:** Relies on procedural constructs such as `always` blocks and control flow statements (`if`, `case`, etc.). It is typically used to model control logic and sequential systems where clocked behavior is involved.

Understanding these modeling styles provides the flexibility to describe, simulate, and synthesize digital systems efficiently and accurately in Verilog.

2.7 Procedural Blocks

Procedural blocks in Verilog allow designers to describe hardware behavior using a sequence of statements that execute in response to events. These blocks are fundamental in behavioral modeling and are used to define both combinational and sequential logic. Verilog provides two main types of procedural blocks: `always` and `initial`.

2.7.1 always Block

The `always` block is one of the most powerful constructs in Verilog. It executes repeatedly throughout simulation whenever one or more of its sensitivity list signals change. It can be used to describe both sequential and combinational behavior depending on how it's written.

For sequential logic, the sensitivity list typically includes a clock edge:

```
1 always @(posedge clk or posedge rst)
2   if (rst)
3     q <= 0;
4   else
5     q <= d;
```

In this example:

- The block is triggered on the positive edge of `clk` or `rst`.
- If `rst` is high, the output `q` is reset to 0.
- Otherwise, `q` takes the value of `d` on the next rising clock edge.

The `<=` symbol denotes a non-blocking assignment, which is essential in sequential logic to avoid race conditions and to reflect actual hardware behavior accurately.

For combinational logic, the `always` block may use the `@(*)` sensitivity list and blocking assignments:

```
always @(*)
  y = a & b;
```

2.7.2 initial Block

The `initial` block is used to specify behavior that runs only once at the beginning of simulation. It is not synthesizable and is typically used in testbenches, stimulus generation, or variable initialization.

```
initial begin
  clk = 0;
  forever #5 clk = ~clk;
end
```

In this example:

- The signal `clk` is initialized to 0 at time 0.
- The `forever` loop toggles the clock every 5 time units, simulating a square wave.

Key Differences

- **`always`** blocks run continuously and respond to changes in specified signals.
- **`initial`** blocks run only once at the start of simulation.
- `always` blocks can be synthesized into hardware; `initial` blocks are simulation-only.
- Use non-blocking assignments (`<=`) for sequential logic; use blocking assignments (`=`) for combinational logic.

Table 2.2 provides a concise comparison between the `always` and `initial` blocks in Verilog, highlighting their differences in execution behavior, synthesis support, and typical usage in digital design.

Table 2.2: Comparison of `always` and `initial` blocks

Feature	`always`	`initial`
Execution	Repeats on events	Runs once at start
Use Case	Hardware behavior	Simulation-only
Synthesizable	Yes	No
Sensitivity	Required (`@`)	Not required
Applications	Logic, FSMs	Clocks, setup
Assignments	`=` / `<=`	Mostly `=`
Triggers	Event-driven	Time 0 only

Example Usage

The following code examples illustrate how `always` and `initial` blocks are typically used in Verilog designs:

Example of an `always` block for sequential logic:

```
always @(posedge clk or posedge rst)
  if (rst)
    q <= 0;
  else
    q <= d;
```

This describes a synchronous resettable D flip-flop.

Example of an `initial` block for clock generation:

```
initial begin
  clk = 0;
  forever #5 clk = ~clk;
end
```

This example generates a 10-time-unit clock period signal, useful for testbenches.

Procedural blocks provide the necessary control structure for describing time-dependent behavior in Verilog. They are central to modeling both synchronous and asynchronous digital systems. A solid understanding of how to use `always` and `initial` blocks allows designers to write testbenches, implement finite state machines, and model complex logic with precision and clarity.

2.8 Conditional and Looping Constructs

Verilog supports a set of control flow constructs that enable conditional decision-making and repeated execution within procedural blocks. These constructs—`if-else`, `case`, and loops like `for`—allow designers to describe logic behavior in a compact and readable manner. They are typically used in behavioral modeling with `always` or `initial` blocks.

2.8.1 if-else

The `if-else` statement allows conditional execution of statements based on a Boolean expression. It is widely used for decision-making in control logic and sequential circuits.

```
if (enable)
  y = a;
else
  y = b;
```

In this example, the value of `y` is selected based on the value of `enable`. If `enable` is true, `y` is assigned the value of `a`; otherwise, it is assigned `b`.

2.8.2 case Statement

The `case` statement is used to select among multiple possible values of a control expression. It is similar to a switch-case construct in software programming languages and is useful for implementing multiplexers, state machines, and decoders.

```
case (sel)
  2'b00: y = a;
  2'b01: y = b;
  2'b10: y = c;
  default: y = 0;
endcase
```

`case` statements improve readability and help avoid deeply nested `if-else-if` chains. It is important to include a `default` clause to ensure all possible conditions are covered, avoiding unintended latch inference.

2.8.3 for Loops

The `for` loop is a control structure that repeats a block of code for a specified number of iterations. While not synthesizable in general-purpose logic, it is often used in testbenches and in `generate` constructs for hardware replication.

```
for (i = 0; i < 4; i = i + 1)
  sum = sum + data[i];
```

This loop accumulates values from an array `data` into `sum`. In synthesizable code, `for` loops are primarily used to generate repetitive hardware structures, such as multiple flip-flops or logic cells, especially within `generate` blocks.

Best Practices

- Use `if-else` for binary decisions and simple control logic.
- Use `case` for multi-way branching and clear selection logic.
- Always include a `default` in `case` to avoid latches.
- Use `for` loops with care in synthesizable designs—ensure bounds and loop control are fixed and deterministic.

Table 2.3 compares the if-else and case constructs in Verilog, highlighting their ideal use cases, synthesis behavior, and common pitfalls in control logic design.

Table 2.3: Comparison of `if-else` and `case` constructs

Feature	`if-else`	`case`
Use Case	Conditions, flags	Selection, decoding
Readability	Less with many branches	Better for large choices
Conditions	Complex allowed	Single expression match
Synthesis	Priority logic	Parallel logic
Default	`else` optional	`default` recommended
Common Pitfall	Missing `else` infers latch	Incomplete cases infer latch

Conditional and looping constructs enhance the expressiveness of Verilog, making it easier to describe complex behaviors and control flows. Proper use of these constructs is essential for writing clean, efficient, and synthesizable HDL code.

2.9 Tasks and Functions

Tasks and functions in Verilog are reusable blocks of code used to modularize operations, enhance readability, and reduce duplication. They are similar to subroutines in software programming and are particularly useful in behavioral modeling and testbench development.

Functions

A Verilog `function` performs a computation and returns a single value. Functions execute in zero simulation time and cannot contain time-controlling statements (like `#delay`, `@`, or `wait`). They are typically used for combinational logic operations such as arithmetic, encoding, or parity checks.

```
function [7:0] add;
  input [7:0] a, b;
  begin
    add = a + b;
  end
endfunction
```

In this example:

- The function `add` returns an 8-bit value.
- Inputs `a` and `b` are 8-bit operands.
- The function computes and returns their sum.

Functions can be called within continuous assignments, procedural blocks, or expressions:

```
assign result = add(operand1, operand2);
```

Tasks

A `task` is similar to a function but is more versatile. Unlike functions, tasks can:

- Have zero or multiple return values (via `output` or `inout` arguments)
- Contain timing control (e.g., delays, events)
- Perform complex sequences of operations

Tasks are useful for simulation-only operations like test vector application, printing, and waveform monitoring, or for modeling complex logic sequences in behavioral designs.

```
task display_result;
  input [7:0] result;
  begin
    $display("The result is: %d", result);
  end
endtask
```

Tasks must be invoked from within a procedural block (such as `initial` or `always`):

```
initial begin
  display_result(8'd255);
end
```

Comparison of Functions and Tasks

Table 2.4 summarizes the key differences between Verilog functions and tasks in terms of return values, timing control, usage context, and purpose. This distinction is important for designing modular and efficient hardware descriptions.

Table 2.4: Comparison of Verilog functions and tasks

Feature	**Function**	**Task**
Return Values	One	Zero or more (via outputs)
Timing Control	Not allowed	Allowed (#, @)
Usage Context	Combinational logic	Complex sequences, testbenches
Call Context	Anywhere (including assign)	Only in procedural blocks
Purpose	Computation	Action/Procedure

Best Practices

- Use functions for simple, side-effect-free computations.
- Use tasks when delays or multiple results are needed.
- Avoid unnecessary timing control in synthesis-targeted code.
- Modularize commonly used code to improve maintainability.

Synthesis-Compatible Examples

While tasks and functions are often used in simulation, they can also be written to be compatible with synthesis when adhering to specific constraints. The following examples demonstrate synthesis-friendly usage:

Synthesizable Function Example (8-bit Adder)

```
function [7:0] add8;
  input [7:0] a, b;
  begin
    add8 = a + b;
```

```
  end
endfunction
```

Usage:

```
always @(*) begin
  sum = add8(x, y);  // Combinational logic
end
```

Synthesizable Task Example (2-to-1 Multiplexer)

```
task mux2to1;
  input [7:0] in0, in1;
  input sel;
  output [7:0] out;
  begin
    if (sel)
      out = in1;
    else
      out = in0;
  end
endtask
```

Usage:

```
always @(*) begin
  mux2to1(a, b, select, y);  // Multiplexer logic
end
```

Synthesis Guidelines

- Avoid delays (`#`), event controls (`@`), and system tasks (`$display`) in synthesis.
- Functions must return a single value and cannot have side effects.
- Tasks may have multiple outputs and can encapsulate more complex logic structures.
- Both must be called inside procedural blocks (e.g., `always`) for synthesis.

These examples show how tasks and functions can enhance code modularity without compromising synthesizability, particularly in combinational designs or structured data paths.

Tasks and functions are powerful tools in Verilog that enhance code modularity and abstraction. While functions are preferred for pure logic evaluation, tasks are ideal for simulations, testbenches, and more complex operations. Proper use of these constructs leads to cleaner, more maintainable, and reusable HDL code.

2.10 Timing Control

Timing control statements in Verilog allow the simulation of time-dependent behaviors, making it possible to model delays, synchronization, and sequencing of events. These constructs are critical for testbenches, clock generation, and stimulus application—but are generally not synthesizable and are used primarily for simulation.

2.10.1 # Delay

The `#` operator introduces a delay in simulation time before the next statement executes. It is useful for simulating clock pulses, input stimulus, or modeling propagation delay.

```
1 #5 clk = ~clk;
```

In this example, the signal `clk` is toggled every 5 time units, which could represent a simulated clock of 10 time units period. This construct is often used in clock generation blocks within testbenches.

Note: Delay control using `#` is not synthesizable and should not be used in synthesizable RTL design.

2.10.2 Event Control

Event control waits for a specific change or condition on a signal before executing the next statement. It is commonly used to model behavior triggered by clock edges or changes in input signals.

```
1 @(posedge clk)
```

This construct tells the simulator to wait until the rising edge of `clk`. It is typically used in sequential logic or testbenches to synchronize with clock cycles.

A more complete example:

```
1 always @(posedge clk)
2   q <= d;
```

This describes a positive edge-triggered D flip-flop.

Types of Event Control

- `@(posedge clk)` – Waits for rising edge of `clk`
- `@(negedge rst)` – Waits for falling edge of `rst`
- `@(a or b)` – Waits for a change in either `a` or `b`
- `@*` or `@(*)` – Infers sensitivity to all right-hand side variables (used for combinational logic)

Usage Context

- Use `#` delays in `initial` blocks for waveform generation and stimulus control.
- Use `@(posedge clk)` in `always` blocks for writing synchronous sequential logic.
- Avoid both constructs in synthesizable logic unless modeling specific delays in simulation only.

Timing control is essential for verifying the behavior of digital circuits under time-dependent conditions. While these constructs are powerful tools in simulation, designers must carefully avoid their use in synthesizable RTL code, except where permitted by synthesis tools for testbenches or constraints.

2.11 Testbenches and Simulation

A testbench is a Verilog module used to verify the correctness of a design by simulating how the circuit behaves under different input conditions. Unlike synthesizable modules, testbenches are not synthesized onto hardware but serve as a powerful tool to validate functionality through simulation.

Purpose of a Testbench

A well-constructed testbench typically performs the following tasks:

- **Instantiates** the DUT
- **Applies** various input stimulus to the DUT
- **Monitors** and evaluates output responses for correctness

Basic Testbench Example

The following is a simple testbench that applies sequential input values to a DUT:

```
module test_example;
  reg a, b;
  wire y;

  // Instantiate DUT
  and_gate dut (.a(a), .b(b), .y(y));

  initial begin
    a = 0; b = 0;
    #10 a = 1;
    #10 b = 1;
    #10 a = 0;
    #10 $finish;
  end
endmodule
```

Key Components of a Testbench

- **Initial Blocks:** Control the sequence of test vectors using delays.
- **Waveform Monitoring:** Use system tasks like `$monitor` and `$display` to observe values.
- **Clock Generation:** Simulate clocks using toggling constructs in `initial` or `always` blocks.
- **Assertions:** Check output values against expected results (in advanced testbenches).

Example with Monitoring

```
initial begin
  $monitor("Time=%0t : a=%b b=%b => y=%b", $time, a, b, y);
end
```

This code continuously prints signal values when any of them change during simulation.

Simulation Tools

To run testbenches, simulation tools like ModelSim, XSIM (Vivado), Icarus Verilog, or Synopsys VCS are used. These tools:

- Compile both the DUT and testbench files
- Run time-based simulations
- Generate waveforms for analysis (using `$dumpfile`, `$dumpvars`)

Testbenches are essential for verifying digital circuit functionality before hardware implementation. By simulating the behavior of the DUT with various input scenarios, designers can catch logic errors, ensure design correctness, and iterate rapidly—making testbenches a fundamental part of HDL design flow.

2.12 Design Hierarchy and Instantiation

Verilog designs are inherently modular and hierarchical. This structure allows complex systems to be built from smaller, reusable building blocks. Each block is defined as a `module`, and larger designs are constructed by instantiating these modules within one another.

2.12.1 Module Instantiation

Module instantiation is the process of creating an instance of a lower-level module inside a higher-level module. This allows for structured design, code reuse, and clearer abstraction of system components.

```
1 AND_gate u1 (.a(x), .b(y), .y(z));
```

In this example:

- `AND_gate` is the name of the module being instantiated.
- `u1` is the unique instance name.
- `.a(x)` connects the internal port `a` of `AND_gate` to the signal `x` in the parent module.
- `.b(y)` and `.y(z)` follow the same pattern.

2.12.2 Design Hierarchy

Design hierarchy allows developers to build systems in layers:

- A **leaf module** contains only behavioral or structural logic.
- A **parent module** may instantiate multiple submodules.
- The **top-level module** is the highest point of the hierarchy and is used for synthesis or simulation entry.

Hierarchical designs enhance:

- **Reusability** – Modules can be reused in other projects.
- **Maintainability** – Each module can be developed and tested independently.
- **Scalability** – Large designs become manageable through decomposition.

Example: 2-bit AND Using Hierarchy

To build a 2-bit AND gate, the 1-bit `AND_gate` module can be reused.

Leaf Module:

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

Top Module:

```
module top_module(input [1:0] A, B, output [1:0] Y);
  AND_gate u0 (.a(A[0]), .b(B[0]), .y(Y[0]));
  AND_gate u1 (.a(A[1]), .b(B[1]), .y(Y[1]));
endmodule
```

Hierarchy Diagram

Hierarchy and module instantiation are key to creating scalable Verilog designs. By breaking down functionality into smaller modules, designers can improve clarity, simplify debugging, and increase reusability.

Figure 2.2 shows a top-level module, `top_module`, that instantiates two identical submodules: `AND_gate u0` and `AND_gate u1`. This demonstrates the modular reuse approach common in Verilog design.

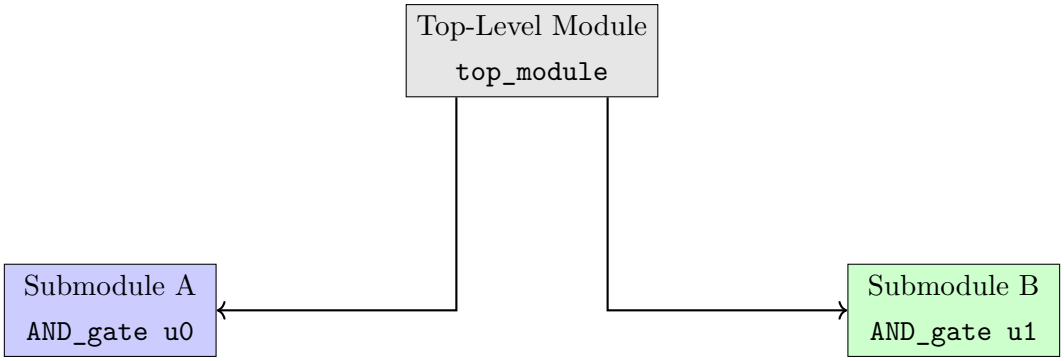


Figure 2.2: Design hierarchy of a top-level module instantiating two submodules

This approach supports a clean separation of functionality, making the design both scalable and maintainable as complexity grows.

2.13 System Tasks

System tasks in Verilog provide built-in capabilities for debugging, simulation control, and output monitoring. These are prefixed with a dollar sign (`$`) and are used exclusively for simulation—they are not synthesizable.

Common System Tasks

- **$display:** Prints formatted output to the simulator console.
- **$monitor:** Continuously monitors and displays values when any change occurs.
- **$dumpfile:** Specifies the output file name for waveform dumping (e.g., for GTKWave).
- **$dumpvars:** Specifies which variables/signals to record in the waveform file.

Example: Display and Monitor

```
initial begin
  $display("Time=%0t | a=%b b=%b y=%b", $time, a, b, y);
  $monitor("Monitor => t=%0t a=%b b=%b y=%b", $time, a, b, y);
end
```

`$display` prints only once at the time it's executed, while `$monitor` continues printing whenever any variable in its list changes.

Example: Waveform Dumping for GTKWave

To generate a VCD (Value Change Dump) file:

```
initial begin
  $dumpfile("test.vcd");    // Name of the waveform file
  $dumpvars(0, testbench); // Dump all signals in testbench
end
```

You can later view the file using tools like GTKWave.

Usage Notes

- System tasks are for simulation only; they are ignored during synthesis.
- They are essential for writing effective testbenches and debugging design behavior.
- Use them early during testbench setup for better traceability.

System tasks like `$display`, `$monitor`, `$dumpfile`, and `$dumpvars` are powerful tools that improve the observability and traceability of Verilog simulations. Although not part of the synthesizable logic, they are indispensable for design verification and debugging.

2.14 Synthesis Considerations

Synthesis is the process of converting Verilog code into a gate-level netlist that can be implemented on hardware such as FPGAs or ASICs. Not all Verilog constructs are synthesizable; therefore, designers must follow certain guidelines to ensure the code can be correctly synthesized.

Key Guidelines for Synthesizable Verilog

- **Avoid Delays and Initial Blocks:** Constructs like `#10`, `$display`, or `initial begin` are simulation-only features and are ignored by synthesis tools. Use clock-driven logic with proper reset mechanisms instead.
- **Use Only Synthesizable Constructs:** Stick to constructs supported by synthesis tools. Avoid non-synthesizable tasks like `$monitor`, file I/O, and infinite loops. Ensure that all conditionals are fully covered to avoid unintended latch inference.
- **Fully Specify Procedural Blocks:** For combinational logic, use `always @(*)` and cover all conditional branches (`if`, `case`, etc.). For sequential logic, use `always @(posedge clk)` and include asynchronous resets when needed.

- **Avoid Latch Inference:** Ensure all outputs are assigned in every branch of a conditional statement. Incomplete assignments can cause unintentional latch generation, which may lead to synthesis mismatches or functional errors.
- **Define Proper Reset Behavior:** Use synchronous or asynchronous resets explicitly to ensure proper initialization in hardware. Tools do not recognize `initial` values for registers in synthesized designs.
- **Use Consistent Clock Domains:** Avoid mixing unrelated clocks without proper synchronization (e.g., synchronizers or FIFO buffers), as it may cause metastability or timing violations in the synthesized design.

Recommended Practices

- Use coding templates recommended by FPGA/ASIC vendors (e.g., Xilinx, Intel, Synopsys).
- Simulate thoroughly before synthesis to ensure correctness.
- Validate synthesis results using timing reports and RTL netlists.

Table 2.5 provides a concise comparison of constructs commonly used in Verilog, helping designers distinguish between those suitable for synthesis and those intended for simulation only.

Table 2.5: Comparison of synthesizable and non-synthesizable constructs

Synthesizable Constructs	**Non-Synthesizable Constructs**
`assign`, `always @(posedge clk)`	`initial` blocks used for simulation only
`if-else`, `case`, static `for` loops	System tasks: `$display`, `$monitor`, `$finish`
`wire`, `reg`, vectors, and arrays	File I/O functions: `$readmemh`, `$fopen`
Synchronous resets, FSMs with complete state coverage	Timing delays: `#10`, `#5`, etc.
Combinational and arithmetic expressions (`+, -, *, &`)	Real/integer types (not mapped to logic gates)
Module instantiation with proper port binding	Timing control in tasks or functions

To ensure successful synthesis, Verilog designs must adhere to guidelines for synthesizable constructs, clocking, resets, and signal assignments. Understanding the limitations

of synthesis tools and writing clean, RTL-compliant code enables efficient and reliable hardware implementation.

2.15 Simulation vs. Synthesis

Simulation and synthesis are two distinct but essential phases in digital hardware design. Simulation focuses on verifying the logical behavior of the design, while synthesis transforms the design into a hardware netlist that can be implemented on physical devices such as FPGAs or ASICs. Table 2.6 presents a detailed comparison between simulation and synthesis in Verilog-based design workflows, highlighting their distinct purposes, supported constructs, and tool-specific considerations essential for both verification and hardware implementation.

Table 2.6: Comparison of simulation and synthesis in Verilog design workflow

Simulation	**Synthesis**
Used to verify functional behavior of a design	Converts RTL code into a gate-level netlist for hardware
Executed by simulators (e.g., ModelSim, VCS, Icarus)	Executed by synthesis tools (e.g., Vivado, Quartus, Synopsys Design Compiler)
Allows all Verilog constructs, including delays (`#`), system tasks (`$display`, `$monitor`), and file operations	Only synthesizable constructs are permitted; delays and file I/O are ignored
Supports testbenches, assertions, and dynamic constructs	Focuses on static logic, deterministic behavior, and complete signal definition
Provides waveform output for debugging (e.g., VCD/FSDB files)	Produces a netlist used for place-and-route or implementation
Can use non-synthesizable blocks like `initial`, `$dumpvars`, or infinite loops for modeling/testing	Requires hardware-realistic logic: proper reset paths, clock domains, and static resource usage
Checks logical and functional correctness	Focuses on area, speed, power, and physical implementability

Simulation allows designers to check for functional correctness, explore design behavior under various scenarios, and debug logic. It supports a wide range of constructs such as delays, file I/O, and non-synthesizable tasks. Synthesis, on the other hand, translates the

RTL Verilog into a gate-level netlist, and only a strict subset of Verilog is allowed during this phase.

While simulation provides flexibility for exploring and validating design behavior, synthesis imposes stricter constraints to ensure that the Verilog code can be physically realized. Writing code that passes both simulation and synthesis is key to producing correct and efficient digital systems.

2.16 Common Mistakes and Debugging Tips

Writing correct and synthesizable Verilog code requires not only syntax accuracy but also good design practices. Many issues arise from subtle errors that can cause simulation mismatches or synthesis failure. Below are some of the most common pitfalls and strategies for debugging them.

Common Mistakes

- **Mixing Blocking and Non-Blocking Assignments:** Using `=` (blocking) and `<=` (non-blocking) incorrectly in sequential or combinational logic can lead to simulation/synthesis mismatches. Use `<=` in clocked processes and `=` in combinational `always @(*)` blocks.

- **Omitting Default Branches:** In `case` statements or conditional logic, missing a default assignment can lead to latch inference or undefined behavior. Always include a `default` case or ensure all conditions are covered.

- **Undeclared or Misspelled Signals:** Referencing signals that are not declared or incorrectly typed can lead to simulation errors or unintended logic. Enable compiler warnings to catch such issues early.

- **Unintended Combinational Loops:** Feedback loops without storage elements (like flip-flops) create combinational loops, which cause instability and prevent synthesis. Always separate combinational and sequential logic properly.

- **Improper Reset Logic:** Not resetting all registers during power-on or simulation may lead to unpredictable behavior. Use synchronous or asynchronous reset blocks with full initialization.

- **Improper Sensitivity Lists:** In older versions of Verilog, an incomplete sensitivity list (e.g., missing signals in `always @(a or b)`) may not reflect the actual logic, leading to simulation mismatches. Use `always @(*)` to automatically include all relevant inputs.

- **Floating or Unconnected Outputs:** Failing to drive outputs can lead to undefined behavior or synthesis warnings. Ensure all outputs are either assigned in all branches or tied to known values.

Debugging Tips

Effective debugging is crucial for identifying logic errors, incomplete designs, and timing issues in Verilog-based systems. The following practices help streamline the process:

- **Use waveform viewers:** Tools like `GTKWave`, `ModelSim`, and `Vivado Simulator` allow you to visually inspect signal transitions, making it easier to trace logic faults and state behavior over time.

- **Add diagnostic output:** Use `$display`, `$monitor`, and `$strobe` to print variable values, simulation progress, and conditional triggers during simulation.

- **Use static analysis tools:** Tools such as `Verilator`, `Yosys`, and `Vivado Lint` can detect undeclared signals, unreachable code, or conflicting assignments early in the development process.

- **Verify state machine coverage:** Ensure that all states in an FSM are reachable and that transitions and outputs are fully defined to avoid unintended latching or undefined behavior.

- **Test reset and clock behavior:** Simulate both reset activation and normal clock cycles to verify initialization and synchronous operation, especially for sequential logic.

- **Modular testing:** Isolate and simulate small modules or components before integrating them into the top-level design. This helps identify bugs at the source.

- **Check for combinational loops:** Ensure all combinational paths have clearly defined inputs and outputs with no unintentional feedback.

Table 2.7 summarizes frequent pitfalls in Verilog design along with practical resolutions to improve code reliability, simulation accuracy, and synthesizability. Anticipating and avoiding common mistakes during the design phase saves significant debugging effort later. Consistent coding style, thorough simulation, and proactive verification are key to developing robust Verilog designs.

Table 2.7: Summary of common Verilog mistakes and recommended resolutions

Common Mistake	**Resolution**
Mixing blocking (`=`) and non-blocking (`<=`) assignments in sequential code	Use `<=` in clocked processes, `=` in combinational logic
Omitting `default` case in `case` statements	Always add a `default` branch to prevent latch inference
Referencing undeclared or misspelled signals	Enable compiler warnings; declare all signals explicitly
Creating unintended combinational loops	Avoid feedback in combinational logic; isolate sequential logic with registers
Missing reset logic for registers	Include synchronous or asynchronous resets in sequential blocks
Incomplete sensitivity lists (e.g., `always @(a or b)`)	Use `always @(*)` to automatically include all relevant signals
Unconnected or floating outputs	Assign outputs in all branches or set default values
Relying on `initial` blocks for hardware reset	Use proper reset conditions with `if (reset)` inside clocked processes

2.17 Advanced Topics Preview

As your Verilog expertise grows, you will encounter more complex design challenges that go beyond basic combinational and sequential logic. The following topics provide a gateway to advanced digital design techniques and industry-relevant practices.

- **Finite State Machines (FSMs):** FSMs are essential for modeling sequential behavior and control flow in digital systems. They are widely used in protocol handlers, controllers, and data path managers. Advanced FSM design includes encoding techniques (e.g., one-hot, gray), hierarchical FSMs, and synchronized transitions.
- **IP Integration:** Intellectual Property (IP) cores are pre-designed and reusable hardware blocks. Most modern FPGA and ASIC projects incorporate vendor or third-party IP (e.g., memory controllers, DSP blocks, processors). Understanding how to instantiate and configure IP cores is critical for efficient system development.
- **Synthesizable Parameterized Modules:** Parameterization enhances module reusability by allowing modules to be configured with different bit-widths or architectural settings. This supports scalable designs such as variable-size ALUs, configurable pipelines, and generic data buses.

- **Interface Protocols (AXI, SPI, UART):** Communication protocols are vital for real-world hardware interfacing. AXI (Advanced eXtensible Interface) is widely used in high-performance SoCs. SPI and UART are common in peripheral communication. Understanding protocol timing, handshaking, and signal mapping is key to building reliable interfaces.

What's Next?

These topics will be explored in later chapters, where you'll learn to design controllers, integrate external components, and write portable, scalable Verilog code suited for real-world hardware platforms. Mastery of these advanced techniques will prepare you for professional FPGA or ASIC development workflows.

2.18 Summary

Verilog HDL is a foundational hardware description language used to design, model, and simulate digital systems. This chapter introduced the core elements of Verilog, including its syntax, data types, procedural constructs, and modeling styles—gate-level, dataflow, and behavioral. Understanding how to apply each modeling style is key to writing clean, synthesizable designs.

Practical design principles such as module instantiation, testbench development, and common debugging strategies are also covered. Key distinctions between simulation and synthesis were explored to emphasize the importance of writing code that is both functionally correct and hardware-realistic.

By mastering these fundamentals, designers can confidently build scalable and reusable modules, implement control logic, interface protocols, and prepare for more advanced topics like state machines, IP integration, and high-level design reuse.

Verilog remains a vital tool in the workflow of digital system development, supporting both ASIC and FPGA targets with a structured and simulation-driven approach.

The laboratory exercise for Chapter 2: *Fundamentals of Verilog HDL* is Lab 3: Basic Logic Gates, which is provided in the Appendix section.

2.19 Exercises

1. **Dataflow Adder:** Write a Verilog module for a 4-bit binary adder using dataflow modeling (use the `assign` keyword).

2. **Behavioral Counter:** Implement a 3-bit up-counter with synchronous reset using behavioral modeling. Include clock and reset signals.

3. **Multiplexer (Gate-Level):** Design a 4-to-1 multiplexer using only gate-level primitives (e.g., `and`, `or`, `not`).

4. **Testbench Practice:** Create a Verilog testbench for a 2-input AND gate. Apply various input combinations and verify output.

5. **Blocking vs Non-blocking:** Demonstrate the difference between blocking (`=`) and non-blocking (`<=`) assignments using a sequential logic example. Describe the effect on simulation behavior.

6. **FSM Sketching:** Outline the states and transitions of a simple traffic light controller (no code required). Identify how you would encode the FSM in Verilog.

7. **Code Debugging (Challenge):** Given a faulty Verilog snippet with a combinational loop, identify the mistake and rewrite it using proper constructs.

Bibliography

[1] C. Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes, 2004.

[2] M. Manohar and J. Bhasker, *Digital System Design Using FPGA: Implementation with Verilog and VHDL*. New York, NY, USA: McGraw-Hill, 2017.

[3] R. Merrick, *Getting Started with FPGAs: Digital Circuit Design, Verilog, and VHDL for Beginners*. San Francisco, CA: No Starch Press, 2023.

[4] S. D. Brown and Z. G. Vranesic, *Fundamentals of Digital Logic with Verilog Design*. New York, NY, USA: McGraw-Hill, 2003.

[5] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[6] ASIC-World.com, "Verilog Tutorial – syntax, testbenches, FSM, memory, PLI," [Online]. Available: https://www.asic-world.com/verilog/veritut.html

[7] ChipVerify.com, "Verilog Tutorial: From basics to hardware circuits," [Online]. Available: https://www.chipverify.com/tutorials/verilog

[8] R. Madhavan, "Verilog Quick Reference," Stanford University, Handout, 2003. [Online]. Available: https://web.stanford.edu/class/ee183/handouts_win2003/VerilogQuickRef.pdf

[9] Wikipedia contributors, "Verilog (Hardware description language)," *Wikipedia*, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Verilog

Chapter 3

Combinational Logic Design

Chapter Objectives

- Understand the principles of combinational logic design.
- Implement combinational circuits in Verilog.
- Use dataflow, behavioral, and structural modeling for simulation and design.

3.1 Introduction to Combinational Logic

Combinational logic refers to digital circuits where the output depends only on the current input values. These circuits do not have memory and do not rely on past input states. When an input changes, the output changes right away after a small delay caused by the logic gates.

Combinational logic is widely used in digital systems to perform various functions. It plays a key role in arithmetic operations through components like adders and subtractors. It is also essential for data routing using multiplexers and demultiplexers, for encoding and decoding digital signals, and for comparing values using comparators. These building blocks are found in CPUs, controllers, and other digital processing units.

Designing a combinational circuit involves several key steps. First, the designer must clearly define the inputs and outputs. Then, a truth table is created to show all possible input combinations and their corresponding outputs. From the truth table, Boolean expressions are derived to describe the logic. These expressions are used to draw the logic circuit or write the design in a hardware description language like Verilog. Finally, the design is tested using simulation to ensure it behaves as expected.

Combinational logic is the starting point for learning digital design. It is used to build many important components of computers and electronics. Once you understand how to

design and simulate these circuits, you'll be ready to move on to sequential logic, where memory and timing come into play.

This chapter will guide you through logic gates, useful modules, Verilog coding, and best practices for designing combinational circuits.

3.2 Basic Logic Gates

Logic gates are the fundamental building blocks of all digital circuits. They perform basic Boolean operations that form the foundation of combinational logic. Each gate takes one or more binary inputs and produces a binary output based on a specific logic function. Figure 3.1 illustrates the standard symbols and configurations of basic logic gates used in digital systems, including AND, OR, NOT, NAND, NOR, XOR, and XNOR gates. Each gate shows how input signals are processed to produce logical outputs, forming the foundation for combinational logic design.

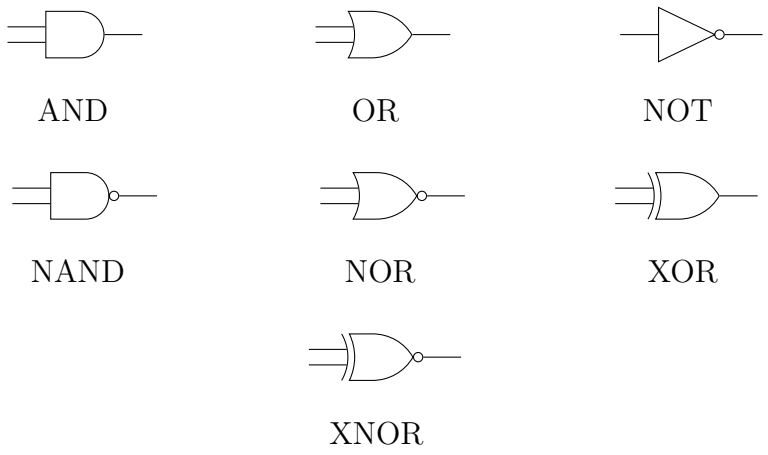


Figure 3.1: Common logic gates

Overview of Logic Gates

- **AND Gate:** Produces a high output (1) only when *all* inputs are high. Commonly used in conditional control, masking operations, and enable logic.
- **OR Gate:** Produces a high output (1) if *any* of the inputs are high. Useful in generating flags or combining multiple conditions.
- **NOT Gate (Inverter):** Produces the complement of the input. Used for inverting signals and in logic expression simplification.
- **NAND Gate:** The complement of an AND gate. Outputs low only when all inputs are high. NAND gates are functionally complete and can be used to build any other logic gate.

- **NOR Gate:** The complement of an OR gate. Outputs high only when all inputs are low. Also functionally complete.
- **XOR Gate (Exclusive OR):** Outputs high when inputs differ. Used in arithmetic logic (e.g., binary addition), parity checkers, and comparators.
- **XNOR Gate (Exclusive NOR):** Outputs high when inputs are equal. Useful in equality testing and digital comparators.

Truth Table for Two-Input Logic Gates

Table 3.1 illustrates the output results of common two-input logic gates—AND, OR, NAND, NOR, XOR, and XNOR—for all possible binary input combinations (A and B). These gates form the fundamental building blocks of digital circuits and are essential for implementing combinational logic functions.

Table 3.1: Truth table for basic logic gates

A	B	AND	OR	NAND	NOR	XOR	XNOR
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1

Verilog Examples

Basic logic gates can be modeled in Verilog using the 'assign' statement, which performs continuous assignment. This is the most common approach for modeling simple combinational logic.

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

```
module or_gate(input a, input b, output y);
  assign y = a | b;
endmodule
```

```
module not_gate(input a, output y);
  assign y = ~a;
endmodule
```

```
module xor_gate(input a, input b, output y);
  assign y = a ^ b;
endmodule
```

Functional Completeness

NAND and NOR gates are known as *functionally complete* gates. This means any Boolean function can be implemented using just NAND or just NOR gates. For example, a NOT gate can be built using a single NAND gate by tying both inputs together.

```
// NOT gate using NAND
assign y = ~(a & a); // Equivalent to y = ~a;
```

Application in Digital Design

Basic gates are often used in combination to build more complex modules such as:

- **Adders and Subtractors:** XOR and AND gates are integral to bitwise addition.
- **Multiplexers:** Use a combination of AND, OR, and NOT gates for selection logic.
- **Encoders/Decoders:** Combinational circuits that use gates for binary translation.
- **ALUs (Arithmetic Logic Units):** The core component of processors, built using gates.

Understanding basic gates is essential before tackling more complex digital design topics. These gates serve as the building blocks for all digital systems—from microcontrollers and FPGAs to full-scale CPUs.

3.3 Boolean Algebra and Simplification

Boolean algebra is a branch of mathematics that deals with binary variables and logical operations. It provides a symbolic framework for designing, analyzing, and optimizing digital circuits. Each variable in Boolean algebra can take on only two values: 0 (false) and 1 (true). The basic operations include logical AND, OR, and NOT, which correspond to physical gates in digital circuits.

Key Boolean Laws and Theorems

To manipulate and simplify Boolean expressions, a set of laws and identities are used. These include:

- **Identity Law:**
  - $A + 0 = A$ (OR with 0 has no effect)
  - $A \cdot 1 = A$ (AND with 1 has no effect)
- **Null Law:**
  - $A + 1 = 1$ (OR with 1 always gives 1)
  - $A \cdot 0 = 0$ (AND with 0 always gives 0)
- **Complement Law:**
  - $A + A' = 1$ (OR of a variable and its complement is always 1)
  - $A \cdot A' = 0$ (AND of a variable and its complement is always 0)
- **Idempotent Law:**
  - $A + A = A$
  - $A \cdot A = A$
- **Involution Law:**
  - $(A')' = A$
- **Associative Law:**
  - $A + (B + C) = (A + B) + C$
  - $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- **Commutative Law:**
  - $A + B = B + A$
  - $A \cdot B = B \cdot A$
- **Distributive Law:**
  - $A \cdot (B + C) = A \cdot B + A \cdot C$
  - $A + (B \cdot C) = (A + B) \cdot (A + C)$
- **De Morgan's Laws:**
  - $\overline{A + B} = \overline{A} \cdot \overline{B}$
  - $\overline{A \cdot B} = \overline{A} + \overline{B}$

Advantages of Boolean Simplification

Simplifying Boolean expressions helps in several ways:

- **Reduces Logic Gates:** Simplified expressions lead to fewer logic gates in hardware, reducing chip area and power.
- **Increases Speed:** Less gate delay improves timing and circuit speed.
- **Improves Readability:** Easier to understand and debug compact logic expressions.
- **Minimizes Cost:** Fewer components lead to reduced manufacturing cost and power consumption.

Simplification Techniques

Several methods are used to simplify Boolean expressions:

- **Algebraic Simplification:** Applying Boolean laws manually to reduce the number of terms and operations.
- **Truth Tables:** Comparing output values of original and simplified expressions to ensure correctness.
- **Karnaugh Maps (K-Maps):** A visual tool for minimizing expressions of up to 4–6 variables by grouping adjacent 1s.
- **Quine–McCluskey Algorithm:** A tabular method for simplification, especially suited for implementation in software tools.

Example

Consider the expression:

$$Y = AB + A\overline{B} \tag{3.1}$$

This can be simplified using the Distributive Law:

$$Y = A(B + \overline{B}) = A(1) = A \tag{3.2}$$

Boolean algebra is not only a theoretical tool but a practical necessity in digital design. By mastering its laws and simplification techniques, engineers can optimize digital circuits for efficiency, reliability, and performance. Understanding Boolean algebra is fundamental to any logic synthesis or digital hardware course.

3.4 Common Combinational Circuits

Combinational circuits are digital logic systems where the output depends only on the current inputs, without any memory or storage elements. This section explores commonly used combinational building blocks that form the basis of larger digital designs.

3.4.1 Multiplexers

A multiplexer (MUX) is a data selector that routes one of several input signals to a single output based on the value of the select lines. It is widely used in datapaths, buses, and control systems.

Example: 2-to-1 Multiplexer

```
module mux2to1(input a, input b, input sel, output y);
  assign y = sel ? b : a;
endmodule
```

This circuit outputs 'a' when 'sel=0' and 'b' when 'sel=1'. Larger multiplexers (e.g., 4-to-1, 8-to-1) can be built using nested conditionals or by chaining smaller MUX units.

3.4.2 Decoders

A decoder activates exactly one output based on a binary input. They are essential in memory addressing, control signal generation, and instruction decoding.

Example: 2-to-4 Decoder

```
module decoder2to4(input [1:0] in, output [3:0] out);
  assign out = 1 << in;
endmodule
```

When "in=2'b10", the output is "out=4'b0100", meaning the third output line is active. For n input bits, a decoder produces 2^n output lines.

3.4.3 Encoders and Priority Encoders

An encoder performs the inverse function of a decoder: it takes 2^n inputs and encodes the active one into an n-bit binary code. If multiple inputs are active, a **priority encoder** outputs the highest-priority active input.

Use Cases:

- Keyboard scanners
- Interrupt request handling
- Data compression in control logic

3.4.4 Comparators

Comparators evaluate the magnitude relationship between two binary numbers and generate outputs such as:

- A > B
- A < B
- A = B

They are used in arithmetic units, control structures, and sorting mechanisms.

Example: 4-bit Equality Comparator (pseudocode)

```
assign equal = (a == b);
```

More complex comparators also produce “greater than” and “less than” signals.

3.4.5 Adders and Subtractors

Arithmetic operations are foundational in digital design. The most basic elements include:

Half Adder: Computes sum and carry for two input bits.

- sum = $A \oplus B$
- carry = $A \wedge B$

Full Adder: Adds two input bits and a carry-in.

```
module full_adder(input a, b, cin, output sum, cout);
  assign sum = a ^ b ^ cin;
  assign cout = (a & b) | (b & cin) | (a & cin);
endmodule
```

Full adders can be chained to create multi-bit ripple carry adders. Subtractors can be built using 2’s complement logic or dedicated subtraction circuits.

These fundamental combinational circuits — multiplexers, decoders, encoders, comparators, and adders — are building blocks for larger systems like ALUs, control units, and memory controllers. Mastery of these elements is crucial for efficient and modular digital design.

3.5 Design Examples

This section presents several practical combinational logic design examples in Verilog. These designs are commonly used in digital systems such as ALUs, control units, and display drivers. Each example is written in synthesizable Verilog and demonstrates a different method of implementing logic functions.

3.5.1 4-bit Ripple Carry Adder

The ripple carry adder is a basic arithmetic circuit that performs binary addition. It consists of four full adders connected in series, where each adder passes its carry-out to the next stage's carry-in.

Full Adder Module:

```
module full_adder (
  input a, b, cin,
  output sum, cout
);
  assign sum = a ^ b ^ cin;
  assign cout = (a & b) | (b & cin) | (a & cin);
endmodule
```

4-bit Ripple Carry Adder:

```
module ripple_carry_adder_4bit (
  input [3:0] a, b,
  input cin,
  output [3:0] sum,
  output cout
);
  wire c1, c2, c3;

  full_adder fa0 (a[0], b[0], cin,  sum[0], c1);
  full_adder fa1 (a[1], b[1], c1,   sum[1], c2);
  full_adder fa2 (a[2], b[2], c2,   sum[2], c3);
  full_adder fa3 (a[3], b[3], c3,   sum[3], cout);
endmodule
```

This adder demonstrates hierarchical design and bitwise operations.

3.5.2 4x1 Multiplexer Using Case Statement

A 4-to-1 multiplexer selects one of four inputs to pass to the output based on a 2-bit selector input. It is a key component in datapaths and control units.

```
module mux4to1 (
  input [1:0] sel,
  input [3:0] in,
  output reg y
);
  always @(*) begin
```

```
    case (sel)
      2'b00: y = in[0];
      2'b01: y = in[1];
      2'b10: y = in[2];
      2'b11: y = in[3];
    endcase
  end
endmodule
```

This case-based structure is more scalable and easier to extend compared to nested ternary operators.

3.5.3 BCD to 7-Segment Display Decoder

This circuit maps a 4-bit BCD (Binary-Coded Decimal) input to a 7-segment display format. It is frequently used in digital clocks, calculators, and counters to represent numeric values visually.

```
module bcd_to_7seg (
  input [3:0] bcd,
  output reg [6:0] seg
);
  always @(*) begin
    case (bcd)
      4'd0: seg = 7'b1000000; // "0"
      4'd1: seg = 7'b1111001; // "1"
      4'd2: seg = 7'b0100100; // "2"
      4'd3: seg = 7'b0110000; // "3"
      4'd4: seg = 7'b0011001; // "4"
      4'd5: seg = 7'b0010010; // "5"
      4'd6: seg = 7'b0000010; // "6"
      4'd7: seg = 7'b1111000; // "7"
      4'd8: seg = 7'b0000000; // "8"
      4'd9: seg = 7'b0010000; // "9"
      default: seg = 7'b1111111; // blank
    endcase
  end
endmodule
```

Each segment bit controls an LED; a '0' lights the segment assuming active-low logic. For FPGA applications, these outputs connect directly to the segment pins of a 7-segment display.

These examples demonstrate how Verilog can be used to implement fundamental combinational blocks that are reusable and modular. Understanding their structure lays the groundwork for more complex systems such as ALUs, datapaths, and memory controllers.

3.6 Modeling Techniques

Verilog HDL supports three primary modeling techniques to describe digital systems: dataflow modeling, behavioral modeling, and structural modeling. Each technique offers a unique level of abstraction and is tailored to different stages of design and verification. Understanding the appropriate context and application of each model is essential for efficient digital circuit design and synthesis.

3.6.1 Dataflow Modeling

Dataflow modeling emphasizes the description of how data moves through the system, using logical and arithmetic expressions. It reflects the combinational logic of the circuit using continuous assignment statements with the keyword `assign`. This style is particularly effective for describing circuits with straightforward functional relationships between inputs and outputs, such as arithmetic units and encoders.

Dataflow modeling allows the designer to focus on Boolean equations and the relationships among signals rather than the underlying gate-level or register-level structure.

Key Features:

- Utilizes operators (e.g., `+, -, *, &, |, ^, «, »`).
- Suitable for describing arithmetic and logical functions.
- Synthesizes directly to combinational logic.
- Does not require explicit clocking or procedural blocks.

Example: 4-bit Adder Using Dataflow

```
module adder (
    input [3:0] A,
    input [3:0] B,
    output [4:0] SUM
);
    assign SUM = A + B; // Continuous assignment
endmodule
```

Use Cases:

- Arithmetic logic units (ALUs)

- Multiplexers and demultiplexers
- Comparators
- Bitwise and logical operations

3.6.2 Behavioral Modeling

Behavioral modeling describes the operation of a circuit based on its functionality and behavior over time, often using control flow constructs similar to those in software programming languages. It is written inside `always` or `initial` blocks and enables the use of constructs like `if-else`, `case`, loops (`for`, `while`, `repeat`), and blocking/non-blocking assignments.

This modeling technique is ideal for defining control logic, finite state machines (FSMs), sequential circuits, and algorithms. It allows the designer to abstract away hardware implementation details and focus on the logical behavior of the system.

Key Features:

- Supports conditional statements and complex logic flow.
- Enables procedural assignment using `always` blocks.
- Supports sequential and clocked logic.
- Useful for simulation, testbenches, and high-level design.

Example: 4-bit Comparator Using Behavioral Modeling

```
module comparator (
    input [3:0] A,
    input [3:0] B,
    output reg A_gt_B
);
    always @ (A or B) begin
        if (A > B)
            A_gt_B = 1;
        else
            A_gt_B = 0;
    end
endmodule
```

Use Cases:

- Finite State Machines (FSMs)
- Control units

- UART controllers
- Algorithmic state machines (ASMs)

3.6.3 Structural Modeling

Structural modeling is the most detailed and low-level form of hardware description in Verilog. It mirrors the physical construction of a circuit by explicitly defining how components are interconnected. In this style, circuits are described hierarchically by instantiating modules (subcircuits) and connecting them using wires and ports.

This technique is essential when designing at the gate-level, integrating IP cores, or building systems from smaller submodules. It provides precise control over design hierarchy and interconnections.

Key Features:

- Uses module instantiation to connect components.
- Mimics schematic-level design.
- Useful for system-level integration and hierarchical design.
- Promotes design reuse and modularity.

Example: Full Adder Using Structural Modeling

```
module half_adder (
    input A, B,
    output SUM, CARRY
);
    xor x1(SUM, A, B);
    and a1(CARRY, A, B);
endmodule

module full_adder (
    input A, B, Cin,
    output SUM, Cout
);
    wire s1, c1, c2;

    half_adder ha1(A, B, s1, c1);
    half_adder ha2(s1, Cin, SUM, c2);
    or o1(Cout, c1, c2);
endmodule
```

Use Cases:

- Hierarchical design of datapath components
- Top-level integration of submodules
- Gate-level netlists
- Verification of synthesis and placement results

Comparison of Modeling Techniques

Table 3.2 compares the three primary Verilog modeling techniques—dataflow, behavioral, and structural—in terms of abstraction level, coding style, suitability, clock usage, synthesizability, and complexity.

Table 3.2: Comparison of Verilog modeling techniques

Feature	**Dataflow**	**Behavioral**	**Structural**
Abstraction	Medium	High	Low
Code Style	Assignments	Procedural	Instantiation
Suited For	Combinational	FSMs, Control	Hierarchy
Clock Use	No	Yes	Depends
Synthesizable	Yes	Yes	Yes
Complexity	Moderate	High-level	Low-level

In practice, designers often combine all three modeling techniques within a single project. For instance, arithmetic units may be written in dataflow style, control logic in behavioral style, and top-level integration in structural style. The ability to choose and mix these modeling paradigms provides the designer with powerful flexibility to handle both design complexity and synthesis requirements efficiently.

3.7 Design Optimization and Synthesis Tips

Efficient digital design in Verilog HDL not only relies on functional correctness but also on optimization for synthesis. Writing hardware description code that synthesizes into minimal, high-performance hardware is a critical skill for engineers working with FPGAs or ASICs. This section provides key practical tips for design optimization and synthesis-aware coding.

- **Avoid Unnecessary Logic Duplication:** Redundant logic can increase gate count, power consumption, and propagation delay. When writing conditional statements or arithmetic operations, avoid repeating the same computation in multiple branches or expressions. Use intermediate wires or registers to store reused values.

```
// Inefficient
assign result = (sel) ? (A + B) : (A + B + 1);

// Optimized
wire [3:0] sum = A + B;
assign result = (sel) ? sum : (sum + 1);
```

- **Combine Similar Functions into Shared Logic:** If multiple operations or modules share similar logic (e.g., common sub-expressions or modules with similar structures), consolidate them into a single, parameterized block. This not only saves area but improves maintainability and clarity of the design.

```
// Shared mux logic for multiple data channels
function [7:0] mux2;
    input [7:0] in1, in2;
    input sel;
    begin
        mux2 = sel ? in2 : in1;
    end
endfunction

assign outA = mux2(dataA1, dataA2, sel);
assign outB = mux2(dataB1, dataB2, sel);
```

- **Use Generate Loops for Repeated Instances:** When instantiating repeated logic blocks like adders, counters, or bit-sliced arithmetic units, use 'generate' statements with 'for' loops instead of writing each instance manually. This improves code scalability, modularity, and reduces human error.

```
genvar i;
generate
    for (i = 0; i < 8; i = i + 1) begin : gen_and
        and u_and (Y[i], A[i], B[i]);
    end
endgenerate
```

- **Use Clock Gating Carefully:** While clock gating can reduce power consumption, improper use can lead to glitches and synthesis issues. Use built-in clock gating cells if supported by the synthesis tool, or apply gating at enable logic level rather than manipulating the clock signal directly.

- **Use Non-blocking Assignments in Sequential Logic:** Always use non-blocking assignments ('<=') within 'always @(posedge clk' blocks to avoid race conditions and ensure predictable simulation and synthesis behavior.

```
// Correct usage
always @(posedge clk) begin
    q <= d;
end
```

- **Use Reset Logic Wisely:** Reset signals should be used only when necessary. Overusing resets on large registers or datapaths increases area and slows down timing. For registers that do not require initialization, allow default power-on states and rely on the design flow for proper initialization.

- **Constrain Critical Paths and Timing:** Use synthesis and implementation tools to identify and optimize critical timing paths. Write timing constraints ('.sdc' or '.xdc' files) to ensure the synthesized design meets clock frequency requirements.

- **Minimize Use of Complex Control Logic in One Block:** Keep control logic modular and readable. Avoid deeply nested 'if-else' or 'case' statements in a single 'always' block. This facilitates better synthesis mapping and verification.

- **Leverage IP Cores When Possible:** Most FPGA toolchains provide optimized IP blocks (e.g., FIFOs, multipliers, DDR controllers). Using vendor-provided IP blocks saves design time and ensures efficient implementation compared to hand-coded RTL.

- **Validate With Synthesis Reports:** After synthesis, examine area reports, timing summaries, and inferred logic reports. These give insight into resource utilization (LUTs, FFs, BRAMs), maximum clock frequency, and synthesized logic behavior.

Effective optimization in Verilog design requires a balance between abstraction and hardware efficiency. Awareness of how HDL constructs map to physical resources allows designers to write cleaner, faster, and more resource-conscious RTL. These practices are crucial not only for achieving performance goals but also for enabling portability and scalability of digital systems.

3.8 Simulation and Testbenches

Simulation is an essential part of digital design in Verilog HDL. Before synthesizing a design for hardware implementation, simulation verifies the logical correctness of the

code. This process is performed using testbenches, which are simulation-only modules that provide stimulus to the DUT and observe its outputs.

A testbench does not represent synthesizable hardware—it is meant exclusively for verification. It uses initial blocks, test vectors, time delays, and monitoring statements to validate the behavior of a module under different conditions.

Structure of a Testbench:

- Declares signals (`reg`, `wire`) that connect to the DUT.
- Instantiates the DUT with port mapping.
- Contains one or more `initial` blocks to apply stimulus.
- May include `$monitor`, `$display`, or `$dumpfile/$dumpvars` for waveform generation and output tracking.

Example: 4-bit Adder Testbench

The following example illustrates a simple testbench for a 4-bit ripple-carry adder module named `adder4bit`. It applies input values to the DUT and checks the corresponding outputs at different simulation times.

```
module test_adder;
  reg [3:0] a, b;
  reg cin;
  wire [3:0] sum;
  wire cout;

  // Instantiate the Design Under Test (DUT)
  adder4bit uut (.a(a), .b(b), .cin(cin), .sum(sum), .cout(cout))
     ;

  initial begin
    // Apply first test vector
    a = 4'b0001; b = 4'b0010; cin = 0;
    #10; // Wait for 10 time units

    // Apply second test vector
    a = 4'b1111; b = 4'b0001; cin = 0;
    #10;

    // End simulation
    $finish;
  end
```

```
endmodule
```

Explanation:

- The `reg` types are used to drive the inputs `a`, `b`, and `cin`.
- The `wire` types receive the outputs `sum` and `cout` from the DUT.
- The `uut` instance represents the 4-bit adder being tested.
- The `initial` block applies two test cases, with a 10-time unit delay between them.
- `$finish` terminates the simulation after the tests.

Enhancing the Testbench: To make testbenches more powerful and informative, the following additions are often used:

- `$display` to print output values at each time step.
- `$monitor` to automatically print values whenever they change.
- `$dumpfile` and `$dumpvars` to generate VCD (Value Change Dump) waveform files for viewing in GTKWave or other tools.
- `for` or `repeat` loops for applying multiple test vectors.

Example With Monitoring and Waveform Output:

```
initial begin
  $display("Time\t a\t b\t cin\t sum\t cout");
  $monitor("%g\t %b\t %b\t %b\t %b\t %b", $time, a, b, cin, sum,
     cout);
  $dumpfile("adder_test.vcd");
  $dumpvars(0, test_adder);
end
```

Testbenches play a critical role in pre-silicon validation and debugging. A well-constructed testbench ensures that the HDL design behaves as intended under a variety of scenarios and edge cases. Simulation also helps identify timing mismatches, logic errors, and incorrect assignments before committing to synthesis or FPGA implementation.

3.9 Common Mistakes and Debugging

Even experienced Verilog designers can encounter subtle bugs and synthesis issues due to common pitfalls in HDL coding. These mistakes often lead to incorrect simulation

results, inefficient hardware, or unexpected behavior after synthesis. This section outlines frequent errors and provides strategies for identifying and correcting them during the design process.

- **Forgetting Default Cases in Case Statements:**
  When using `case` or `casez` constructs in behavioral modeling, omitting a `default` clause can result in unintended latches during synthesis. This is because synthesis tools may infer that the output should retain its previous value when no matching case is found—leading to inferred memory elements (latches) instead of combinational logic.

```
// Problematic
always @(*) begin
  case (sel)
    2'b00: out = a;
    2'b01: out = b;
    // missing default
  endcase
end
```

  Fix: Always include a `default` case to cover all conditions and avoid latches.

```
default: out = 4'b0000;
```

- **Incorrect Bit-Width Assignments:**
  Mismatched bit-widths can cause truncation, sign extension, or simulation/synthesis mismatches. This commonly occurs when assigning wider values to narrower variables or mixing signed and unsigned values without care.

```
reg [3:0] a;
reg [7:0] b;
assign b = a; // OK (zero-extension)
assign a = b; // Truncated: only lower 4 bits used
```

  Fix: Explicitly match bit widths or use appropriate casting when assigning values.

```
assign a = b[3:0]; // safe truncation
assign b = {4'b0000, a}; // zero-extended to 8 bits
```

- **Mixing Blocking and Non-blocking Assignments in Combinational Circuits:**
  Using both blocking (`=`) and non-blocking (`<=`) assignments within the same `always`

block or across interacting `always` blocks can lead to unpredictable simulation results and synthesis mismatches. This mistake is particularly common in FSMs or arithmetic pipelines.

```
always @(posedge clk) begin
  a = b;        // blocking
  c <= a + 1;   // non-blocking (reads stale a)
end
```

Fix: Use non-blocking assignments consistently in sequential logic (`always @(posedge clk)`) and reserve blocking assignments for combinational logic (`always @(*)`).

```
always @(posedge clk) begin
  a <= b;
  c <= a + 1;
end
```

- **Not Resetting Registers When Required:**
  Registers that require known initial values must be reset explicitly using synchronous or asynchronous reset logic. Failing to reset can result in undefined behavior in hardware, especially in FPGAs where power-on values are not guaranteed.

- **Improper Sensitivity Lists:**
  In behavioral modeling, incomplete sensitivity lists (for pre-2001 Verilog) can cause mismatches between simulation and synthesis. Although newer tools support `always @(*)`, older styles may need explicit signals listed.

- **Latch Inference from Incomplete Assignments:**
  When not all code paths assign a value to an output inside an `always` block, the synthesis tool may infer a latch to preserve the value. This is often unintentional and leads to unwanted sequential elements.

```
always @(*) begin
  if (enable)
    out = data;
  // missing else branch: causes latch
end
```

Fix: Ensure every path assigns a value, or include an `else` clause to define the default.

- **Not Using Waveform Debugging Tools:**
  Many bugs are easier to identify through waveform analysis rather than raw console

output. Not using tools like GTKWave, ModelSim's waveform viewer, or Vivado Simulator can hinder debugging of timing errors, glitches, or value transitions.

- **Failing to Isolate Testbench From DUT:**
  Allowing the testbench logic to unintentionally interfere with DUT internals (e.g., driving DUT signals outside of module scope) can skew simulation results and mask design flaws.

- **Assuming Synthesis Matches Simulation Exactly:**
  Constructs like delays (`#10`), timing control, and certain system tasks (e.g., `$display`, `$stop`) are ignored during synthesis. Simulation-only code should be confined to testbenches or clearly separated using 'ifdef' synthesis guards.

- **Not Reviewing Synthesis Warnings:**
  Ignoring synthesis tool warnings can lead to critical issues such as unconnected outputs, inferred latches, or logic pruning. Always examine and resolve these warnings for reliable hardware implementation.

Careful coding practices, consistent simulation-synthesis discipline, and proper use of debugging tools are essential for producing robust Verilog designs. Understanding these common mistakes not only helps in early-stage debugging but also improves long-term code quality and maintainability.

3.10 Summary

Combinational logic is a fundamental component of digital systems, where outputs are determined solely by current inputs. This chapter explained how to describe and implement combinational circuits using Verilog HDL.

The discussion began with basic logic gates—AND, OR, NOT, NAND, NOR, XOR, and XNOR—and demonstrated their implementation in Verilog. Common combinational modules such as multiplexers, adders, decoders, and comparators were also explored as essential building blocks for more complex digital systems.

Three key modeling styles were introduced:

- **Dataflow** – logic described using continuous assignments.
- **Behavioral** – logic defined using high-level constructs such as `if` and `case`.
- **Structural** – systems constructed by interconnecting smaller modules.

Important design practices were emphasized, including strategies for avoiding redundant logic and ensuring synthesizable Verilog code. Simulation techniques using testbenches were covered, along with methods to identify common issues such as missing default cases or incorrect bit widths.

This chapter equips readers with the skills needed to create, test, and optimize combinational logic circuits. The next chapter transitions to sequential logic, introducing memory elements and clock signals to enable time-dependent behavior in digital designs.

The laboratory exercises for Chapter 3: *Combinational Logic Design* include Lab 4: Adder Design, Lab 5: MUX and DMUX Designs, Lab 6: Encoder and Decoder Design, Lab 7: Rotator and Shifter Design, and Lab 8: Simple ALU Design, all of which are provided in the Appendix section.

3.11 Exercises

1. **4-bit Comparator:** Design a Verilog module that compares two 4-bit binary numbers `A` and `B`. The output should indicate whether `A > B`, `A == B`, or `A < B`.

2. **2-to-4 Decoder with Enable:** Write a Verilog module for a 2-to-4 line decoder with an active-high enable input. Use behavioral modeling for clarity.

3. **8-bit Adder/Subtractor:** Implement an 8-bit arithmetic unit that performs addition when `ctrl = 0` and subtraction when `ctrl = 1`. Use dataflow modeling.

4. **Multiplexer Testbench:** Create a testbench to simulate a 4-to-1 multiplexer. Apply different combinations of inputs and select lines. Use `$display` to show output.

5. **Behavioral Full-Adder:** Rewrite a full-adder module using behavioral modeling with `case` statements or `if-else` constructs instead of Boolean expressions.

6. **Logic Simplification:** Given a truth table for a 3-input logic circuit, derive the simplified Boolean expression using Karnaugh maps, then implement the logic in Verilog.

7. **Gate Count Estimation (Challenge):** Estimate the gate count of a 4-bit ripple-carry adder implemented using full-adders. Discuss the delay and area trade-offs in your design.

Bibliography

[1] C. Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes, 2004.

[2] M. Manohar and J. Bhasker, *Digital System Design Using FPGA: Implementation with Verilog and VHDL*. New York, NY, USA: McGraw-Hill, 2017.

[3] R. Merrick, *Getting Started with FPGAs: Digital Circuit Design, Verilog, and VHDL for Beginners*. San Francisco, CA: No Starch Press, 2023.

[4] S. Brown and Z. G. Vranesic, *Fundamentals of Digital Logic with Verilog Design*, 3rd ed. New York, NY, USA: McGraw-Hill, 2014.

[5] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[6] D. Ramanathan, "Part 14: Combinational Logic in Verilog: Explained with 5 Examples," *Maker.io – DigiKey*, May 5, 2025. [Online]. Available: https://www.digikey.com/en/maker/tutorials/2025/part-14-combinational-logic-in-verilog-explained-with-5-examples

[7] ChipVerify, "Combinational Logic," [Online]. Available: https://www.chipverify.com/digital-fundamentals/combinational-logic

[8] S. Arar, "Describing Combinational Circuits in Verilog," *All About Circuits*, Jan. 6, 2019. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/describing-combinational-circuits-in-verilog/

Chapter 4

Sequential Logic Design

Chapter Objectives

- Understand the principles of sequential logic and memory elements.
- Design sequential circuits in Verilog using flip-flops, counters, and shift registers.
- Simulate and verify sequential designs with proper clocking and reset strategies.

4.1 Introduction to Sequential Logic

Sequential logic is an important part of digital design that allows circuits to remember past events and respond over time. Unlike **combinational logic**, which depends only on current inputs, sequential logic produces outputs based on both current inputs and stored past values.

The key building blocks of sequential logic are **latches** and **flip-flops**, which hold binary data and maintain it from one clock cycle to the next. These elements rely on a **clock signal**—a regular timing pulse—to control when data is stored and when the system changes state. This makes the circuit behave in a reliable and organized way.

Sequential circuits can be classified into two categories:

Types of Sequential Circuits

- **Synchronous:** All memory elements change state together, controlled by a clock signal. This is the most common and reliable type.
- **Asynchronous:** Changes happen based on input changes without a clock. These are faster but harder to design and may face timing issues.

Common Sequential Components

- **Counters and Timers:** For counting and timing operations.
- **Shift Registers:** Convert data between serial and parallel forms.
- **Finite State Machines (FSMs):** Used for control flow and decision-making.
- **Datapaths and Control Units:** Core parts of processors handling computation and instruction flow.
- **Communication Interfaces:** Support protocols like UART, SPI, and I2C for data transfer.

Clock signals control when sequential circuits update. Using edge-triggered flip-flops ensures stable state changes.

Reset signals set the system to a known state at startup or during errors.

Sequential logic adds time-based behavior, enabling memory, control, and complex processing in digital systems.

4.2 Memory Elements: Latches and Flip-Flops

Sequential digital systems rely on memory elements to store binary information across clock cycles. These storage elements form the basis for registers, counters, finite state machines, and other components that require state retention. Memory elements are broadly categorized into two types: **latches** and **flip-flops**.

4.2.1 Latches

Latches are *level-sensitive* storage elements, meaning they are transparent when the control signal (typically called `enable` or `EN`) is active. They respond continuously to the input as long as the enable signal is asserted. When the enable signal is deasserted, the latch holds its value, effectively storing data until the next enable pulse.

SR Latch (Set-Reset Latch)

The SR (Set-Reset) latch is the most fundamental latch design, typically implemented using two cross-coupled NOR or NAND gates. It has two inputs: Set (S) and Reset (R), and two complementary outputs: Q and $\overline{Q}$.

- When S=1 and R=0, the latch is *set* (Q=1).
- When S=0 and R=1, the latch is *reset* (Q=0).

- When both S and R are 0, the latch *retains* its current state.
- When both S and R are 1 (in NOR-based implementation), the state is *invalid* or *undefined*.

Because of this invalid condition, the SR latch is rarely used directly in modern design but remains important for understanding the foundation of more robust memory elements.

D Latch (Data Latch)

The D latch simplifies the SR latch by eliminating the possibility of an invalid state. It has one data input (D) and an enable signal (EN). When EN is high, the latch is transparent, and the output Q follows the input D. When EN is low, the output holds its last value regardless of changes at the D input.

```
always @(*) begin
  if (en)
    q = d;
end
```

The D latch is commonly used in gated memory circuits and level-sensitive systems where data needs to be captured during specific time windows.

4.2.2 Flip-Flops

Flip-flops are *edge-triggered* memory elements. Unlike latches, which respond to level-sensitive control signals, flip-flops update their outputs only at the rising or falling edge of a clock signal. This makes them ideal for synchronous design, where operations must occur in lockstep with a global clock.

The most commonly used flip-flop is the **D Flip-Flop**, which stores the value at the data input (D) on the active edge of the clock (typically the rising edge).

D Flip-Flop Example

```
always @(posedge clk)
  q <= d;
```

This behavior ensures data is sampled and stored at well-defined moments, which simplifies timing analysis and improves circuit reliability.

Other Types of Flip-Flops

In addition to the D flip-flop, there are other useful variants:

- **T Flip-Flop (Toggle Flip-Flop):** Toggles the output state on each clock edge when T=1. It is commonly used in binary counters.
- **JK Flip-Flop:** A general-purpose flip-flop that combines the behavior of SR and T flip-flops. When both inputs J and K are high, the output toggles.

Comparison Between Latches and Flip-Flops

Table 4.1 highlights the differences between latches and flip-flops in terms of triggering mechanism, control signals, timing precision, typical usage, and structural complexity.

Table 4.1: Comparison of latches and flip-flops

Feature	**Latch**	**Flip-Flop**
Trigger Type	Level-sensitive	Edge-triggered
Control Signal	Enable (EN)	Clock (clk)
Timing Precision	Less precise (sensitive to input glitches during EN)	Precise control on clock edge
Usage	Gated memory, asynchronous systems	Synchronous digital systems
Complexity	Simple structure	Slightly more complex

Latches and flip-flops together enable the construction of sequential circuits that can store, process, and control digital data in response to timing signals and input events.

4.3 Registers and Register Banks

Sequential digital systems often require storing and manipulating multiple bits of data across clock cycles. This need is fulfilled by using registers—groups of flip-flops—and register banks, which are arrays of such registers. These storage structures play a crucial role in implementing datapaths, buffering intermediate values, and storing processor state.

4.3.1 Multi-Bit Registers

A **multi-bit register** consists of a collection of flip-flops, each holding one bit of data, that collectively store a binary word. Registers can range from 2-bit counters to 32-bit or 64-bit processor registers. In Verilog, a basic 8-bit register is declared using:

```
1 reg [7:0] data;
```

This declaration defines a single 8-bit storage element. The bits are indexed from `[7]` (most significant bit) to `[0]` (least significant bit).

To update a register on the rising edge of a clock with asynchronous reset, an `always` block is used:

```
always @(posedge clk or posedge reset) begin
  if (reset)
    data <= 8'b0;
  else
    data <= next_data;
end
```

Such registers are commonly used for:

- Temporary data storage between operations
- Buffering inputs or outputs
- Holding control flags or intermediate results
- Forming part of larger arithmetic or logical units

4.3.2 Register Bank Implementation

A **register bank** or **register file** is a collection of registers indexed via an address or control signal. It enables simultaneous or sequential read/write access to a group of registers and is widely used in CPU architectures, ALU interfaces, and I/O buffering schemes.

In Verilog, a 16-entry register file with 8-bit words can be declared as:

```
reg [7:0] reg_file [0:15];
```

This creates a memory-like structure with 16 addressable 8-bit locations. Accessing specific registers is straightforward:

```
reg_file[3] = 8'hA5;           // Write the value 0xA5 to
   register 3
output_data = reg_file[7];     // Read value from register 7
```

For clocked, synchronous operation, the following construct is used:

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

assign read_data = reg_file[read_addr];
```

This structure enables modular and scalable data storage. In microprocessor designs, register banks are typically dual-ported to allow simultaneous read and write access.

Applications of Register Banks

Register banks are critical components in digital systems and have several use cases:

- **CPU Register Files:** Store general-purpose registers used during program execution.
- **Instruction Decoding:** Provide operands for ALU operations.
- **Pipeline Buffers:** Hold intermediate results between pipeline stages.
- **I/O Buffers:** Temporarily store data in peripheral interfaces.

Design Considerations

- Use proper indexing and ensure address decoding is correctly implemented.
- For synthesis, ensure register files are written in a way that is compatible with RAM inference (in FPGA tools).
- Apply reset conditions only when needed, as large register banks may require significant hardware for global resets.

Registers and register banks provide essential memory elements for storing and retrieving data in digital systems. Understanding how to implement them efficiently is fundamental to designing complex systems like processors, communication interfaces, and control units.

4.4 Counters

Counters are fundamental sequential logic circuits that change state in a predictable sequence in response to clock pulses. They are widely used in digital systems to count clock cycles, events, or external signals, as well as to control sequencing, timing, and data flow.

A counter typically consists of flip-flops connected in such a way that their outputs represent the current count value. The width of the counter (`n` bits) determines the range, i.e., it can count from 0 to $2^n - 1$.

4.4.1 Up Counter

An up counter increases its count by one on every rising clock edge. It rolls over to zero after reaching the maximum count value based on its bit width. This type of counter is ideal for use cases like pulse counting, iteration control, and memory address generation.

```
reg [3:0] count;

always @(posedge clk or posedge reset)
  if (reset)
    count <= 4'b0000;
  else
    count <= count + 1;
```

Design Note: Always include a synchronous or asynchronous reset to ensure deterministic behavior upon startup.

4.4.2 Down Counter

A down counter performs the reverse of an up counter. It decrements its value on each clock edge and wraps around when it reaches zero. These counters are commonly used in applications like countdown timers and loop iteration termination.

```
reg [3:0] count;

always @(posedge clk or posedge reset)
  if (reset)
    count <= 4'b1111;
  else
    count <= count - 1;
```

Design Note: In most hardware applications, underflow behavior should be explicitly handled or wrapped safely to avoid glitches.

4.4.3 Up-Down Counter

Up-down counters offer bi-directional counting functionality. A control signal (e.g., `dir`) determines whether the counter increments or decrements. These are useful in applications requiring flexible navigation, such as elevator logic or stack pointer updates.

```
reg [3:0] count;
input dir; // dir = 1: count up, dir = 0: count down

always @(posedge clk or posedge reset)
```

```
  if (reset)
    count <= 4'b0000;
  else if (dir)
    count <= count + 1;
  else
    count <= count - 1;
```

Design Tip: Ensure glitch-free transition of the direction signal, especially when interfacing with asynchronous modules.

4.4.4 Mod-N Counter

A Mod-N counter counts from 0 to N-1 and then resets to zero. It is frequently used in digital systems for dividing clock frequencies (e.g., Mod-10 for a decade counter) or generating periodic timing events.

```
parameter N = 10;
reg [$clog2(N)-1:0] count;

always @(posedge clk or posedge reset)
  if (reset)
    count <= 0;
  else if (count == N-1)
    count <= 0;
  else
    count <= count + 1;
```

Implementation Tip: Use `$clog2(N)` to dynamically allocate the correct number of bits for the counter width.

Synchronous vs Asynchronous Counters

Counters can be:

- **Synchronous:** All flip-flops share the same clock, making them predictable and easier to analyze.
- **Asynchronous (Ripple):** Flip-flops are clocked by the output of preceding flip-flops. These are simpler but prone to glitches and are not suitable for high-speed systems.

Applications of Counters

- **Timing and delay generation:** Generate specific delays using clock division.

- **Frequency division:** Reduce clock rate for slow peripherals (e.g., LEDs, UART).
- **Sequence generation:** Control signal sequencing in FSMs and datapaths.
- **Event counting:** Count occurrences of input pulses (e.g., sensor signals).
- **Address generation:** Index data in memory or FIFOs.

Best Practices

- Use parameterized designs to make counters reusable.
- Implement synchronous resets in designs targeting FPGAs.
- Always validate counters with simulation to ensure rollover and terminal conditions are correctly handled.

Counters form the basis of many time-based and sequential control functions in digital systems. With Verilog, designers can easily define and customize counters of various types to suit application needs.

4.5 Shift Registers

Shift registers are sequential logic circuits made of flip-flops that store and shift binary data either to the left or right under the control of a clock signal. They are widely used in digital systems for data transfer, serialization, buffering, timing delays, and pattern generation.

A shift register typically consists of a chain of D flip-flops, where the output of one flip-flop is connected to the input of the next. On each active clock edge, data is shifted from one flip-flop to the next, creating a ripple effect through the register.

4.5.1 Types of Shift Registers

- **Serial-In Serial-Out (SISO):** Accepts data one bit at a time at its input and produces the output one bit at a time after a number of clock cycles equal to the register length. It is simple and used for basic data serialization.
- **Serial-In Parallel-Out (SIPO):** Loads data serially but allows all bits to be read simultaneously in parallel. This type is used when data is received serially but needs to be processed in parallel (e.g., UART receivers).
- **Parallel-In Serial-Out (PISO):** Loads data into the register in parallel and then shifts it out serially one bit at a time. This is useful when transmitting multiple-bit data using a single output line.

- **Bidirectional Shift Register:** Can shift data either to the left or right, depending on a direction control signal. This type is used in arithmetic operations (e.g., multiplication/division by 2), digital filters, and counters.

4.5.2 Design Example: 8-Bit SISO Register

The example below shows an 8-bit shift register that shifts in data from the input 'in' and moves it from left to right on each clock cycle.

```
module shift_register_siso (
  input clk,
  input reset,
  input in,
  output reg [7:0] out
);

always @(posedge clk or posedge reset)
  if (reset)
    out <= 8'b00000000;
  else
    out <= {out[6:0], in}; // Right shift

endmodule
```

4.5.3 PISO Shift Register Example

This PISO register loads all 8 bits at once when 'load' is high, and then shifts one bit out on every clock cycle.

```
module shift_register_piso (
  input clk,
  input reset,
  input load,
  input [7:0] in_data,
  output reg out_bit
);

reg [7:0] data;

always @(posedge clk or posedge reset)
  if (reset)
    data <= 8'b0;
```

```
  else if (load)
    data <= in_data;
  else begin
    out_bit <= data[7];
    data <= {data[6:0], 1'b0}; // Shift left
  end

endmodule
```

4.5.4 Bidirectional Shift Register Example

This design supports both left and right shifting controlled by the 'dir' signal.

```
module shift_register_bidir (
  input clk,
  input reset,
  input dir,        // Direction control: 0 = right, 1 = left
  input in_bit,
  output reg [7:0] out
);

always @(posedge clk or posedge reset)
  if (reset)
    out <= 8'b00000000;
  else if (dir)
    out <= {out[6:0], in_bit}; // Shift right
  else
    out <= {in_bit, out[7:1]}; // Shift left

endmodule
```

Applications of Shift Registers

- **Serial Communication:** UART and SPI protocols use shift registers to convert data between serial and parallel formats.
- **Data Buffering:** Temporary storage and delay lines for synchronizing data streams.
- **Bit Manipulation:** Used in bit-serial operations like encryption and error detection.
- **LED and LCD Control:** Drives display elements in sequences using limited I/O.

- **Finite State Machines:** Shift registers are used to track states or patterns.
- **Counters and Arithmetic Units:** Help implement binary division or multiplication by powers of 2.

Advantages of Shift Registers

- Efficient hardware usage for serialization.
- Simple design and implementation in Verilog.
- Scalable and flexible for different bit widths.

Limitations

- Slower access to all bits compared to RAM or registers.
- Require multiple clock cycles for full-word operations in serial mode.

4.6 Edge Detection and Pulse Generation

Edge detection and pulse generation are essential techniques in sequential logic design, especially when dealing with asynchronous inputs or timing-sensitive operations. These mechanisms allow a digital system to recognize when a signal changes value—typically from low to high (rising edge) or high to low (falling edge)—and respond accordingly with a single-cycle pulse or trigger.

4.6.1 Why Edge Detection is Important

In many real-world systems, input signals such as buttons, sensors, or communication lines arrive asynchronously with respect to the system clock. To handle these inputs reliably:

- Edge detection enables the system to act only when a change (transition) is observed.
- Pulse generation ensures that the resulting action (like triggering a counter or state transition) occurs for just one clock cycle.

Without edge detection, inputs might be sampled multiple times or missed entirely, leading to incorrect behavior.

4.6.2 Rising Edge Detection

A rising edge occurs when a signal transitions from logic `0` to logic `1`. Detecting this transition requires storing the previous signal value and comparing it with the current one.

```
reg signal_edge;
reg prev_signal;

always @(posedge clk) begin
  prev_signal <= signal;
  signal_edge <= signal & ~prev_signal; // Detect rising edge
end
```

In this example:

- `prev_signal` holds the previous state of the signal.
- `signal_edge` becomes `1` for exactly one clock cycle after a rising transition.

4.6.3 Falling Edge Detection

A falling edge occurs when a signal transitions from `1` to `0`:

```
reg signal_edge;
reg prev_signal;

always @(posedge clk) begin
  prev_signal <= signal;
  signal_edge <= ~signal & prev_signal; // Detect falling edge
end
```

This technique can be used to detect the release of a button, end of data burst, or end-of-frame conditions.

4.6.4 Pulse Generation

Pulse generation creates a narrow, one-cycle-wide pulse based on a condition or event, often using edge detection as a basis.

Example: Generate a pulse when a button is pressed (rising edge).

```
reg [1:0] sync;
wire button_rise;

always @(posedge clk) begin
```

```
  sync <= {sync[0], button};  // 2-stage synchronizer
end

assign button_rise = sync[1] & ~sync[0];  // Rising edge
    detection
```

This design:

- Synchronizes the asynchronous input (`button`) to the system clock to avoid metastability.
- Detects a clean rising edge transition from the synchronized signal.
- Produces a single-clock pulse on `button_rise`.

4.6.5 Metastability and Synchronization

Directly sampling asynchronous signals can lead to metastability—an undefined logic state between 0 and 1—which causes erratic behavior.

To safely detect edges:

- Use a two-flip-flop synchronizer.
- Detect edges after synchronization to ensure clean transitions.

Two-Flip-Flop Synchronizer:

```
reg [1:0] sync;

always @(posedge clk)
  sync <= {sync[0], async_signal};
```

This effectively filters glitches and ensures safe clock domain crossing.

4.6.6 Applications of Edge Detection and Pulses

- Button debouncing and press detection
- UART start bit detection
- State machine triggering
- One-shot event handling
- Clock division and toggling

Edge detection and pulse generation are fundamental for responsive and stable digital systems. When dealing with asynchronous events, using synchronizers and single-clock-cycle pulses ensures accurate and safe operation in Verilog-based FPGA designs. Table 4.2 summarizes the key differences between rising and falling edge detection in Verilog, including their logic expressions, use cases, and typical applications in digital design.

Table 4.2: Comparison of rising edge vs. falling edge detection in Verilog

Aspect	**Rising Edge Detection**	**Falling Edge Detection**
Definition	Detects a transition from logic 0 to logic 1	Detects a transition from logic 1 to logic 0
Verilog Expression	`signal & ~prev_signal`	`~signal & prev_signal`
Use Cases	Button press, start of data transfer, signal assertion	Button release, end of data frame, signal deassertion
Output Pulse Width	One clock cycle after rising edge	One clock cycle after falling edge
Edge Triggering Method	Positive edge comparator logic	Negative edge comparator logic
Waveform Signature	Rising slope ($0 \rightarrow 1$) triggers output	Falling slope ($1 \rightarrow 0$) triggers output
Typical Applications	Start-bit detection, toggles, counters	Stop-bit detection, falling-triggered resets

4.7 Clocking and Reset Strategies

Efficient and reliable operation of sequential digital systems depends on proper management of clocks and resets. These foundational signals dictate how data is registered, synchronized, and initialized within a design. This section explores common strategies used in Verilog-based designs for clocking and resetting logic.

4.7.1 Synchronous vs. Asynchronous Reset

Resets are essential for initializing registers and state machines to known values. Two main reset approaches are used in digital design: synchronous and asynchronous.

Synchronous Reset

A synchronous reset affects registers only on the active edge of the clock (typically the rising edge). This means the reset condition must be held until the next clock edge to be effective.

```
1 always @(posedge clk) begin
```

```
  if (rst)
    q <= 0;
  else
    q <= d;
end
```

Advantages:

- Fully compatible with static timing analysis tools.
- Avoids metastability issues at the register level.

Disadvantages:

- Requires clock to be operational during reset.
- May increase logic complexity if reset is used widely.

Asynchronous Reset

An asynchronous reset affects registers immediately upon activation, regardless of the clock. It is usually used in situations requiring rapid response or global reset initialization.

```
always @(posedge clk or posedge rst) begin
  if (rst)
    q <= 0;
  else
    q <= d;
end
```

Advantages:

- Fast response; can reset even if the clock is stopped.
- Simple to implement in startup and emergency reset cases.

Disadvantages:

- Requires synchronization when de-asserting the reset signal to avoid metastability.
- More difficult to analyze timing; not always synthesis-friendly.

4.7.2 Clock Domain Crossing (CDC)

Modern digital systems often operate using multiple clock domains. Communication between modules clocked at different frequencies can lead to metastability, where the receiving flip-flop enters an undefined state.

Common Techniques to Mitigate CDC Issues:

- **Dual Flip-Flop Synchronizer:** Used for synchronizing single-bit control signals across clock domains.
- **Asynchronous FIFOs:** Used for transferring multi-bit data between domains with different clocks.
- **Handshake Protocols:** Ensure proper data transfer acknowledgment between sender and receiver.

Without proper CDC design, systems can experience intermittent bugs that are hard to reproduce and debug.

4.7.3 Clock Gating

Clock gating is a common power optimization technique where parts of a circuit are temporarily disabled by gating the clock. Instead of continuously clocking all flip-flops, logic blocks that are not in use can have their clock signal stopped to reduce dynamic power.

Example of Gated Clock Logic:

```
assign clk_enable = (mode == IDLE) ? 1'b0 : 1'b1;
always @(posedge clk & clk_enable)
  reg_data <= input_data;
```

Benefits:

- Reduces power consumption in idle or inactive regions.
- Helps meet power budgets in low-power embedded systems.

Design Considerations:

- Gated clock signals must be glitch-free to avoid false triggering.
- Use synthesis-supported clock gating cells (e.g., Vivado inferred logic) rather than manual logic combinations.
- Clock gating logic must be carefully verified through formal or static tools.

Clock and reset strategies are not only foundational to system behavior but also impact timing closure, reliability, and power efficiency. Synchronous resets are preferred for timing safety, while asynchronous resets are favored for global initialization. Careful synchronization is required across clock domains, and clock gating enables significant power savings with proper implementation discipline.

4.8 Design Examples

This section presents practical examples of sequential circuit design using Verilog HDL. Each design showcases specific timing, control, or counting mechanisms commonly used in FPGA or ASIC-based systems. These examples serve as templates for real-world embedded applications.

4.8.1 4-Bit Synchronous Counter

This counter increments its value with each rising edge of the clock signal. It resets when the reset signal is high. All bits change synchronously, making it suitable for synchronous digital systems.

```
module counter_4bit (
  input clk,
  input rst,
  output reg [3:0] count
);
  always @(posedge clk or posedge rst) begin
    if (rst)
      count <= 4'b0000;
    else
      count <= count + 1;
  end
endmodule
```

4.8.2 Binary-Coded Decimal (BCD) Counter

This BCD counter cycles through decimal values 0 to 9 in binary format. Once the count reaches 9, it rolls over to 0. These counters are frequently used in digital clocks, calculators, and numeric displays.

```
module bcd_counter (
  input clk,
  input rst,
  output reg [3:0] bcd
);
  always @(posedge clk or posedge rst) begin
    if (rst)
      bcd <= 0;
    else if (bcd == 9)
      bcd <= 0;
```

```
    else
      bcd <= bcd + 1;
  end
endmodule
```

4.8.3 Simple Stopwatch Design

This stopwatch counts seconds when enabled. A clock divider converts the system clock to a slower pulse (e.g., 1Hz). This design is helpful in creating timers, delay circuits, or time interval measurement tools.

```
module stopwatch (
  input clk,
  input rst,
  input start,
  output reg [7:0] seconds
);
  reg [23:0] clk_div;
  always @(posedge clk or posedge rst) begin
    if (rst) begin
      clk_div <= 0;
      seconds <= 0;
    end else if (start) begin
      clk_div <= clk_div + 1;
      if (clk_div == 24'd10_000_000) begin
        seconds <= seconds + 1;
        clk_div <= 0;
      end
    end
  end
endmodule
```

4.8.4 Traffic Light Controller

This example implements an FSM to manage traffic light transitions. Each state corresponds to a light (RED, GREEN, YELLOW), and timing determines transitions. Useful in control systems and FSM demonstration projects.

```
module traffic_light (
  input clk,
  input rst,
  output reg [2:0] lights  // RGY
```

```
);
  typedef enum reg [1:0] {RED, GREEN, YELLOW} state_t;
  state_t state;
  reg [3:0] timer;

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      state <= RED;
      timer <= 0;
    end else begin
      timer <= timer + 1;
      case (state)
        RED:    if (timer == 5)  begin
        state <= GREEN;  timer <= 0; end
        GREEN:  if (timer == 4)  begin
        state <= YELLOW; timer <= 0;  end
        YELLOW: if (timer == 2)  begin
        state <= RED;    timer <= 0; end
      endcase
    end
  end

  always @(*) begin
    case (state)
      RED:    lights = 3'b100;
      GREEN:  lights = 3'b010;
      YELLOW: lights = 3'b001;
    endcase
  end
endmodule
```

4.8.5 UART Bit Transmitter

This module serializes 8-bit data by prepending a start bit and appending a stop bit. Data is shifted out bit by bit using a shift register and a busy signal ensures synchronization.

```
module uart_tx (
  input clk,
  input rst,
  input start,
  input [7:0] data_in,
  output reg tx,
```

```
  output reg busy
);
  reg [3:0] bit_cnt;
  reg [9:0] shift_reg;

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      tx <= 1;
      busy <= 0;
      bit_cnt <= 0;
    end else if (start && !busy) begin
      shift_reg <= {1'b1, data_in, 1'b0}; // stop + data + start
      bit_cnt <= 0;
      busy <= 1;
    end else if (busy) begin
      tx <= shift_reg[0];
      shift_reg <= shift_reg >> 1;
      bit_cnt <= bit_cnt + 1;
      if (bit_cnt == 9)
        busy <= 0;
    end
  end
endmodule
```

These design examples provide essential building blocks for real-world applications and serve as starting points for more complex digital systems such as processors, communication interfaces, and controllers.

4.9 Modeling Sequential Circuits

Sequential circuits form the foundation of digital systems that require memory or timed responses. Unlike combinational logic, which outputs only depend on current inputs, sequential circuits incorporate memory elements (like flip-flops and latches) to store and update state over time. In Verilog, these circuits are modeled using "always" blocks sensitive to clock edges and reset conditions.

4.9.1 Sequential Always Blocks

Sequential logic is typically modeled using 'always' blocks with sensitivity to clock and optional reset signals. The correct use of non-blocking assignments ('<=') ensures that

all variables update simultaneously at the end of the time step, avoiding unintended dependencies or race conditions.

```
always @(posedge clk or posedge rst)
  if (rst)
    q <= 0;
  else
    q <= d;
```

Using non-blocking assignments is essential for modeling flip-flops correctly. Blocking assignments (‘=’) are reserved for combinational logic or testbench-only constructs.

4.9.2 State Retention

Sequential circuits rely on the ability to retain and update state values across multiple clock cycles. This behavior is typically implemented using:

- **Flip-Flops:** Edge-triggered memory elements that update on rising or falling clock edges.
- **Latches:** Level-sensitive devices that update when enabled.

For example, a register made of D flip-flops stores an 8-bit value:

```
reg [7:0] data;

always @(posedge clk)
  data <= new_value;
```

4.9.3 Initialization

During simulation, initialization is often done with the ‘initial’ block to set starting values for variables. This block is not synthesizable and is used for testbenches or functional simulations only.

```
initial begin
  count = 0;
  state = IDLE;
end
```

For synthesis and hardware deployment, initial values must be assigned via reset signals—either synchronous or asynchronous:

- **Synchronous reset:** Occurs on the active clock edge.

- **Asynchronous reset:** Occurs immediately when the reset is asserted.

```
always @(posedge clk or posedge rst)
  if (rst)
    count <= 0;
  else
    count <= count + 1;
```

4.9.4 Clock Sensitivity and Gating Considerations

Clock signals drive the timing of sequential logic. Designers must ensure that only one edge of the clock (usually the rising edge) is used across a module to avoid metastability or logic hazards.

Avoid using gated clocks unless absolutely necessary. Instead, enable signals are preferred:

```
always @(posedge clk)
  if (enable)
    q <= d;
```

4.9.5 Best Practices for Modeling

- Always use non-blocking assignments in sequential logic.
- Keep combinational logic in separate "always @(*)" blocks.
- Separate state register and next-state logic in FSM designs.
- Include reset behavior to initialize all storage elements.
- Avoid latches by fully defining outputs for all conditional paths.

Accurate modeling of sequential circuits ensures synthesizability, reliable behavior in hardware, and predictable simulation results. Mastery of clocking, reset mechanisms, and timing semantics is essential for RTL designers targeting FPGA or ASIC platforms.

4.10 Simulation of Sequential Logic

Simulation of sequential circuits is a critical step in digital system design, as it allows verification of functional correctness, timing behavior, and edge-case conditions before deploying the design on hardware. Sequential logic typically includes state-holding elements like flip-flops and latches, making their behavior dependent on clocking and reset

conditions. Simulation ensures these behaviors are captured accurately under various test conditions.

Purpose of Simulation

Simulation helps to:

- Verify correct state transitions in FSMs.
- Confirm synchronous behavior with respect to clock edges.
- Validate reset sequences and system initialization.
- Detect functional errors early in the design phase.
- Perform timing estimation and waveform analysis.

Testbench Structure

A testbench is a non-synthesizable module used solely for simulation. It typically includes:

- Signal declarations and initialization.
- Clock generation block.
- Stimulus generation using `initial` blocks.
- Optional `$display`, `$monitor`, and assertions for output verification.
- Termination logic using `$finish`.

Example: Testbench for a 4-bit Synchronous Counter

```
module tb_counter;
  reg clk = 0, rst = 0;
  wire [3:0] count;

  // Instantiate the counter module
  counter uut (
    .clk(clk),
    .rst(rst),
    .count(count)
  );

  // Clock generation: toggles every 5 time units (period = 10)
```

```
13    always #5 clk = ~clk;
14
15    // Test stimulus
16    initial begin
17      // Apply reset
18      rst = 1;
19      #10;
20      rst = 0;
21
22      // Run simulation for 100 time units
23      #100;
24
25      // Finish simulation
26      $finish;
27    end
28
29    // Optional display of counter value
30    initial begin
31      $display("Time\tclk\trst\tcount");
32      $monitor("%g\t%b\t%b\t%b", $time, clk, rst, count);
33    end
34  endmodule
```

Expected Behavior

- During the initial reset phase (`rst = 1`), the counter is set to zero.
- Once reset is deasserted (`rst = 0`), the counter increments with each rising edge of the clock.
- The waveform should show a clear sequence of 4-bit binary values starting from 0.

Waveform Analysis

Simulation tools like GTKWave or ModelSim can display waveforms that represent signal changes over time. In these waveforms, you can verify:

- Correct rising edge triggering.
- Reset functionality and recovery.
- Continuous and correct increment of the `count` signal.

Best Practices

- Always test both reset assertion and deassertion.
- Simulate for sufficient time to observe full behavior.
- Use assertions to catch incorrect transitions or values.
- Incorporate corner cases in test sequences (e.g., maximum count value).
- Include comments and descriptive signal names for better traceability.

Simulation not only helps debug logic errors early but also provides confidence that the design will work as expected once implemented in hardware. It bridges the gap between code and circuit by offering a virtual observation of time-dependent behaviors.

4.11 Synthesis Considerations

Synthesis is the process of converting RTL descriptions written in Verilog into a gate-level netlist that can be mapped onto hardware such as FPGAs or ASICs. While simulation checks functionality, synthesis ensures physical realizability. Therefore, writing synthesis-friendly code is essential for successful hardware implementation.

1. Avoid Unintended Latches

Unintended latches occur when combinational logic does not specify outputs for all possible input conditions, causing the synthesis tool to infer a level-sensitive latch to retain state. These can lead to unpredictable or hazardous hardware behavior. To avoid this:

- Always provide default assignments inside `always @(*)` blocks.
- Ensure all branches of conditional logic assign values to outputs.
- Use full `case` statements with `default` branches.

2. Ensure Complete State Coverage

In FSMs, every possible binary encoding of the state register must be accounted for. Omitting states or transitions can:

- Cause the FSM to hang in undefined states.
- Result in logic bloat due to synthesis optimizations.
- Complicate formal verification or static timing analysis.

Include a `default` clause or formal assertion to catch illegal states during simulation.

3. Use Synchronous Clocking Practices

Proper clock design is crucial:

- Avoid clock gating logic; instead, use enable signals.
- Ensure a single clock source is used per domain to minimize clock skew.
- Avoid using logic signals as clocks, which may cause glitches or metastability.

4. Manage Reset Strategies Carefully

Reset logic ensures the design starts in a known state:

- Use asynchronous reset for rapid global initialization (common in FPGAs).
- Use synchronous reset to ensure predictable timing (suitable for ASICs).
- Always verify that all flip-flops are properly reset in RTL.

5. Avoid Combinational Loops

Combinational loops are logic paths without flip-flop separation, leading to:

- Synthesis errors or warnings.
- Oscillatory behavior or metastability.
- Difficulty in achieving timing closure.

All feedback paths should include at least one register stage.

6. Use Non-Blocking Assignments in Sequential Blocks

To avoid race conditions and mismatches between simulation and synthesized behavior:

- Use non-blocking assignments (`<=`) inside `always @(posedge clk)` blocks.
- Use blocking assignments (`=`) only in combinational logic.

7. Avoid Non-Synthesizable Constructs

Some Verilog features are meant for simulation only:

- `$display`, `$monitor`, `$finish`, and `$stop` are ignored during synthesis.
- `initial` blocks are not supported in FPGA hardware and must be replaced by reset logic.

8. Optimize for Area, Speed, and Power

Synthesis tools offer trade-offs. You can guide them with design choices:

- Use pipelining to improve performance.
- Minimize resource duplication with shared logic.
- Consider clock gating and operand isolation for power efficiency.

9. Constraint Definition and Timing Analysis

Use synthesis constraint files (e.g., XDC in Vivado or SDC in Quartus) to define:

- Clock definitions and relationships.
- Input/output delays for accurate setup/hold analysis.
- False paths and multicycle paths for optimization.

10. Verify with Synthesis Reports

After synthesis, analyze:

- **Area Report:** Resource usage (LUTs, FFs, BRAMs, DSPs).
- **Timing Report:** Setup/hold violations, worst slack.
- **Utilization Summary:** Logic distribution, congestion.

These reports help identify design bottlenecks and areas for optimization.

In summary, good synthesis practices ensure your Verilog code not only works in simulation but also synthesizes into robust, efficient hardware. Understanding the nuances of synthesis constraints, clocking strategy, and resource management is critical to achieving a functional and optimized design.

4.12 Common Pitfalls

When designing sequential logic in Verilog, both beginners and experienced engineers can inadvertently introduce design errors that lead to mismatches between simulation and synthesis, cause unstable behavior in hardware, or fail to meet timing constraints. Understanding and avoiding these pitfalls is critical for creating robust and synthesizable hardware designs.

Incorrect Edge Sensitivity

Verilog supports event-based modeling, where operations are triggered on rising (`posedge`) or falling (`negedge`) clock edges. Failing to explicitly define edge sensitivity or selecting the wrong edge can introduce subtle bugs or synthesis errors.

- Always use `posedge` for synchronous designs unless negative-edge triggering is explicitly required.
- Include asynchronous reset in the sensitivity list when needed.

Incorrect Example (Level-sensitive):

```
always @(clk) // Missing edge qualifier; may infer latch or
    glitchy behavior
  count <= count + 1;
```

Correct Example (Edge-triggered):

```
always @(posedge clk or posedge rst)
  if (rst)
    count <= 0;
  else
    count <= count + 1;
```

Mixing Blocking and Non-Blocking Assignments

Using blocking assignments (`=`) in sequential logic can cause race conditions or unintended ordering. Always use non-blocking assignments (`<=`) inside `always @(posedge clk)` blocks to model flip-flop behavior correctly.

- Blocking assignments are useful in purely combinational logic.
- Avoid mixing blocking and non-blocking assignments in the same block.

Problematic Example (Mixed assignments):

```
always @(posedge clk) begin
  temp = data;        // blocking assignment
  result <= temp;     // non-blocking assignment
end
```

Corrected Example:

```
always @(posedge clk) begin
  temp <= data;
  result <= temp;
end
```

Glitches Due to Incomplete Combinational Logic

In combinational blocks (`always @(*)`), all possible input cases must be covered. Failing to do so can result in inferred latches or unstable output (glitches) in synthesized hardware.

Problematic Example (Latch inferred):

```
always @(*) begin
  if (sel)
    out = a;
  // Missing 'else' path; infers a latch
end
```

Safe Example (Fully specified logic):

```
always @(*) begin
  if (sel)
    out = a;
  else
    out = b;
end
```

Improper Reset Handling

Proper reset logic is essential to bring sequential circuits into a known state after configuration or power-up.

- Always specify initial states using reset logic rather than relying on initial blocks.
- Asynchronous resets react immediately; synchronous resets are aligned with clock edges.
- In FSMs, failing to reset the state variable can cause undefined behavior.

Using Simulation-Only Constructs in Synthesis

Verilog supports simulation-only features like `$display`, `$monitor`, and `initial` blocks. These do not map to hardware and must not be used in synthesizable designs.

Simulation-Only Example:

```
initial begin
  clk = 0;
  forever #5 clk = ~clk;
end
```

Synthesis-Safe Clocking: Use dedicated clock input pins and global clock buffers in hardware.

Other Pitfalls

- **Inferred Latches:** Caused by missing `else` conditions in `if` statements or incomplete `case` statements.
- **Case Statement Omissions:** Always include a `default` clause to avoid inferred logic hazards.
- **Clock Gating in RTL:** Manually gating clocks using `if (enable) clk = ...` is not synthesizable; use enable signals instead.
- **Unused or Floating Signals:** Unused logic may get optimized away. Tie off or monitor unused inputs/outputs explicitly.
- **Overclocking or Underclocking:** Incorrect clock assumptions can result in setup/hold violations.

Awareness of these common pitfalls allows designers to write Verilog code that is both portable and robust across simulation, synthesis, and physical implementation stages. Adhering to best practices in RTL design significantly enhances code maintainability, testability, and correctness.

4.13 Summary

Sequential logic is essential for building digital systems that remember past events and make decisions over time. Unlike combinational logic, which only depends on current inputs, sequential circuits use memory elements like flip-flops and latches to store state information.

This chapter presented the key components and concepts behind sequential logic:

- **Latches and Flip-Flops:** Devices that store single-bit data. Flip-flops operate on clock edges to ensure predictable timing.
- **Registers and Register Banks:** Groups of flip-flops that store multi-bit values, commonly used in datapaths and processors.
- **Counters:** Circuits for counting in various modes (up, down, bidirectional), often used for control and timing.

- **Shift Registers:** Structures for moving data serially and converting between serial and parallel formats.
- **Edge Detection and Pulse Generation:** Methods to sense signal transitions and generate time-limited pulses.
- **Clocking and Reset Strategies:** Crucial design practices that ensure correct operation and initialization of sequential systems.
- **Design Examples:** Applications such as a 4-bit counter, traffic light controller, stopwatch, and UART transmitter demonstrated practical usage of sequential constructs.
- **Simulation and Synthesis:** Techniques were introduced for verifying sequential circuits through testbenches and preparing them for hardware implementation.
- **Common Pitfalls:** Guidance was provided to avoid typical mistakes like incorrect reset handling, clock edge mismatches, and improper assignment usage.

Mastery of sequential logic is fundamental for developing complex and robust digital designs. Understanding the interplay of memory, control, and timing enables efficient and reliable system development.

The laboratory exercises for Chapter 4: *Sequential Logic Design* consist of Lab 9: Counters and Clock Divider Design, Lab 10: Display Counter on FPGA, and Lab 11: Multiplier Design, all of which are provided in the Appendix section.

4.14 Exercises

1. **D Flip-Flop:** Design a D flip-flop module in Verilog with an asynchronous reset. Simulate its behavior for various inputs and clock transitions.
2. **3-bit Synchronous Counter:** Implement a 3-bit up-counter with synchronous reset and enable. Verify it counts from 0 to 7 and wraps around correctly.
3. **Shift Register:** Create a 4-bit serial-in, parallel-out (SIPO) shift register using behavioral modeling. Demonstrate its operation using a testbench.
4. **Finite State Machine (FSM):** Design a Moore-type FSM that detects the sequence 1011 on a serial input. Use one-hot state encoding and describe the output logic.
5. **Traffic Light Controller (Mini-Project):** Write a Verilog module to control a traffic light for a simple intersection. Use a state machine with timing delays simulated using counters.

6. **Blocking vs Non-Blocking:** Illustrate the difference between blocking (‘=’) and non-blocking (‘<=’) assignments using a small example. Simulate both versions and compare the outputs.

7. **Parameterized Counter (Advanced):** Create a parameterized counter module that supports adjustable bit-width and count direction (up/down). Verify it through simulation.

8. **Clock Divider:** Design a clock divider module that reduces the frequency of an input clock by a factor of 8. Show its waveform using a testbench.

Bibliography

[1] C. Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes, 2004.

[2] M. Manohar and J. Bhasker, *Digital System Design Using FPGA: Implementation with Verilog and VHDL*. New York, NY, USA: McGraw-Hill, 2017.

[3] R. Merrick, *Getting Started with FPGAs: Digital Circuit Design, Verilog, and VHDL for Beginners*. San Francisco, CA: No Starch Press, 2023.

[4] S. Brown and Z. G. Vranesic, *Fundamentals of Digital Logic with Verilog Design*, 3rd ed. New York, NY, USA: McGraw-Hill, 2014.

[5] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[6] D. Ramanathan, "Part 15: Sequential Logic Using Verilog," *Maker.io – DigiKey*, Jun. 4, 2025. [Online]. Available: https://www.digikey.com/en/maker/tutorials/2025/part-15-sequential-logic-using-verilog

[7] ChipVerify.com, "Sequential Logic." [Online]. Available: https://www.chipverify.com/digital-fundamentals/sequential-logic

[8] S. Arar, "Using Verilog to Describe a Sequential Circuit," *All About Circuits*, Mar. 1, 2019. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/using-verilog-to-describe-a-sequential-circuit/

[9] Y. Penta and R. Islam, "Design and Verification of a Synchronous First In First Out (FIFO)," *arXiv preprint*, Apr. 15, 2025. [Online]. Available: https://arxiv.org/abs/2504.10901

Chapter 5

Finite State Machines and Control

Chapter Objectives

- Understand the principles and types of finite state machines (FSMs).
- Design and implement Moore and Mealy FSMs in Verilog.
- Simulate, verify, and debug FSM-based control logic systems.

5.1 Introduction to Finite State Machines

A FSM is a conceptual model used in digital system design to implement control logic. It operates by transitioning between a limited number of predefined states based on the current inputs. Each state represents a specific operational mode of the system, and the transitions between states occur in response to changes in input conditions. The FSM may also produce outputs that depend on either the current state alone (in the case of Moore machines) or on both the state and input (in the case of Mealy machines).

FSMs are commonly used in hardware to manage control flow, coordinate sequencing operations, interpret user interactions, and enforce communication protocols. Examples include processors, data communication interfaces such as UART and SPI, controllers for vending machines, traffic lights, and many more embedded systems.

By describing FSMs in Verilog, designers can develop structured, modular, and synthesizable logic that reflects real-time behavior. This makes FSMs a core component of digital circuit design, enabling reliable and maintainable hardware implementations for a wide range of applications.

5.2 Types of FSMs

FSMs are classified based on how their outputs are generated. The two most common models are the Moore machine and the Mealy machine. Each has unique characteristics that influence design complexity, timing behavior, and output stability.

5.2.1 Moore Machine

In a Moore machine , the output depends only on the current state of the system. This means the output changes only when the FSM transitions to a new state, not immediately in response to input changes.

Features:

- Outputs are registered and synchronized with the clock.
- Simplifies timing analysis since outputs don't glitch on input transitions.
- May require more states to achieve the same logic as a Mealy machine.

Example:

```
always @(posedge clk)
  state <= next_state;

always @(*) begin
  case (state)
    IDLE: output = 0;
    ACTIVE: output = 1;
    DONE: output = 0;
  endcase
end
```

5.2.2 Mealy Machine

In a Mealy machine, the output is a function of both the current state and the input. This allows the FSM to react immediately to changes in input, without waiting for a clock edge.

Features:

- Potentially fewer states compared to Moore.
- Outputs can change as soon as inputs change (may be combinational).
- More sensitive to input glitches; careful design needed to avoid hazards.

Example:

```
always @(posedge clk)
  state <= next_state;

always @(*) begin
  case (state)
    IDLE: begin
      next_state = (start) ? ACTIVE : IDLE;
      output = 0;
    end
    ACTIVE: begin
      next_state = (done) ? DONE : ACTIVE;
      output = (input_signal) ? 1 : 0;
    end
  endcase
end
```

5.2.3 Comparison of Moore and Mealy FSMs

- **Moore Machine:**
  - Output depends only on current state.
  - More stable and synchronous outputs.
  - Easier to design and test.
  - Requires more states in some designs.
- **Mealy Machine:**
  - Output depends on both state and input.
  - Can respond faster to inputs.
  - May use fewer states.
  - Requires more careful timing analysis.

Both Moore and Mealy models are widely used in FSM design. The choice between them depends on design requirements such as output timing, stability, and state complexity. Table 5.1 compares their key characteristics to aid in selecting the appropriate model.

Table 5.1: Comparison of Moore and Mealy FSMs

Feature	**Moore Machine**	**Mealy Machine**
Output depends on	Only the current state	Current state and inputs
Output timing	Changes only on state transitions (clock edge)	Can change immediately with input (before clock edge)
Glitch susceptibility	Low (more stable)	Higher (due to immediate response to inputs)
State count	May require more states	Often fewer states
Design complexity	Simpler and more predictable	More complex timing behavior
Use cases	Preferred in synchronous systems with stable output	Useful for fast response and compact logic

5.3 FSM Design Process

Designing a FSM involves a structured approach to ensure correct functionality, efficiency, and synthesizability. The following steps outline a typical FSM design process:

1. **Define the problem or control sequence.**
   Clearly describe what the FSM needs to do. Identify input conditions, required responses, and when outputs should be triggered. This forms the functional specification.

2. **Identify the number of states.**
   Based on the problem, determine distinct operational modes or steps. Each unique situation the machine must recognize or handle becomes a state.

3. **Draw the state diagram with transitions.**
   Create a graphical diagram showing all states as circles and transitions between them as arrows. Label transitions with input conditions and annotate any output actions.

4. **Assign binary codes to the states.**
   Use binary encoding (e.g., binary, one-hot, or Gray code) to represent each state. This step is essential for hardware implementation and affects complexity and speed.

5. **Write the transition and output logic.**
   Use truth tables, state equations, or directly define next-state and output logic using combinational expressions based on inputs and current state.

6. **Implement in Verilog.**
   Translate the design into synthesizable Verilog code using the two-process FSM template:

- A sequential block for updating the current state.
- A combinational block for computing the next state and output.

7. **Simulate and test.**
   Use simulation tools to verify correctness under all input conditions. Check state transitions, output timing, and verify edge cases. Debug and revise as necessary.

This systematic process ensures that FSMs are well-structured, easy to debug, and suitable for synthesis on FPGA or ASIC platforms. It is applicable to a wide range of digital systems such as traffic light controllers, protocol handlers, and pipeline controllers.

5.4 State Diagrams and Transition Tables

State diagrams and transition tables are visual and tabular methods used to describe the behavior of an FSM. These tools help designers visualize how the system transitions between states based on inputs and what outputs are generated during these transitions.

State Diagrams

A **state diagram** uses:

- **Circles** to represent the states.
- **Arrows** to represent transitions between states.
- **Labels** on arrows indicating input conditions and, optionally, output actions (especially in Mealy machines).

State diagrams provide an intuitive and high-level understanding of the system, making them useful in both design and documentation stages.

Transition Tables

A **transition table** provides the same information in tabular form:

- Rows list current states and input conditions.
- Columns indicate the next state and output.

This form is more suitable for implementation and automation in synthesis tools or HDL code.

5.4.1 Example: Sequence Detector (Detecting "1011")

A sequence detector monitors a serial input stream to identify when a specific pattern (e.g., "1011") appears. The FSM outputs '1' when the full pattern has been detected.

States:

- **S0:** Initial state, no bits matched.
- **S1:** Detected '1'.
- **S2:** Detected '10'.
- **S3:** Detected '101'.
- **S4:** Detected '1011' (output 1).

State Diagram:

Figure 5.1 shows the Moore FSM for detecting the "1011" sequence. Each state represents progress in matching the pattern. Transitions are labeled with the input bit, and the output is asserted only in state S4.

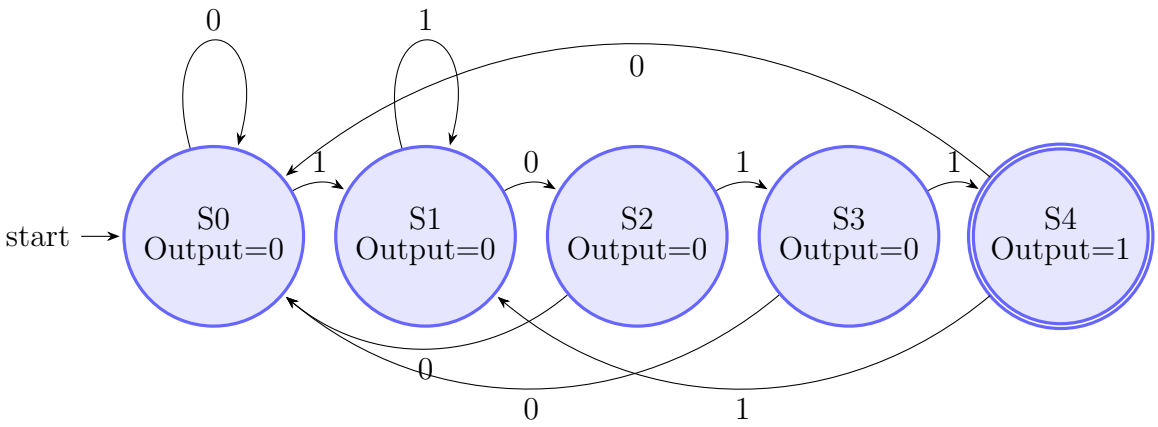


Figure 5.1: Moore FSM for detecting the sequence "1011"

Transition Table:

Table 5.2 lists the state transitions and output values for the 1011 sequence detector. Each row shows how the FSM moves from one state to another based on the input bit, along with the corresponding output for each state.

Note: This is a Moore machine—output '1' is generated only in state S4.

This example shows how both diagrammatic and tabular representations help in understanding and implementing FSM-based sequence detection logic.

```
module sequence_detector_1011 (
    input wire clk,
    input wire rst_n,
```

Table 5.2: Transition table for 1011 sequence detector (Moore FSM)

Current State	Input (X)	Next State	Output (Z)
S0	0	S0	0
S0	1	S1	0
S1	0	S2	0
S1	1	S1	0
S2	0	S0	0
S2	1	S3	0
S3	0	S0	0
S3	1	S4	0
S4	0	S0	1
S4	1	S1	1

```
    input wire x,           // Serial input
    output reg z            // Output: 1 when "1011" is detected
);

    // State encoding
    typedef enum logic [2:0] {
        S0 = 3'b000,    // Initial state
        S1 = 3'b001,    // Detected '1'
        S2 = 3'b010,    // Detected '10'
        S3 = 3'b011,    // Detected '101'
        S4 = 3'b100     // Detected '1011'
    } state_t;

    state_t current_state, next_state;

    // State register
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            current_state <= S0;
        else
            current_state <= next_state;
    end

    // Next-state logic
    always @(*) begin
        case (current_state)
            S0:    next_state = (x == 1'b1) ? S1 : S0;
```

```
            S1:    next_state = (x == 1'b0) ? S1 : S2;
            S2:    next_state = (x == 1'b1) ? S3 : S0;
            S3:    next_state = (x == 1'b1) ? S4 : S0;
            S4:    next_state = (x == 1'b1) ? S1 : S0;
            default: next_state = S0;
        endcase
    end

    // Output logic (Moore: output based on state only)
    always @(*) begin
        case (current_state)
            S4: z = 1'b1;
            default: z = 1'b0;
        endcase
    end

endmodule
```

5.5 State Encoding Techniques

State encoding is the process of assigning binary values to the symbolic states of a FSM. The choice of encoding significantly affects the complexity, speed, and area of the resulting hardware. Common encoding schemes include Binary, One-Hot, and Gray encoding.

5.5.1 Binary Encoding

In binary encoding, the minimum number of bits required to represent all states is used. For an FSM with N states, $\lceil \log_2 N \rceil$ bits are sufficient. The following Verilog code implements a 4-state finite state machine using binary encoding, where only 2 bits are required to represent all states. This approach minimizes flip-flop usage but may increase combinational logic complexity.

```
module fsm_binary (
    input wire clk,
    input wire rst_n,
    input wire in,
    output reg [1:0] state_out
);

    typedef enum logic [1:0] {
        S0 = 2'b00,
```

```
        S1 = 2'b01,
        S2 = 2'b10,
        S3 = 2'b11
    } state_t;

    state_t current_state, next_state;

    // State register
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            current_state <= S0;
        else
            current_state <= next_state;
    end

    // Next state logic
    always @(*) begin
        case (current_state)
            S0: next_state = (in) ? S1 : S0;
            S1: next_state = (in) ? S2 : S0;
            S2: next_state = (in) ? S3 : S0;
            S3: next_state = (in) ? S0 : S1;
            default: next_state = S0;
        endcase
    end

    // Output current state for observation
    always @(*) begin
        state_out = current_state;
    end

endmodule
```

Advantages:

- Requires fewer flip-flops.
- Minimizes register usage.

Disadvantages:

- Combinational logic can become complex.
- Slower performance in FPGAs due to logic depth.

5.5.2 One-Hot Encoding

In one-hot encoding, each state is assigned a separate bit in the state vector. Only one bit is high (‘1’) at any given time, and all others are low (‘0’). For N states, N flip-flops are required. This Verilog example demonstrates a 4-state FSM using one-hot encoding, where each state is represented by a unique bit position. Although this technique consumes more flip-flops, it simplifies state transition logic and is highly efficient for FPGA implementations.

```
module fsm_one_hot (
    input wire clk,
    input wire rst_n,
    input wire in,
    output reg [3:0] state_out
);

    typedef enum logic [3:0] {
        S0 = 4'b0001,
        S1 = 4'b0010,
        S2 = 4'b0100,
        S3 = 4'b1000
    } state_t;

    state_t current_state, next_state;

    // State register
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            current_state <= S0;
        else
            current_state <= next_state;
    end

    // Next state logic
    always @(*) begin
        case (current_state)
            S0: next_state = (in) ? S1 : S0;
            S1: next_state = (in) ? S2 : S0;
            S2: next_state = (in) ? S3 : S0;
            S3: next_state = (in) ? S0 : S1;
            default: next_state = S0;
        endcase
```

```
    end

    // Output current state
    always @(*) begin
        state_out = current_state;
    end

endmodule
```

Advantages:

- Simplified next-state logic.
- Faster implementation in FPGAs due to parallel decoding.

Disadvantages:

- High flip-flop usage (one per state).
- Less suitable for ASICs where area is critical.

5.5.3 Gray Encoding

Gray encoding ensures that only one bit changes between two successive states. It is typically used in FSMs where minimizing transitions is important, such as asynchronous or noisy environments. The code below shows a 4-state FSM using Gray encoding , in which only one bit changes between consecutive states. This reduces switching activity and is suitable for noise-sensitive or asynchronous designs.

```
module fsm_gray (
    input wire clk,
    input wire rst_n,
    input wire in,
    output reg [1:0] state_out
);

    typedef enum logic [1:0] {
        S0 = 2'b00, // 0
        S1 = 2'b01, // 1
        S2 = 2'b11, // 3
        S3 = 2'b10  // 2
    } state_t;

    state_t current_state, next_state;
```

```
16
17      // State register
18      always @(posedge clk or negedge rst_n) begin
19          if (!rst_n)
20              current_state <= S0;
21          else
22              current_state <= next_state;
23      end
24
25      // Next state logic
26      always @(*) begin
27          case (current_state)
28              S0: next_state = (in) ? S1 : S0;
29              S1: next_state = (in) ? S2 : S0;
30              S2: next_state = (in) ? S3 : S0;
31              S3: next_state = (in) ? S0 : S1;
32              default: next_state = S0;
33          endcase
34      end
35
36      // Output current state
37      always @(*) begin
38          state_out = current_state;
39      end
40
41  endmodule
```

Advantages:

- Reduces transition noise and glitches.
- Suitable for error-prone or power-sensitive designs.

Disadvantages:

- Slightly more complex transition logic.
- Less common in standard FSM synthesis flows.

Table 5.3 compares common state encoding techniques used in FSM design in terms of flip-flop requirements, speed on FPGAs, and logic complexity. Each method has trade-offs that influence the choice depending on resource availability and performance goals.

Table 5.3: Comparison of state encoding techniques

Encoding	Flip-Flops Required	Speed (in FPGAs)	Logic Complexity
Binary Encoding	$\lceil \log_2 N \rceil$	Moderate	High
One-Hot Encoding	N	High	Low
Gray Encoding	$\lceil \log_2 N \rceil$	Moderate	Moderate

5.6 FSM Implementation in Verilog

A FSM can be implemented in Verilog using a structured coding style that includes state declaration, state registers, next-state logic, and output logic. Below is a typical example using a 3-state controller with both Moore and Mealy output styles.

5.6.1 State Declaration

States are typically declared using `parameter` or `localparam` for clarity and ease of debugging:

```
parameter IDLE = 2'b00, LOAD = 2'b01, DONE = 2'b10;
```

This assigns unique binary values to each symbolic state.

5.6.2 State Register

The state register holds the current state and updates on each rising clock edge. An asynchronous reset is included:

```
always @(posedge clk or posedge rst)
  if (rst) state <= IDLE;
  else state <= next_state;
```

This is the sequential part of the FSM that stores the active state.

5.6.3 Next State Logic

The next state is determined combinationally based on the current state and relevant inputs:

```
always @(*) begin
  case (state)
    IDLE: next_state = start ? LOAD : IDLE;
    LOAD: next_state = ready ? DONE : LOAD;
    DONE: next_state = IDLE;
    default: next_state = IDLE;
```

```
  endcase
end
```

This logic ensures proper transitions between states based on the system inputs.

5.6.4 Output Logic

Output logic can be written in two main styles: Moore and Mealy.

Moore-style (output depends only on state):

```
assign output = (state == DONE);
```

Mealy-style (output depends on state and inputs):

```
assign output = (state == LOAD) && input_ready;
```

The Moore model provides more stable outputs, while the Mealy model allows faster responses based on input conditions.

This modular FSM structure—comprising state declarations, transition logic, and output control—is highly synthesizable and forms the basis for many control-oriented digital systems.

5.6.5 Complete FSM Module Example

The following Verilog module demonstrates a complete FSM implementation combining state declaration, state transition logic, and both Moore and Mealy output styles. The FSM transitions through three states—`IDLE`, `LOAD`, and `DONE`—and models a simple control unit that activates a load operation and asserts a done signal once the sequence completes.

```
module fsm_controller (
    input  wire clk,          // Clock input
    input  wire rst,          // Asynchronous reset
    input  wire start,        // Start signal
    input  wire ready,        // Ready signal (used in transition
       )
    input  wire input_ready,  // Input signal for Mealy output
    output wire done_flag,    // Moore output
    output wire load_enable   // Mealy output
);

    // State encoding using parameters
    parameter IDLE = 2'b00,
              LOAD = 2'b01,
```

```
                DONE = 2'b10;

    // State registers
    reg [1:0] state, next_state;

    // State transition logic (sequential)
    always @(posedge clk or posedge rst) begin
        if (rst)
            state <= IDLE;
        else
            state <= next_state;
    end

    // Next state logic (combinational)
    always @(*) begin
        case (state)
            IDLE: next_state = (start) ? LOAD : IDLE;
            LOAD: next_state = (ready) ? DONE : LOAD;
            DONE: next_state = IDLE;
            default: next_state = IDLE;
        endcase
    end

    // Moore-style output: depends only on current state
    assign done_flag = (state == DONE);

    // Mealy-style output: depends on current state and input
    assign load_enable = (state == LOAD) && input_ready;

endmodule
```

The following Verilog testbench stimulates the FSM controller with a sequence of input signals (`start`, `ready`, and `input_ready`) to verify correct state transitions and the behavior of both Moore (`done_flag`) and Mealy (`load_enable`) outputs under various timing conditions.

```
`timescale 1ns / 1ps

module fsm_controller_tb;

    // Inputs
    reg clk;
```

```
    reg rst;
    reg start;
    reg ready;
    reg input_ready;

    // Outputs
    wire done_flag;
    wire load_enable;

    // Instantiate the FSM module
    fsm_controller uut (
        .clk(clk),
        .rst(rst),
        .start(start),
        .ready(ready),
        .input_ready(input_ready),
        .done_flag(done_flag),
        .load_enable(load_enable)
    );

    // Clock generation: 10ns period
    always #5 clk = ~clk;

    // Test sequence
    initial begin
        $display("Starting FSM simulation...");
        $dumpfile("fsm_controller_tb.vcd");  // For GTKWave or
           ModelSim
        $dumpvars(0, fsm_controller_tb);

        // Initialize inputs
        clk = 0;
        rst = 1;
        start = 0;
        ready = 0;
        input_ready = 0;

        // Reset pulse
        #10 rst = 0;
        #10;

```

```
47          // Stimulus: start signal
48          start = 1;
49          input_ready = 1;
50          #10 start = 0;
51
52          // FSM should enter LOAD state now
53          #20;
54
55          // FSM remains in LOAD if ready = 0
56          ready = 0;
57          #20;
58
59          // Trigger transition to DONE
60          ready = 1;
61          #10;
62
63          // FSM returns to IDLE
64          ready = 0;
65          input_ready = 0;
66          #20;
67
68          $display("Simulation complete.");
69          $finish;
70      end
71
72  endmodule
```

This example clearly shows how to build a finite state machine in Verilog using a two-process model: one for updating the state (synchronous block), and another for determining the next state (combinational block). It also illustrates the practical use of Moore and Mealy outputs within the same controller.

5.7 Example: Serial Data Receiver FSM

This section presents an FSM-based controller for a simple serial data receiver. The design detects a start bit, sequentially receives 8 data bits, and waits for a stop bit. The receiver assembles the data into a complete byte and signals readiness when the byte is valid.

5.7.1 States

The FSM has four operational states:

- **IDLE:** Waits for a falling edge (logic '0') indicating the start bit.
- **START:** Confirms the start bit is valid and prepares for data sampling.
- **RECEIVE:** Shifts in 8 serial bits, one per clock cycle.
- **STOP:** Verifies the stop bit (logic '1') and finalizes the received byte.

5.7.2 Implementation Steps

1. **Draw the FSM State Diagram:** Identify transitions based on input `rx` (serial line), counter values, and conditions.
2. **Define State Encoding:** Use binary or one-hot encoding depending on optimization goals.
3. **Write Verilog Code:** Implement the FSM with sequential logic, serial shift register, and output flag.
4. **Simulate the Design:** Apply a serial input waveform and verify data assembly and state transitions.
5. **Verify Functionality:** Check that the `data_valid` flag is asserted with a correctly received byte.

Design Notes

- Sampling should ideally occur at the midpoint of each bit time.
- A baud rate counter may be included for proper bit timing (not shown in simplified example).
- For robustness, noise filtering and oversampling can be added in practical designs.

5.7.3 Serial Data Receiver FSM: Verilog Design and Testbench

The following Verilog code describes a FSM that detects a start bit, receives 8 bits serially, and verifies a stop bit before asserting a valid data output. This simple receiver operates under the assumption that each bit occupies one clock cycle.

Verilog Implementation:

```
module serial_rx_fsm (
    input wire clk,
    input wire rst,
    input wire rx,
    output reg [7:0] data_out,
    output reg data_valid
);

    typedef enum logic [1:0] {
        IDLE    = 2'b00,
        START   = 2'b01,
        RECEIVE = 2'b10,
        STOP    = 2'b11
    } state_t;

    state_t state, next_state;
    reg [2:0] bit_cnt;
    reg [7:0] shift_reg;

    always @(posedge clk or posedge rst) begin
        if (rst)
            state <= IDLE;
        else
            state <= next_state;
    end

    always @(*) begin
        case (state)
            IDLE:    next_state = (~rx) ? START : IDLE;
            START:   next_state = RECEIVE;
            RECEIVE: next_state = (bit_cnt == 3'd7) ? STOP :
               RECEIVE;
            STOP:    next_state = IDLE;
            default: next_state = IDLE;
        endcase
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            bit_cnt     <= 0;
            shift_reg   <= 8'b0;
```

```
            data_out    <= 8'b0;
            data_valid  <= 1'b0;
        end else begin
            case (state)
                IDLE: begin
                    bit_cnt    <= 0;
                    data_valid <= 0;
                end
                START: bit_cnt <= 0;
                RECEIVE: begin
                    shift_reg <= {rx, shift_reg[7:1]};
                    bit_cnt   <= bit_cnt + 1;
                end
                STOP: begin
                    if (rx) begin
                        data_out   <= shift_reg;
                        data_valid <= 1;
                    end else
                        data_valid <= 0;
                end
            endcase
        end
    end

endmodule
```

Testbench:

The testbench sends a serial bitstream for the byte `8'b10101011` (transmitted LSB-first) and verifies whether the FSM correctly reconstructs the byte and asserts the `data_valid` signal.

```
module serial_rx_fsm_tb;

    reg clk, rst, rx;
    wire [7:0] data_out;
    wire data_valid;

    serial_rx_fsm uut (
        .clk(clk),
        .rst(rst),
        .rx(rx),
        .data_out(data_out),
```

```
        .data_valid(data_valid)
    );

    always #10 clk = ~clk;

    task send_bit(input bit val);
        begin
            rx = val;
            #20;
        end
    endtask

    task send_byte(input [7:0] data);
        integer i;
        begin
            send_bit(1'b0); // Start bit
            for (i = 0; i < 8; i = i + 1)
                send_bit(data[i]);
            send_bit(1'b1); // Stop bit
            #40;
        end
    endtask

    initial begin
        $dumpfile("serial_rx_fsm_tb.vcd");
        $dumpvars(0, serial_rx_fsm_tb);
        clk = 0; rst = 1; rx = 1;
        #25 rst = 0;
        #40 send_byte(8'b10101011);
        #100;
        $finish;
    end

endmodule
```

This simulation verifies that the FSM properly transitions through its states and outputs the correct byte along with a valid flag after receiving the stop bit.

5.8 Hierarchical FSM Design

In complex digital systems, a single monolithic FSM may become difficult to manage and scale. To improve modularity and maintainability, designers often adopt a hierarchical FSM (HFSM) structure, where the control logic is decomposed into smaller, more manageable sub-FSMs.

- **Modular Decomposition:** The overall system behavior is divided into functional blocks, each controlled by its own FSM. These sub-FSMs interact or are coordinated by a top-level FSM.
- **Control Abstraction:** Higher-level FSMs govern the activation, resetting, or mode selection of lower-level FSMs. This abstraction mirrors the top-down design methodology used in large systems.
- **Scalability and Reusability:** Sub-FSMs can be reused across multiple designs or instantiated multiple times, especially in protocols like UART, SPI, or memory interfaces.
- **Application Example:** In a UART receiver, one FSM may manage frame detection (start/stop bits), while another FSM handles data extraction and buffering.

Hierarchical FSMs are especially beneficial in protocol controllers, network stacks, SoC interfaces, and multi-mode digital blocks where layered control logic is required.

Hierarchical FSM: Modular Verilog Example and Testbench

For complex systems, hierarchical FSM design improves clarity and reuse by breaking control logic into smaller FSM modules. This example demonstrates a two-level FSM structure:

- A **top-level FSM** that controls the system states: `IDLE`, `ACTIVE`, and `WAIT`.
- A **sub-FSM** that counts from 0 to 3 while in the `ACTIVE` state.

Sub-FSM (Counter FSM):

```
module sub_counter_fsm (
    input  wire clk,
    input  wire rst,
    input  wire enable,
    output reg  done
);
```

```
    reg [1:0] count;
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            count <= 0;
            done  <= 0;
        end else if (enable) begin
            if (count == 2'd3) begin
                done <= 1;
                count <= 0;
            end else begin
                count <= count + 1;
                done <= 0;
            end
        end else begin
            count <= 0;
            done <= 0;
        end
    end
endmodule
```

Top-Level FSM:

```
module top_controller_fsm (
    input  wire clk,
    input  wire rst,
    input  wire start,
    output wire done
);
    typedef enum logic [1:0] {
        IDLE   = 2'b00,
        ACTIVE = 2'b01,
        WAIT   = 2'b10
    } state_t;

    state_t state, next_state;
    wire sub_done;

    sub_counter_fsm u_sub (
        .clk(clk),
        .rst(rst),
        .enable(state == ACTIVE),
        .done(sub_done)
```

```
    );

    always @(posedge clk or posedge rst)
        if (rst) state <= IDLE;
        else     state <= next_state;

    always @(*) begin
        case (state)
            IDLE:   next_state = start ? ACTIVE : IDLE;
            ACTIVE: next_state = sub_done ? WAIT : ACTIVE;
            WAIT:   next_state = IDLE;
            default: next_state = IDLE;
        endcase
    end

    assign done = (state == WAIT);
endmodule
```

Testbench:

```
module top_controller_fsm_tb;

    reg clk, rst, start;
    wire done;

    top_controller_fsm uut (
        .clk(clk),
        .rst(rst),
        .start(start),
        .done(done)
    );

    always #10 clk = ~clk;

    initial begin
        $dumpfile("top_controller_fsm_tb.vcd");
        $dumpvars(0, top_controller_fsm_tb);

        clk = 0; rst = 1; start = 0;
        #25 rst = 0;

        #20 start = 1;
```

```
23          #20 start = 0;
24
25          #200;
26          start = 1;
27          #20 start = 0;
28
29          #200;
30          $finish;
31      end
32
33  endmodule
```

The simulation demonstrates how the top FSM activates the sub-FSM, which asserts `done` after counting to 3. The top FSM then transitions through `WAIT` before returning to `IDLE`.

5.9 FSM as a Control Unit

FSMs play a crucial role as control units in digital systems. They orchestrate the sequence of operations in datapath components by asserting control signals at appropriate times. Instead of performing computations directly, FSMs coordinate how data flows through the system and when functional blocks should be activated.

- **Instruction Decoding in CPUs:** FSMs decode instruction opcodes and generate timing and control signals to enable registers, ALUs, memory reads/writes, and branching logic.
- **Step Control in ALUs:** In multi-cycle arithmetic operations (e.g., multiplication or division), FSMs manage the progression of control steps such as shifting, accumulating, or looping until completion.
- **Packet Parsing in Network Interfaces:** FSMs are used to detect packet headers, extract fields like destination address or payload, and route packets accordingly in network routers and protocol processors.

FSMs as control units enable deterministic behavior, efficient sequencing, and tight synchronization between control logic and datapath activity in both synchronous and asynchronous digital designs.

FSM as a Control Unit: Datapath + FSM Integration with Testbench

FSMs can act as control units that direct operations in a datapath by sequencing control signals. The following example demonstrates how an FSM controls a basic arithmetic datapath to perform addition in stages.

Verilog Implementation: ALU Controller with FSM

This FSM-based controller performs the following sequence:

1. `LOAD`: Capture inputs into internal registers.
2. `ADD`: Perform addition.
3. `STORE`: Output the result and assert `done`.

```
module alu_controller (
    input wire clk,
    input wire rst,
    input wire start,
    input wire [7:0] in_a,
    input wire [7:0] in_b,
    output reg [7:0] result,
    output reg done
);
    typedef enum logic [1:0] {
        IDLE  = 2'b00,
        LOAD  = 2'b01,
        ADD   = 2'b10,
        STORE = 2'b11
    } state_t;

    state_t state, next_state;
    reg [7:0] reg_a, reg_b, reg_sum;

    always @(posedge clk or posedge rst)
        if (rst) state <= IDLE;
        else     state <= next_state;

    always @(*) begin
        case (state)
            IDLE:  next_state = start ? LOAD : IDLE;
            LOAD:  next_state = ADD;
```

```
            ADD:   next_state = STORE;
            STORE: next_state = IDLE;
            default: next_state = IDLE;
        endcase
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            reg_a <= 0; reg_b <= 0; reg_sum <= 0;
            result <= 0; done <= 0;
        end else begin
            case (state)
                IDLE:   done <= 0;
                LOAD:   begin reg_a <= in_a; reg_b <= in_b; end
                ADD:    reg_sum <= reg_a + reg_b;
                STORE:  begin result <= reg_sum; done <= 1; end
            endcase
        end
    end
endmodule
```

Testbench: Simulation of the FSM and Datapath Operation

The following testbench simulates the FSM by applying inputs `in_a = 15` and `in_b = 27`, and verifies the result output.

```
module alu_controller_tb;

    reg clk, rst, start;
    reg [7:0] in_a, in_b;
    wire [7:0] result;
    wire done;

    alu_controller uut (
        .clk(clk), .rst(rst), .start(start),
        .in_a(in_a), .in_b(in_b),
        .result(result), .done(done)
    );

    always #10 clk = ~clk;  // 50 MHz clock

    initial begin
        $dumpfile("alu_controller_tb.vcd");
```

```
18          $dumpvars(0, alu_controller_tb);
19
20          clk = 0; rst = 1; start = 0;
21          in_a = 0; in_b = 0;
22
23          #25 rst = 0;
24          #20 in_a = 8'd15; in_b = 8'd27; start = 1;
25          #20 start = 0;
26
27          wait (done);
28          #20;
29          $display("A = %d, B = %d, Result = %d", in_a, in_b,
               result);
30
31          #20 $finish;
32      end
33  endmodule
```

This example illustrates the use of an FSM as a sequencer for a datapath, demonstrating clear separation of control and data logic. The simulation confirms correct control flow and result output.

5.10 Debugging FSMs

Debugging FSMs is a crucial step in both simulation and hardware implementation phases. FSM bugs are often difficult to detect as they involve multiple interacting conditions like input combinations, clock edges, and sequential dependencies. A systematic debugging approach can help trace and fix incorrect behaviors effectively.

Common FSM Debug Issues

- **Unexpected State Transitions:** Caused by incorrect combinational logic in the next-state block or incorrect input handling.
- **FSM Stuck in a State:** May happen due to missing transitions, improper reset, or uninitialized variables.
- **Race Conditions:** Occur when multiple signals change simultaneously without proper synchronization.

Debugging Techniques

- **Monitor State Transitions:**
  - Use waveform viewers (e.g., GTKWave, ModelSim) to plot `state` and `next_state` over time.
  - Add `$display("State = %b", state);` to monitor FSM transitions at each clock cycle during simulation.
- **Add One-Hot Debug Outputs:**
  - For one-hot FSMs, assign each state a dedicated debug bit.
  - Connect the state vector or specific state flags to FPGA output pins or LEDs for live debugging.
- **Use Integrated Logic Analyzer (ILA):**
  - FPGA vendors like Xilinx provide ILA cores to probe internal FSM signals (e.g., state registers, inputs, outputs).
  - ILA captures FSM activity in real time, supports triggering on specific states or transitions, and eliminates the need for additional I/O pins.
- **Check Reset Behavior:**
  - Ensure FSM resets to a known initial state and transitions only after reset is deasserted.
  - Use assertions in testbenches to confirm proper reset and idle behavior.
- **Create Self-Checking Testbenches:**
  - Add assertions or `if` conditions in the testbench to detect invalid states or transitions.
  - Use coverage tools to ensure all transitions and states have been exercised during simulation.
- **Use State Encodings that Simplify Debugging:**
  - One-hot and gray encodings are easier to debug in waveform and hardware compared to binary encoding.
  - Add symbolic names to state values using `parameter` or `enum` declarations to improve readability.

Recommendations

- Always verify your FSM thoroughly in simulation before synthesis.
- Instrument internal FSM signals for real-time observation in hardware, especially during system integration.
- Document all states and transitions to aid in both manual tracing and automated verification.

FSM debugging is both an art and a science—combining simulation discipline, tool support, and architectural clarity yields robust and predictable state-based designs.

Table 5.4 provides a comparative overview of various FSM debugging tools and techniques, highlighting their applicability in simulation environments versus real FPGA hardware for effective verification and analysis.

Table 5.4: Comparison of FSM debugging tools in simulation and hardware

Technique / Tool	**Simulation Environment**	**Hardware Environment (FPGA)**
Waveform Viewer	Use tools like GTKWave or ModelSim to visualize `state`, `next_state`, and transitions over time	Not available; requires signal export via debug cores
`$display`, `$monitor`	Textual printouts for FSM activity during simulation	Not synthesizable; useful only in testbenches
Assertions / Coverage	Ensure legal transitions and full FSM traversal	Limited to simulation or formal tools
One-Hot Debug Outputs	Represent each FSM state as a bit for simplified debugging in waveforms	Can be mapped to LEDs, GPIOs, or debug probes for physical observation
Integrated Logic Analyzer (ILA)	Not applicable	Real-time internal signal capture; supports trigger and buffer logic for FSM visualization
Signal Tapping / Embedded Logic Analyzers	Not needed; simulation provides full access	Tools like Vivado ILA or SignalTap (Intel) allow observing FSM signals post-synthesis
Parameterized FSM Encoding	Improves symbolic naming and traceability in simulations	Enables easier identification of state bits in hardware debug tools

5.11 Common Design Mistakes

While designing FSMs, it is easy to introduce subtle yet critical errors that may only manifest during hardware testing or late simulation. Understanding and avoiding these

common mistakes can greatly improve the robustness, readability, and synthesizability of FSMs in Verilog.

- **Incomplete State Transition Definitions:**
  A frequent design error occurs when not all input conditions are handled for each FSM state. This results in inferred latches or unpredictable state retention behavior, especially when using `case` or `if-else` constructs without covering all logical branches. To prevent this, always fully specify transitions for all inputs in every state.

```
// BAD: Missing else or default
if (state == S1 && input)
  next_state = S2;
// No else defined
```

- **Missing Default Branches in Combinational Logic:**
  In FSM next-state logic or output assignments, missing a `default` case in a `case` statement can cause the synthesis tool to infer unintended latches, particularly when some combinations of state/input are not explicitly handled. Best practice is to include a `default:` clause that provides a safe fallback.

```
case (state)
  S0: if (x) next_state = S1;
  S1: next_state = S2;
  // Missing default
endcase
```

- **Using Blocking Assignments in Sequential Logic:**
  Verilog supports both blocking (`=`) and non-blocking (`<=`) assignments. In sequential `always @(posedge clk)` blocks, blocking assignments may cause simulation mismatches and unintended behaviors due to ordering issues. Always use non-blocking assignments for registers and sequential logic.

```
// BAD
always @(posedge clk) begin
  state = next_state; // Blocking
end
```

- **Mixing Moore and Mealy Logic Incorrectly:**
  In Moore FSMs, outputs depend only on the state, while in Mealy FSMs, outputs depend on both state and inputs. Inconsistent mixing — such as assigning outputs in both places — may introduce glitches or timing hazards. Decide on a model and separate logic clearly.

```
// Avoid mixing styles like this:
assign output = (state == S1) || input_signal; // Moore +
   Mealy
```

- **Improper Reset Behavior:**
  Failing to properly initialize FSM state registers leads to unpredictable behavior, especially after configuration or power-up. Use synchronous or asynchronous resets with a clearly defined reset state (typically IDLE). Verify reset functionality in both simulation and hardware.

```
always @(posedge clk or posedge rst) begin
  if (rst)
    state <= IDLE;
  else
    state <= next_state;
end
```

- **Undocumented or Cryptic State Encoding:**
  Using raw binary values without symbolic names makes the FSM difficult to maintain and debug. This also increases the chance of incorrect transitions during code edits. Use `parameter` or `typedef enum logic` (SystemVerilog) to label states clearly.

```
// Good practice
parameter IDLE = 2'b00, LOAD = 2'b01, DONE = 2'b10;
```

- **Combining Control and Datapath Logic Excessively:**
  Embedding complex datapath logic (e.g., ALU operations or memory access) directly into FSM state logic leads to bloated control blocks, harder debugging, and synthesis inefficiencies. Modularize FSMs to handle only control, and delegate operations to separate datapath modules or tasks.

```
// BAD: Overloading FSM with computation
always @(posedge clk) begin
  if (state == CALC)
    result <= a * b + c - d; // Should be in datapath
end
```

These mistakes can compromise the correctness and performance of FSM-based designs. Regular simulation, waveform inspection, and synthesis reports can help in catching these issues early.

Table 5.5 summarizes typical design mistakes encountered in FSM implementation, along with their observable symptoms and potential consequences, helping designers identify and resolve functional or synthesis-related issues early in the development process.

Table 5.5: Diagnostic table of common FSM design mistakes

Design Mistake	**Symptom During Simulation/Synthesis**	**Impact on Design / Consequences**
Incomplete state transition coverage	Unintended latches; warnings from synthesis tool	FSM may get stuck in undefined states or retain previous state erroneously
Missing `default` in `case` blocks	Partial or inconsistent behavior	Leads to synthesis of unintended logic or latch inference
Blocking (`=`) in sequential logic	Race conditions; incorrect simulation results	Functional mismatch between simulation and synthesized hardware
Mixed Moore/Mealy output logic	Glitches in outputs; unstable response	Output changes asynchronously; difficult to meet timing constraints
Improper reset handling	FSM does not start in known state	Startup behavior is unpredictable or fails in real hardware
Cryptic state encoding (no labels)	Hard-to-read waveforms and code	Difficult to debug and verify; increases maintenance effort
Datapath logic embedded in FSM	Large monolithic FSM blocks	Reduced readability; harder timing closure and slower synthesis
No symbolic state naming	Use of raw binary literals (e.g., `2'b01`)	Increases chance of incorrect transitions and debugging difficulty

5.12 Advanced FSM Topics

While basic FSMs manage simple sequential logic and control flow, real-world digital systems often require FSMs that can handle complex timing, parallel processes, and reliability mechanisms. This section introduces several advanced FSM techniques that extend traditional state machine capabilities.

5.12.1 FSM with Wait States

FSMs often require synchronization with slower external components or internal operations that take multiple clock cycles to complete. A wait state is an intermediate state where the FSM stalls until a specified condition is satisfied. This is particularly useful in memory interfaces, communication protocols, or inter-module handshakes.

- **Purpose:** To delay transitions until an enable signal, acknowledgment, or data-ready condition is asserted.
- **Example Use Case:** In a UART receiver FSM, the WAIT state holds the FSM until a full byte is received on the serial line.
- **Implementation Tip:** Use a conditional check in the WAIT state to determine when to proceed.

```
WAIT: if (data_ready) next_state = PROCESS;
      else next_state = WAIT;
```

5.12.2 FSM with Priority Arbitration

When multiple clients request access to a shared resource (such as memory, a bus, or a processor), FSMs can incorporate arbitration logic to grant access based on a defined policy. This avoids conflicts and ensures fairness or efficiency depending on system requirements.

- **Fixed Priority:** Always grants higher priority to one source (e.g., CPU over DMA).
- **Round-Robin Arbitration:** Rotates priority to prevent starvation of lower-priority modules.
- **Dynamic Arbitration:** Uses traffic patterns, urgency, or history to assign priority.
- **Example Use Case:** In a SoC, a bus arbiter FSM manages simultaneous requests from peripherals like Ethernet, UART, and Flash memory.

Design Tip: Implement arbitration logic as a separate FSM or a sub-FSM embedded within a control unit.

5.12.3 FSM with Timeout Counters

Timeout counters help detect unresponsive modules or failed handshaking. A counter is started in a given state, and if the FSM remains in that state beyond a safe duration, the counter expires and forces the FSM into a recovery or error-handling path.

- **Purpose:** Prevent the system from getting stuck in states due to hardware failures, invalid data, or protocol violations.
- **Use Cases:**

  - In I^2C or SPI protocols, where response from the slave might be delayed or missing.
  - In DMA engines, where a transfer may stall due to bus contention.
  - In handshake-based protocols between FSMs operating in different clock domains.

- **Implementation Strategy:**

  1. Declare a counter register.
  2. Increment it in the target state on every clock cycle.
  3. Compare the count with a timeout threshold.
  4. On timeout, transition to a safe state (e.g., ERROR, IDLE).

```
if (state == WAIT_ACK) begin
  if (ack_received)
    next_state = DONE;
  else if (timeout_counter > MAX_WAIT)
    next_state = ERROR;
end
```

Benefits of Advanced FSM Design

- Enhances robustness in unreliable or asynchronous systems.
- Enables FSM-based control over complex interactions in multiprocessor and multi-clock systems.
- Supports modular and hierarchical design when multiple FSMs cooperate (e.g., master/slave FSMs).

5.13 Summary

FSMs are essential for designing control logic in digital systems. They help describe how a system behaves over time based on inputs and previous states.

Verilog makes it easy to model FSMs using structures like `case`, `always` blocks, and clear state definitions. Designers can choose between Moore and Mealy machines, use different state encodings, and apply advanced features like wait states, arbitration, and timeout logic.

A well-designed FSM leads to:

- Clear and organized control logic
- Easier simulation and debugging
- Better reuse and maintenance

Understanding FSMs in Verilog helps you build reliable, scalable digital designs for real-world applications.

The laboratory exercises for Chapter 5: *Finite State Machines and Control* include Lab 12: FSM Design for a Serial Adder and Lab 13: Traffic Light Controller, both of which are provided in the Appendix section.

5.14 Exercises

1. **Sequence Detector – Moore FSM:** Design a Moore finite state machine that detects the bit sequence "1101" on a serial input stream. Implement it in Verilog using one-hot encoding.
2. **Sequence Detector – Mealy FSM:** Redesign the sequence detector from Exercise 1 using a Mealy model. Compare the output behavior and timing to the Moore version.
3. **State Transition Table:** Given a state diagram with 4 states and binary transitions, derive the corresponding state transition table and write the Verilog code for the FSM.
4. **FSM Simulation:** Develop a testbench to simulate an FSM with reset and clock inputs. Apply a stimulus sequence and verify correct state transitions and outputs.
5. **Elevator Controller (Mini-Project):** Design a simplified FSM for a 3-floor elevator system that handles requests, door open/close, and movement logic.
6. **Synchronous vs Asynchronous Reset:** Modify an FSM to include both synchronous and asynchronous reset options. Compare how each reset type affects simulation and behavior.
7. **FSM Optimization:** Given a state transition diagram with redundant states, optimize it by minimizing the number of states without changing the behavior. Rewrite the optimized FSM in Verilog.
8. **Binary to Gray Code Converter FSM (Advanced):** Implement a finite state machine that converts binary inputs to Gray code outputs, one bit at a time.

Bibliography

[1] C. Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes, 2004.

[2] M. Manohar and J. Bhasker, *Digital System Design Using FPGA: Implementation with Verilog and VHDL*. New York, NY, USA: McGraw-Hill, 2017.

[3] P. Minns and I. Elliott, *FSM-Based Digital Design Using Verilog HDL*. Chichester, U.K.: John Wiley & Sons, 2008.

[4] R. Merrick, *Getting Started with FPGAs: Digital Circuit Design, Verilog, and VHDL for Beginners*. San Francisco, CA: No Starch Press, 2023.

[5] S. Brown and Z. G. Vranesic, *Fundamentals of Digital Logic with Verilog Design*, 3rd ed. New York, NY, USA: McGraw-Hill, 2014.

[6] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[7] ChipVerify.com, "Verilog FSM," *ChipVerify Tutorials*. [Online]. Available: https://www.chipverify.com/verilog/verilog-fsm

[8] D. S. Cummings, "Finite State Machine Design & Synthesis using Verilog-2001," *SNUG Conference Presentation*, 2000. [Online]. Available: https://www.snug.org/papers

[9] FPGA4Student.com, "Full Verilog Code for Moore FSM Sequence Detector," *FPGA4Student*, 2017. [Online]. Available: https://www.fpga4student.com/2017/09/verilog-code-for-moore-fsm-sequence-detector.html

[10] Wikipedia contributors, "Finite-state machine," *Wikipedia*, last updated 2025. [Online]. Available: https://en.wikipedia.org/wiki/Finite-state_machine

Chapter 6

Design for Synthesis and Timing

Chapter Objectives

- Understand RTL design and the synthesis process for FPGA implementation.
- Apply timing concepts such as setup/hold time and critical path analysis.
- Optimize Verilog designs using pipelining, resource sharing, and timing constraints.

6.1 Introduction to RTL Design and Synthesis

RTL design is a key concept in digital system design. It describes how data moves between registers and how logic operations are performed on that data. RTL focuses on what happens on each clock cycle and how signals are processed and stored.

At this level, designs are written using HDLs such as Verilog or VHDL. These descriptions include:

- Registers to store data
- Combinational logic to process data
- Control logic to decide how and when data flows

Once the RTL code is written, it is passed to a synthesis tool (like Xilinx Vivado or Intel Quartus), which converts the HDL code into a gate-level netlist made of logic gates and flip-flops. This netlist can then be used to implement the design on an FPGA or ASIC.

Why RTL Design is Important

- It provides a clear structure for designing digital systems.

- It can be used for both simulation and hardware implementation.
- It helps manage complexity in large systems by using modular and reusable components.

Figure 6.1 illustrates the hierarchy of digital design abstraction, starting from RTL modeling, through gate-level logic, down to the transistor-level implementation, showing how high-level code is ultimately mapped to physical hardware. RTL design is the bridge between high-level design ideas and actual hardware. It is the standard approach for designing digital circuits that are fast, efficient, and ready for synthesis.

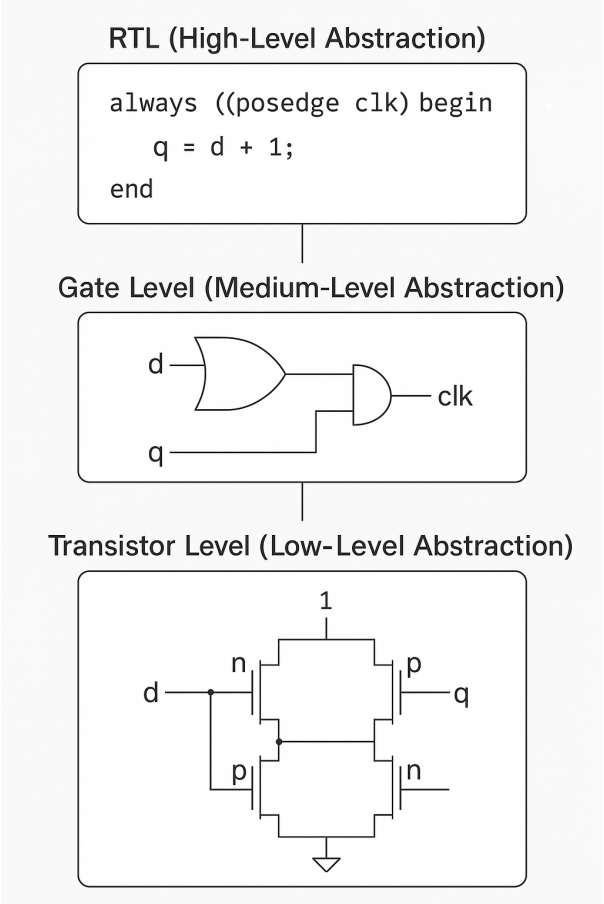


Figure 6.1: Three levels of design abstraction (register, gate, and transistor)

6.2 What is Synthesis?

Synthesis is a fundamental step in the digital design flow that translates high-level HDL code—such as Verilog or VHDL—into a lower-level representation of hardware structures. Specifically, synthesis tools convert RTL descriptions into a gate-level netlist composed of logic gates, flip-flops, multiplexers, and other digital primitives. This netlist forms the

blueprint for how the design will be physically implemented on the target device, such as an FPGA or ASIC.

At a high level, the synthesis process performs the following tasks:

- **Parsing and Elaboration:** The tool analyzes and flattens the HDL code, resolving all module hierarchies and parameters into a complete design structure.
- **Logic Optimization:** Redundant or functionally equivalent logic is simplified to reduce area and power consumption while maintaining correct behavior.
- **Technology Mapping:** The optimized design is then mapped onto the library of logic elements (e.g., look-up tables, D flip-flops, carry chains) available on the target FPGA or ASIC platform.
- **Constraint Handling:** User-defined constraints, such as clock frequencies, I/O standards, and timing exceptions, are applied during synthesis to guide the logic implementation and ensure timing goals are met.

Synthesis results in a gate-level netlist, typically described in an intermediate file format such as EDIF or Verilog, which is used in the downstream implementation stages—placement, routing, and bitstream generation.

Modern Electronic Design Automation (EDA) tools such as **Xilinx Vivado**, **Intel Quartus Prime**, and **Synopsys Design Compiler** not only perform synthesis but also provide comprehensive reports on:

- *Logic Utilization:* The number of logic elements, registers, block RAMs, and DSP slices used.
- *Timing Analysis:* Setup and hold time checks, clock domain crossing issues, and critical path estimation.
- *Power Estimation:* Dynamic and static power consumption based on switching activity and clock gating.
- *Design Warnings and Errors:* Feedback regarding coding practices, unused logic, inferred latches, or timing violations.

Understanding synthesis is crucial for writing efficient Verilog code that leads to optimal hardware implementation. Designers must adhere to synthesizable coding guidelines and apply proper timing constraints to ensure that the resulting hardware meets functional and performance requirements.

Figure **??**: A simplified flow diagram of the synthesis process in digital design. The process begins with high-level HDL code, which is then passed through synthesis tools

to generate a gate-level netlist. This netlist is subsequently mapped onto the physical resources of the target hardware device, such as an FPGA or ASIC.

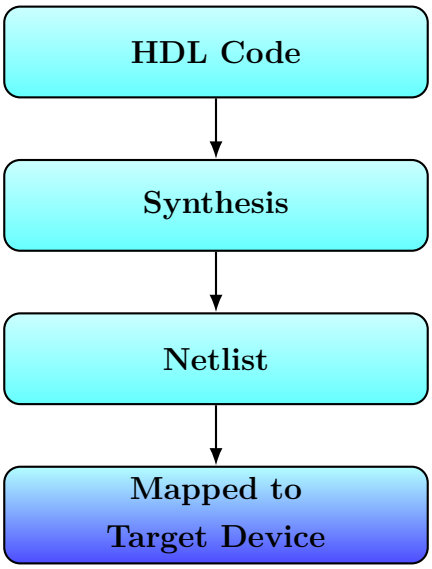


Figure 6.2: HDL-to-hardware flow

Synthesis is the bridge between high-level behavioral descriptions and low-level hardware realization. It is the key step that enables the transition from a conceptual RTL design to a physically deployable digital system on an FPGA or ASIC platform.

6.3 Synthesizable vs. Non-Synthesizable Code

When writing Verilog code intended for FPGA or ASIC implementation, it is critical to distinguish between synthesizable and non-synthesizable constructs. Synthesizable code is converted by synthesis tools into actual hardware elements such as flip-flops, multiplexers, logic gates, and memory blocks. Non-synthesizable code, on the other hand, is used for simulation, testbenches, or behavioral modeling, and cannot be implemented on physical hardware.

6.3.1 Synthesizable Constructs

Synthesizable constructs in Verilog are those that map directly to hardware resources and can be successfully interpreted by synthesis tools such as Xilinx Vivado, Intel Quartus, or Synopsys Design Compiler. Below are key categories of synthesizable Verilog code, along with brief explanations:

- **Always blocks with edge sensitivity:** These are used to model sequential logic, such as registers and flip-flops. The most common form is the use of `always @(posedge clk)`

or `@(negedge clk)` blocks, which trigger on a clock edge. This type of block is synthesizable because it maps to actual clocked flip-flops in hardware.

```
always @(posedge clk or posedge rst) begin
  if (rst)
    q <= 0;
  else
    q <= d;
end
```

- **Assign statements for combinational logic:** Continuous assignments using the `assign` keyword are synthesizable and used for modeling simple combinational logic such as gates and multiplexers.

```
assign y = (sel) ? a : b;
```

- **Simple for-loops and case statements:** Loops with static iteration ranges and case statements for conditionally selecting between fixed alternatives are commonly used in hardware design. They are synthesizable as long as the iteration bounds are constant and the logic within the loop or case is purely combinational or corresponds to hardware operations.

```
// Synthesis-friendly for-loop
for (i = 0; i < 8; i = i + 1)
  sum = sum + data[i];

// Synthesis-friendly case statement
always @(*) begin
  case (opcode)
    4'b0001: out = a + b;
    4'b0010: out = a - b;
    default: out = 0;
  endcase
end
```

These constructs are foundational in RTL design and are universally supported by synthesis tools. It is essential for FPGA and ASIC designers to focus on these constructs to ensure that their Verilog code can be successfully synthesized and mapped onto physical hardware.

6.3.2 Non-Synthesizable Constructs

Non-synthesizable constructs are Verilog statements and features that are intended only for simulation and testbench environments. These constructs do not correspond to actual hardware resources and therefore cannot be synthesized into an FPGA or ASIC design. While they are useful for verifying functionality, monitoring signals, and debugging during simulation, they must be avoided in the synthesizable portion of a design.

Below are common categories of non-synthesizable constructs:

- **initial blocks:** The `initial` block is executed only once at the beginning of simulation and is typically used for initializing variables, setting up stimulus, or defining temporary behavior. Since hardware cannot implement a "one-time" execution mechanism at time zero, `initial` blocks are not synthesizable.

```
initial begin
  count = 0;
  enable = 1;
end
```

- **Delays (#):** Delay statements such as `#10` introduce a time delay in simulation. These are useful for modeling timing behavior in testbenches, but they do not have a hardware equivalent. Real hardware timing is governed by clock cycles and physical delays in routing and logic, not by user-defined time steps.

```
#5 clk = ~clk;  // Simulation-only, toggles clock every 5
   time units
```

- **System tasks like `$display`, `$monitor`:** These tasks are used for printing messages, displaying variable values, or monitoring signal activity during simulation. They are invaluable for debugging and visualization, but have no place in synthesized hardware.

```
$display("Time = %0t, Data = %b", $time, data);
$monitor("Counter value: %d", counter);
```

- **File I/O operations:** Reading from or writing to external files (e.g., using `$readmemh`, `$fopen`, `$fscanf`) is often used in simulation for test vectors, data patterns, or logging. However, file systems do not exist in synthesized hardware, making these constructs non-synthesizable.

```
initial begin
  $readmemh("input_data.txt", memory_array);
end
```

Non-synthesizable constructs play an important role in verifying digital designs, but they must be strictly separated from the hardware description intended for synthesis. Most synthesis tools will ignore or flag these constructs as errors if included in the design hierarchy that is to be mapped onto hardware. Table 6.1 compares commonly used synthesizable and non-synthesizable Verilog constructs to help clarify their roles in hardware design versus simulation and testing.

Table 6.1: Comparison of synthesizable and non-synthesizable Verilog constructs

Synthesizable Constructs	**Non-Synthesizable Constructs**
`always @(posedge clk)` for sequential logic	`initial` blocks for testbench initialization
`always @(*)` for combinational logic	Delay statements like `#10`, `#100`
`assign` statements for continuous assignment	System tasks: `$display`, `$monitor`, `$stop`
Simple `for`-loops with static bounds	File I/O operations: `$readmemh`, `$fopen`, `$fscanf`
`case`, `if-else` for decision logic	Event-driven constructs: `wait`, `forever`, `fork-join`
Flip-flop/register-based storage	One-time execution or artificial constructs for simulation only
Synthesis-friendly FSM design	Behavioral modeling that does not reflect physical logic

6.4 Coding Guidelines for Synthesis

To ensure reliable and efficient translation of Verilog HDL into synthesizable hardware, designers must adhere to specific coding practices. These guidelines help prevent common synthesis errors, reduce unintended logic (such as latches), and improve the clarity and maintainability of the design. Below are essential synthesis-friendly coding rules:

- **Use non-blocking assignments (<=) for sequential logic:**
  Non-blocking assignments should be used inside clocked `always @(posedge clk)` blocks to model flip-flops and registers. This allows all assignments to occur concurrently on the clock edge, mimicking real hardware behavior and avoiding race conditions.

```
1  always @(posedge clk) begin
2    q <= d;  // Correct: non-blocking for sequential logic
3  end
```

- **Avoid mixed assignments in the same block:**
  Never mix blocking (`=`) and non-blocking (`<=`) assignments in the same `always` block. This can lead to unintended simulation-synthesis mismatches and synthesis ambiguity.

```
// Avoid this:
always @(posedge clk) begin
  a = b;      // Blocking
  c <= a;     // Non-blocking
end
```

  Instead, use either all blocking or all non-blocking assignments, depending on the type of logic (combinational or sequential).

- **Ensure all paths are defined in `case` and `if` statements:**
  For combinational logic described in `always @(*)` blocks, all possible input conditions must be covered. Failure to do so may lead to inferred latches, which are often undesired and can create unpredictable timing behavior.

```
// Correct: Full case statement with default
always @(*) begin
  case (sel)
    2'b00: y = a;
    2'b01: y = b;
    2'b10: y = c;
    default: y = 0; // Prevents latch
  endcase
end
```

- **Avoid latches by fully specifying logic conditions:**
  Latches are unintentionally inferred when outputs are not assigned under all conditions within a combinational block. To avoid this, always assign a default value or ensure every possible condition is explicitly handled.

```
// Poor style: may infer latch
always @(*) begin
  if (enable)
    y = a;
  // missing else -> latch inferred
end

// Recommended:
always @(*) begin
```

```
10      y = 0;              // Default assignment
11      if (enable)
12        y = a;
13    end
```

Following these guidelines helps ensure that Verilog designs are synthesis-friendly, functionally correct, and efficiently mapped to hardware. Adhering to consistent coding practices also enhances design portability across different synthesis tools and target devices.

6.5 Timing Concepts in Digital Design

Timing analysis is a critical aspect of digital circuit design, as it ensures correct data transfer between logic elements and proper operation of sequential circuits. Failure to meet timing requirements can lead to metastability, data corruption, or unpredictable system behavior. This section introduces the fundamental timing concepts used in both FPGA and ASIC design.

6.5.1 Propagation Delay

Propagation delay (t_{pd}) is the amount of time it takes for a signal to travel from the input to the output of a logic gate or combinational circuit. It is determined by the physical characteristics of the gate (e.g., gate capacitance, transistor switching time) and the fan-out load it drives.

- For a basic logic gate, propagation delay is typically measured from the 50% point of the input transition to the 50% point of the output transition.
- In a combinational path, the total propagation delay is the sum of delays of all the gates and interconnects from source to destination.

Impact: Excessive propagation delay may prevent signals from stabilizing before being latched by a flip-flop, resulting in timing violations.

6.5.2 Setup and Hold Time

Setup and hold times define the timing window during which data must remain stable with respect to a clock edge to be reliably sampled by a flip-flop or register.

- **Setup Time (t_{setup}):** The minimum amount of time the input data must be held stable *before* the active clock edge. Violating setup time can cause data to be incorrectly latched or result in metastability.

- **Hold Time (t_{hold}):** The minimum time the input data must remain stable *after* the clock edge. Violating hold time can lead to premature data changes that corrupt the register content.

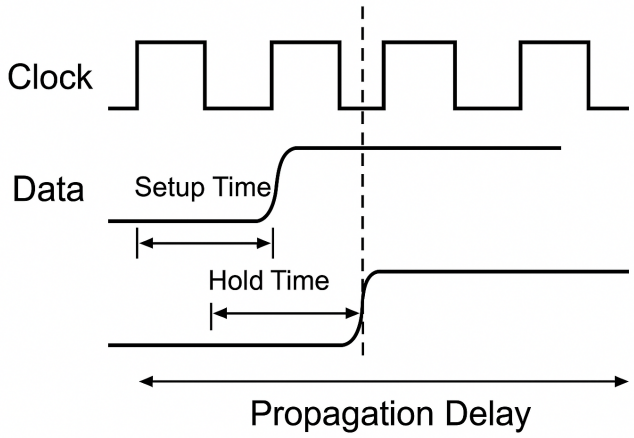


Figure 6.3: Timing diagram illustrating setup and hold time requirements

Figure 6.3 shows an example timing diagram depicting setup and hold time intervals relative to the clock edge.

Impact: Ensuring that setup and hold time constraints are met is essential for reliable operation of synchronous digital systems.

6.5.3 Clock Skew

Clock skew is the difference in arrival time of the clock signal at different elements of a digital circuit. Ideally, the clock signal should reach all flip-flops simultaneously, but in practice, varying delays in the clock distribution network cause discrepancies.

- **Positive skew:** Clock arrives later at the destination flip-flop than at the source flip-flop.
- **Negative skew:** Clock arrives earlier at the destination flip-flop than at the source flip-flop.

Impact:

- Positive skew may help fix setup violations but worsen hold time issues.
- Negative skew may assist with hold timing but can lead to setup time violations.

Design Considerations: Clock skew must be minimized through careful layout and clock tree balancing. In FPGA design, modern tools automatically perform clock routing optimization to manage skew and ensure timing closure.

6.6 Static Timing Analysis

Static Timing Analysis (STA) is a critical step in the digital design flow used to verify the timing behavior of synchronous circuits without requiring functional simulation. Unlike dynamic simulation, which checks behavior under specific input vectors, STA evaluates all possible timing paths across the design to ensure that they meet defined timing constraints.

STA inspects every register-to-register, input-to-register, and register-to-output path and computes delays using models for logic elements and interconnects. This analysis helps ensure that data is launched and captured correctly within the clock cycle boundaries, under worst-case conditions.

Key Concepts in STA

- **Setup Violations:** A setup violation occurs when data arrives at a register too late—after the required setup time window before the active clock edge. This results in the possibility that the flip-flop may latch incorrect or unstable data. STA flags any path where the data arrival time plus clock delay exceeds the clock period minus setup time.

- **Hold Violations:** A hold violation happens when data changes too soon after the clock edge—before the minimum hold time is satisfied. This may result in the flip-flop capturing new data prematurely, violating the stability window required immediately after the clock edge.

- **Slack:** Slack is the difference between the required time and the actual arrival time of a signal. Positive slack means the path meets timing, while negative slack indicates a timing violation. STA tools generate reports highlighting paths with the worst slack, which are critical for timing closure.

- **Clock Domains and Skew:** STA considers paths both within and across different clock domains. It also takes into account clock skew—the difference in arrival times of the clock at different points in the design—which can either help or hurt timing depending on its direction.

- **Constraints:** Timing constraints, typically written in Synopsys Design Constraints (SDC) format, define primary clocks, generated clocks, input/output delays, false paths, and multi-cycle paths. These constraints guide STA tools to analyze timing under realistic operating conditions.

Advantages of STA

- Fast and exhaustive—analyzes all possible timing paths without needing exhaustive test vectors.
- Pinpoints exact timing violations to help designers focus on critical paths.
- Independent of functional correctness—purely checks temporal behavior of the circuit.

Example Tools: Popular tools that perform STA include:

- *Xilinx Vivado Timing Analyzer*
- *Intel Quartus Prime TimeQuest*
- *Synopsys PrimeTime*

Static Timing Analysis is an indispensable technique for validating timing closure in digital systems. It helps ensure that designs not only work correctly but also operate reliably at the target clock frequency under all process, voltage, and temperature conditions.

6.7 Critical Path and Optimization

In digital circuit design, timing performance is fundamentally constrained by the longest delay between two clocked elements. This section explores the concept of the critical path and the optimization techniques used to reduce path delays and improve operating frequency.

6.7.1 Critical Path

The **critical path** is the longest timing path between two sequential elements (typically flip-flops or registers) in a digital circuit. It determines the minimum allowable clock period—and hence the maximum clock frequency—at which the system can reliably operate.

- It includes the propagation delay through combinational logic, plus clock-to-Q delay from the source register and setup time of the destination register.
- Any data that takes longer to traverse than the available clock period results in a setup violation, causing incorrect or unstable data to be latched.

- In large or complex designs, the critical path is a major design bottleneck that limits performance and must be carefully optimized.

Mathematically, the maximum frequency is given by:

$$f_{\max} = \frac{1}{T_{\text{critical}}} \tag{6.1}$$

where T_{critical} is the delay of the critical path.

6.7.2 Optimization Techniques

To meet timing constraints and improve throughput, designers can apply various optimization techniques to shorten critical path delays. Common strategies include:

- **Pipelining:** Pipelining introduces additional flip-flops along a long combinational path, dividing it into shorter stages. This reduces the logic depth between registers and allows the circuit to run at a higher clock frequency, though it increases latency.

```
stage1_out <= compute_stage1(in_data);
stage2_out <= compute_stage2(stage1_out);
```

- **Logic Restructuring:** Rewriting or reorganizing combinational logic to reduce the number of logic levels or eliminate redundant computations. This includes techniques like factoring expressions, simplifying logic equations, or using more efficient algorithms.

- **Retiming:** Retiming is a technique where flip-flops are repositioned across logic blocks (i.e., moved forward or backward in the data path) without changing the circuit's functionality. This technique helps balance the delay across stages and reduce the critical path.

- **Resource Sharing:** In designs where the same computation is performed by multiple functional units, combining them using multiplexers and control logic can reduce overall area and load, which may in turn reduce critical delays.

```
result <= (sel == 0) ? add_op(a, b) : sub_op(a, b);
```

These techniques are often used iteratively during synthesis and implementation. Timing reports from tools like Vivado or Quartus highlight critical paths, enabling designers to target specific modules or nets for improvement.

Identifying and optimizing the critical path is crucial for achieving high-performance digital systems. By applying methods like pipelining, retiming, and logic restructuring, designers can effectively push the limits of clock frequency while maintaining timing correctness.

6.8 Pipelining and Throughput Improvement

Pipelining is a fundamental technique in digital design used to improve the performance of sequential circuits by increasing throughput. By dividing a complex operation into smaller, manageable stages separated by registers, each stage can operate concurrently on different sets of data, allowing the circuit to produce results more frequently.

6.8.1 Concept of Pipelining

In pipelining, a long combinational logic operation is broken into multiple stages. Each stage performs a portion of the overall task and passes the intermediate result to the next stage through a register. This enables multiple data elements to be processed simultaneously in a staggered fashion, similar to an assembly line in manufacturing.

```
stage1 -> stage2 -> stage3
```

Each stage is clocked, and on every clock cycle, new input data enters the first stage while previously processed data moves to the next stage. Although the first output takes longer to appear (due to latency), subsequent outputs are generated at every clock cycle, significantly improving throughput.

6.8.2 Latency vs. Throughput

- **Latency:** Latency refers to the time it takes for a single data element to travel from the input to the output of the pipeline. In an N-stage pipeline, the latency is N clock cycles, assuming one register per stage.
- **Throughput:** Throughput is the rate at which the circuit produces results. In an ideal pipeline, after the initial latency, a new result is produced every clock cycle. Thus, the throughput is approximately one result per cycle.

Example: Consider a 3-stage pipeline. The first output will appear after 3 clock cycles, but from that point onward, one output will be produced each cycle, assuming continuous input data.

6.8.3 Verilog Example

The following Verilog snippet illustrates a simple two-stage pipeline:

```
1 reg [7:0] stage1, stage2;
2
3 always @(posedge clk)
4   stage1 <= input;
```

```
5
6 always @(posedge clk)
7   stage2 <= stage1;
```

In this example:

- On the first clock cycle, `input` is stored in `stage1`.
- On the second clock cycle, the value in `stage1` is transferred to `stage2`.

By adding more stages and balancing logic delays across them, designers can significantly increase the maximum operating frequency of a circuit, especially when the critical path delay is high.

Advantages of Pipelining

- Increases the overall throughput of the system.
- Enables higher clock frequencies by shortening the critical path per stage.
- Improves performance for repetitive operations like arithmetic, filtering, or data streaming.

Design Considerations

While pipelining improves throughput, it introduces additional latency and increases resource usage (due to additional registers). Designers must also ensure proper data synchronization and handle hazards such as pipeline stalls and data dependencies in more complex designs.

Pipelining is a powerful performance optimization technique widely used in digital systems, especially in signal processing, processors, and high-speed data paths.

6.9 Clock Domain Crossing

In modern digital systems, it is common to encounter multiple clock domains operating at different frequencies or phases. Transferring data between these asynchronous domains can lead to metastability, data corruption, or timing violations if not handled correctly. This necessitates the use of specialized Clock Domain Crossing (CDC) techniques to ensure reliable data transfer and synchronization.

Challenges of CDC

When a signal crosses from one clock domain to another without proper synchronization, the receiving register may capture the signal during its transition. This can cause the output to become metastable—a state where the register output is neither a logical "1" nor "0" for an extended time—potentially propagating incorrect values through the circuit.

Key problems include:

- Metastability due to violating setup/hold times
- Timing unpredictability across domains
- Loss or duplication of data without flow control

Common CDC Techniques

- **Dual Flip-Flop Synchronizers:** The most widely used method to transfer a single-bit control or flag signal between asynchronous domains. It involves passing the signal through two consecutive flip-flops clocked by the destination domain. This greatly reduces the probability of metastability reaching downstream logic.

```
always @(posedge clk_b)
  sync_1 <= async_signal;

always @(posedge clk_b)
  sync_2 <= sync_1;
```

- **Asynchronous FIFOs:** When multiple bits or streams of data must be transferred across clock domains, asynchronous First-In-First-Out (FIFO) buffers are commonly used. These structures use separate write and read clocks and incorporate Gray code pointers and status flags to manage full/empty conditions and avoid metastability.
  - Write domain inserts data and increments a write pointer.
  - Read domain retrieves data and updates a read pointer.
  - Synchronization logic tracks pointer updates across domains using Gray coding.
- **Handshake Protocols:** Handshake mechanisms ensure safe data transfer by requiring both sender and receiver to acknowledge transactions. Common in low-speed or control signal transfers, these protocols typically use request–acknowledge pairs and avoid timing assumptions.

- Sender asserts a `req` (request) signal.
- Receiver samples the request and returns an `ack` (acknowledge).
- Once acknowledgment is received, the sender can proceed.

Best Practices for CDC

- Avoid direct data transfers between unsynchronized clocks.
- Use synchronizers or CDC IP cores provided by FPGA vendors.
- Apply static CDC analysis tools (e.g., Vivado CDC Checker) to detect unsafe crossings.
- Design with robust metastability resolution time by ensuring flip-flops have sufficient settling time.

Clock Domain Crossing is a critical aspect of digital design whenever multiple clock regions coexist. Proper CDC design techniques ensure data integrity, system stability, and reliable hardware operation across asynchronous domains.

6.10 Reset Strategies

Reset logic is a crucial part of digital circuit design, ensuring that all registers and memory elements are initialized to a known state during power-up or recovery from faults. There are two common types of reset strategies: synchronous and asynchronous resets. Choosing the appropriate reset method depends on design requirements, clock behavior, and synthesis/implementation tool preferences.

6.10.1 Synchronous Reset

A **synchronous reset** is sampled only on the active edge of the clock. This means that the reset signal must be asserted prior to the clock edge and remain stable until it is registered. Synchronous resets offer predictable timing behavior and are fully controlled by the clock.

```
1 always @(posedge clk)
2   if (rst)
3     q <= 0;
4   else
5     q <= d;
```

Advantages:

- Reset behavior is synchronized with the clock, avoiding timing violations.
- Tools can optimize timing more effectively since the reset logic is part of the synchronous data path.
- Preferred in ASIC designs where timing closure is critical.

Limitations:

- Requires the clock to be active to perform a reset.
- Reset signal must meet setup and hold requirements like any other input.
- May introduce latency in systems that need immediate reset upon assertion.

6.10.2 Asynchronous Reset

An **asynchronous reset** affects the system immediately, regardless of the clock state. This type of reset is especially useful for initializing flip-flops at power-on or handling critical system errors.

```
always @(posedge clk or posedge rst)
  if (rst)
    q <= 0;
  else
    q <= d;
```

Advantages:

- Reset is independent of the clock—takes effect as soon as asserted.
- Useful during power-up initialization or system-level emergency shutdown.

Limitations:

- May lead to metastability if reset is deasserted near the clock edge.
- Requires proper synchronization of the reset signal, especially in multi-clock systems.
- Can complicate timing analysis and increase the risk of hold-time violations.

Best Practices for Reset Design

- Choose synchronous reset when possible for better timing control and compatibility with synthesis tools.
- Use asynchronous reset carefully—synchronize the deassertion using flip-flop chains to avoid metastability.
- Apply resets only where necessary. Avoid resetting every register unless explicitly required.
- Consider using global reset controllers or built-in FPGA primitives for large designs.

Table 6.2 compares synchronous and asynchronous reset strategies across several design attributes such as activation timing, metastability risk, and synthesis impact. Choosing the appropriate reset strategy is critical for ensuring correct circuit behavior during initialization and recovery.

Table 6.2: Comparison of synchronous and asynchronous reset strategies

Feature	**Synchronous Reset**	**Asynchronous Reset**
Reset Activation	Triggered on clock edge	Triggered immediately upon assertion
Reset Timing Control	Controlled by clock; predictable timing	Independent of clock; immediate effect
Design Complexity	Simpler for timing analysis	Requires careful handling to avoid metastability
Tool Optimization	Easier for synthesis tools to optimize	May limit optimization due to async logic paths
Power-up Behavior	Needs a valid clock for reset to occur	Effective even when clock is not running
Risk of Metastability	Low (as reset is clocked)	Higher (if deasserted near clock edge)
Recommended Use	Preferred in FPGA and ASIC designs with reliable clocks	Useful for fast system-wide reset or startup initialization

Reset strategies are essential for the reliable operation of digital circuits. Understanding when and how to apply synchronous or asynchronous resets ensures robust initialization and predictable design behavior, especially in complex and multi-clock systems.

6.11 Resource Optimization Techniques

In FPGA and ASIC design, efficient utilization of hardware resources is critical to meet area, power, and cost constraints. Designers must apply optimization techniques at the RTL level to reduce logic footprint, maximize reuse, and exploit the specialized blocks available in modern devices. The following are key strategies to optimize resource usage in Verilog-based hardware design.

- **Use of `generate` Statements:**
  The `generate` construct in Verilog allows for scalable, parameterized hardware instantiation. It is particularly useful when creating repetitive structures such as buses, arrays of registers, or replicated logic blocks. This helps avoid manual duplication and allows synthesis tools to better optimize the structure.

```
genvar i;
generate
  for (i = 0; i < 8; i = i + 1) begin : gen_block
    assign out[i] = in1[i] & in2[i];
  end
endgenerate
```

- **Control Logic Reuse:**
  When multiple operations require similar control signals or timing sequences, designers can share a single control unit across multiple datapaths or functional blocks. This reduces the need for separate state machines and minimizes redundant logic.
  - A shared FSM can orchestrate multiple operations.
  - Control lines can enable or disable different units based on operation type.

- **Multiplexed Datapaths:**
  Instead of duplicating arithmetic or logic units for every computation, resource sharing can be achieved by multiplexing inputs and selecting functions conditionally. This is especially useful in designs with limited resources or when area is a concern.

```
always @(*) begin
  case (op)
    2'b00: result = a + b;
    2'b01: result = a - b;
    2'b10: result = a & b;
    default: result = 0;
  endcase
end
```

Here, one arithmetic logic unit (ALU) performs multiple operations depending on the control signal `op`, reducing the total number of logic elements required.

- **Memory and DSP Slice Utilization:**
  Modern FPGAs include dedicated memory blocks (Block RAM or BRAM) and DSP slices optimized for multiplication and arithmetic operations. Efficiently mapping operations to these hardened blocks improves performance and frees up general-purpose logic.
  - Use `inferring` styles (e.g., for ROMs or RAMs) that synthesis tools can map to BRAM.
  - Utilize multiplication-friendly code structures so that DSP slices are automatically inferred.

```
// Inference of a single-port RAM
always @(posedge clk) begin
  if (we)
    mem[addr] <= din;
  dout <= mem[addr];
end
```

Proper use of attributes or coding templates specific to the FPGA vendor (e.g., Xilinx or Intel) can further enhance mapping efficiency to these resources.

Resource optimization techniques in Verilog design help:

- Reduce LUT and flip-flop usage.
- Minimize routing congestion and power consumption.
- Fit designs into smaller or lower-cost FPGAs.
- Improve performance by leveraging specialized on-chip resources.

In practice, designers must balance resource efficiency with performance, scalability, and timing closure. Most synthesis tools provide reports on resource usage and can identify redundant logic or underutilized blocks to guide optimization.

6.12 Synthesis Reports and Interpretation

After the synthesis stage is completed, FPGA and ASIC tools generate a series of reports that provide valuable insights into the quality and viability of the design. These reports help designers assess whether the hardware implementation meets performance, area, and resource utilization requirements. Interpreting synthesis reports effectively is critical for optimizing the design and ensuring successful downstream implementation.

Key Synthesis Report Types

- **Area Report: Resource Utilization**
  This report summarizes how much of the device's available hardware resources are consumed by the current design. It includes:

  - Look-Up Tables (LUTs)
  - Flip-Flops (FFs)
  - Block RAMs (BRAM)
  - DSP slices (for multiplication and arithmetic)
  - IO blocks and global buffers

  Understanding the area report helps identify whether the design will fit on the target FPGA or ASIC and reveals areas of excessive logic or inefficient coding.

- **Timing Report: Slack and Critical Paths**
  Timing reports evaluate whether the design meets its timing constraints. Two critical parameters are:

  - **Worst Negative Slack (WNS):** The most critical timing violation in the design. If WNS is negative, it means that the timing requirement is not met for at least one path.
  - **Total Negative Slack (TNS):** The cumulative timing violations across all failing paths in the design. A high TNS typically indicates widespread timing issues.

  Timing reports also highlight the critical paths that limit the maximum clock frequency and may need optimization.

- **Utilization Summary: Logic and Routing Distribution**
  This report provides a breakdown of how logic is distributed across the device, including:

  - Percentage usage of logic resources (LUTs, FFs, BRAMs, etc.)
  - Routing congestion and wire utilization
  - Clock resources and buffer usage

  Reviewing the utilization summary is essential for identifying design bottlenecks, over-congested areas, and imbalances in floorplanning, all of which can affect routing success and timing closure.

Common Report Files in FPGA Tools

- `*.synth.rpt`: Detailed synthesis log with RTL optimization summary.
- `utilization.rpt`: Tabulated breakdown of used and available hardware resources.
- `timing_summary.rpt`: List of paths violating setup/hold constraints.
- `clock_utilization.rpt`: Reports how clocks are assigned and loaded across the design.

Best Practices for Report Interpretation

- Always check for negative slack in timing reports and address critical paths.
- Monitor BRAM and DSP usage to ensure efficient mapping to specialized resources.
- Track utilization over design iterations to measure the impact of optimization.
- Use synthesis reports early in the design cycle to avoid surprises in the implementation stage.

Synthesis reports are more than just documentation—they are diagnostic tools that provide feedback on design quality, resource usage, and performance. Mastery of report interpretation enables informed decisions that lead to robust and optimized FPGA/ASIC implementations.

6.13 Toolchain-Specific Constraints

To achieve a functional and timing-accurate implementation of a digital design on an FPGA, designers must provide additional information to the synthesis and implementation tools in the form of constraints. These constraints inform the toolchain about the timing, pin assignments, voltage levels, and other design-specific requirements that are not captured by the HDL code alone.

In most FPGA workflows (such as those using Xilinx Vivado or Intel Quartus), constraints are written in standardized formats like XDC (Xilinx Design Constraints) or SDC (Synopsys Design Constraints).

6.13.1 Timing Constraints

Timing constraints define the required clock periods and relationships between different clocks and paths in the design. The most basic and essential timing constraint is the primary clock definition.

```
create_clock -period 10.000 -name clk [get_ports clk]
```

Explanation:

- `create_clock`: Defines a new clock for the design.
- `-period 10.000`: Specifies a clock period of 10 nanoseconds (i.e., 100 MHz frequency).
- `-name clk`: Names the clock "clk" for reference in other constraints.
- `[get_ports clk]`: Applies the clock to the port named `clk`.

This constraint is necessary for the tool to perform static timing analysis (STA), identify setup and hold violations, and optimize placement and routing accordingly.

6.13.2 I/O Constraints

I/O constraints define how external signals (such as reset, data, and clock lines) are connected to the physical pins of the FPGA and what electrical standards should be used. These are essential to ensure that the design can communicate correctly with other hardware components.

```
set_property PACKAGE_PIN W5 [get_ports reset]
set_property IOSTANDARD LVCMOS33 [get_ports reset]
```

Explanation:

- `set_property PACKAGE_PIN W5`: Assigns the signal `reset` to physical pin `W5` on the FPGA package.
- `set_property IOSTANDARD LVCMOS33`: Specifies the electrical standard for the `reset` signal as `LVCMOS33` (3.3V logic level).
- `[get_ports reset]`: Indicates that the constraint applies to the `reset` port in the HDL design.

These constraints are typically placed in an XDC file (for Vivado) or equivalent constraint file formats in other toolchains. They ensure that the synthesized design is physically realizable on the target device and meets interface requirements.

Best Practices

- Always define the primary clock accurately using `create_clock` to enable meaningful STA.
- Ensure pin assignments match the physical board schematic or reference design.
- Use named constraints files (e.g., `constraints.xdc`) to separate HDL from physical implementation details.
- Validate IOSTANDARD settings against voltage supply rails to avoid I/O damage.
- Use constraint checking tools (e.g., Vivado I/O Planning or Quartus Pin Planner) to detect conflicts or misassignments.

Toolchain-specific constraints are essential for bridging the gap between functional HDL code and physical hardware implementation. They guide synthesis and place-and-route tools to meet performance, reliability, and electrical compatibility requirements.

6.14 Design for Testability

Design for Testability (DFT) refers to the incorporation of specific hardware features and methodologies during the design phase to facilitate efficient post-fabrication testing, fault detection, and diagnosis. As digital systems grow in complexity, ensuring that every logic block and interconnection can be verified after manufacturing is essential for yield, reliability, and debugging.

DFT techniques improve the observability and controllability of internal signals, allowing automated test equipment (ATE) or in-system test controllers to detect defects and validate functionality. Common DFT methods include scan chains, boundary scan (JTAG), and logic built-in self-test (LBIST).

Scan Chains

Scan chains are the cornerstone of structural testing in ASICs and complex digital circuits. They involve replacing standard flip-flops with scan-enabled flip-flops that can form a serial shift register. This allows internal state registers to be loaded and observed directly from external test pins.

- During normal operation, scan flip-flops behave like regular registers.
- In test mode, they form a chain controlled by scan-in, scan-out, and scan-enable signals.

- Test vectors are shifted into the scan chain, applied to combinational logic, and the results are captured and shifted out for comparison.

Benefits:

- Enables deterministic testing of internal paths.
- Supports automatic test pattern generation (ATPG).
- Improves fault coverage with minimal functional impact.

Boundary Scan (JTAG)

Boundary scan, standardized as IEEE 1149.1 (commonly referred to as JTAG), is used to test interconnects between integrated circuits on a board without physical test probes. It involves adding a shift register stage (boundary scan cell) at each I/O pin, controlled through a standardized TAP (Test Access Port).

- Uses four or five pins: TCK (clock), TDI (data in), TDO (data out), TMS (mode select), and optionally TRST.
- Facilitates board-level testing, in-system programming, and debugging.
- Widely used in FPGAs and microcontrollers for configuration and monitoring.

Applications:

- Detecting soldering defects (open/short circuits).
- Verifying inter-chip connections.
- Programming flash or FPGA bitstreams.

Logic Built-In Self-Test (LBIST)

LBIST is an advanced DFT technique where the circuit includes hardware to generate test patterns and evaluate results internally, without the need for external test vectors. It uses pseudorandom pattern generators (e.g., LFSRs) and response analyzers (e.g., MISRs) to perform self-testing.

- Reduces dependency on external ATE equipment.
- Suitable for safety-critical or mission-critical applications.
- Can be triggered during power-on, idle periods, or manually.

Advantages:

- Enables at-speed testing of logic.
- Enhances reliability in field-deployed systems.
- Supports fault isolation and predictive maintenance.

Incorporating DFT strategies early in the design cycle enables high fault coverage, easier debugging, and improved manufacturability. Techniques like scan chains, boundary scan, and LBIST are essential in both ASIC and FPGA workflows, especially for high-reliability and high-volume production systems.

Proper DFT implementation ensures that modern digital systems are not only functionally correct but also verifiable and maintainable throughout their operational lifecycle.

6.15 Summary

RTL synthesis is the process of converting Verilog code into hardware-ready designs. It connects high-level modeling with physical implementation on FPGAs or ASICs.

This chapter covered key concepts that help create efficient, reliable digital systems:

- **Synthesizable Constructs:** Use Verilog features that map directly to hardware, avoiding simulation-only code.
- **Timing Basics:** Understand setup time, hold time, propagation delay, and clock skew to prevent timing failures.
- **Pipelining and Optimization:** Break long operations into stages and share hardware to boost performance and reduce area.
- **Static Timing Analysis (STA):** Automatically checks if the design meets timing requirements.
- **Clock Domain Crossing (CDC):** Use proper techniques like synchronizers and FIFOs to safely transfer data between clocks.
- **Reset and Resources:** Choose the right reset strategy and make efficient use of logic, memory, and DSP blocks.
- **Constraints and Reports:** Apply correct timing and I/O constraints, and read synthesis reports to improve your design.
- **Design for Testability (DFT):** Add scan chains, JTAG, and self-test logic to support manufacturing and debugging.

Understanding these principles helps designers create high-performance, resource-efficient, and testable digital systems.

The laboratory exercises for Chapter 6: *Design for Synthesis and Timing* reinforce Lab 9: Counters and Clock Divider Design, Lab 10: Display Counter on FPGA, and Lab 11: Multiplier Design, all of which are provided in the Appendix section.

6.16 Exercises

1. **Code Optimization for Synthesis:** Rewrite a Verilog module containing redundant logic and blocking assignments to use synthesizable constructs and improve resource efficiency.

2. **Timing Violation Detection:** Simulate a sequential design and identify any setup or hold time violations using timing analysis tools. Describe possible fixes.

3. **Design a Pipelined Adder:** Create a multi-stage pipelined 8-bit adder using registers to meet timing constraints. Analyze latency and throughput.

4. **Use of Constraints:** Define timing constraints (e.g., clock frequency, I/O delays) for a sample design using XDC (Xilinx Design Constraints) or SDC (Synopsys Design Constraints) format.

5. **Critical Path Identification:** Synthesize a design and report its critical path. Suggest structural or pipeline modifications to reduce path delay.

6. **Compare Clocking Strategies:** Implement a design using both synchronous and gated clocking approaches. Evaluate their synthesis results and timing behavior.

7. **Avoiding Latches:** Given a behavioral Verilog snippet with inferred latches, rewrite it using full case/if coverage to avoid unintended storage elements.

8. **Clock Domain Crossing (CDC):** Describe a safe CDC mechanism and implement a synchronizer for a signal crossing from one clock domain to another. Simulate metastability response.

Bibliography

[1] C. Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes, 2004.

[2] M. Manohar and J. Bhasker, *Digital System Design Using FPGA: Implementation with Verilog and VHDL*. New York, NY, USA: McGraw-Hill, 2017.

[3] R. Merrick, *Getting Started with FPGAs: Digital Circuit Design, Verilog, and VHDL for Beginners*. San Francisco, CA: No Starch Press, 2023.

[4] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, Pearson Education, 2003.

[5] ChipVerify.com, "Verilog Synthesis: RTL to Netlist and SDC Constraints," [Online]. Available: https://chipverify.com/verilog/verilog-synthesis

[6] V. Taraate, *Digital Logic Design Using Verilog: Coding and RTL Synthesis*, Section II, Chaps. 10–12. Springer, 2016.

[7] Xilinx Inc., *Vivado Design Suite User Guide: Synthesis (UG901)*, Version 2022.2, Oct. 2022. [Online]. Available: https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug901-vivado-synthesis.pdf

[8] Intel Corporation, "Quartus Prime User Guide: Enabling Timing-Driven Synthesis," Oct. 22, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683472

[9] Wikipedia contributors, "Static Timing Analysis," *Wikipedia*, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Static_timing_analysis

[10] Wikipedia contributors, "Timing Closure (VLSI)," *Wikipedia*, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Timing_closure

[11] Wikipedia contributors, "Delay Calculation," *Wikipedia*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Delay_calculation

Chapter 7

FPGA Architecture and Resources

Chapter Objectives

- Learn the basic components of FPGA architecture.
- Understand how to use and optimize FPGA resources.
- Use tools to plan and analyze FPGA designs.

7.1 Introduction to FPGA Architecture

FPGAs are reconfigurable digital devices that allow designers to implement custom hardware without fabricating a dedicated chip. Unlike CPUs, which operate sequentially, FPGAs support parallel execution by using a matrix of configurable logic blocks (CLBs), memory elements, and programmable interconnects.

CLBs include LUTs for logic and flip-flops for storage. These are connected by flexible routing networks, enabling complex digital circuits. FPGAs also contain on-chip Block RAM, DSP slices for arithmetic, I/O blocks, and clocking resources.

Modern FPGAs may integrate processors and IP cores, making them suitable for a wide range of applications such as signal processing, embedded systems, and hardware acceleration.

This chapter introduces the core components of FPGA architecture and how they support efficient, high-performance digital designs.

7.2 Basic Building Blocks of an FPGA

FPGAs are composed of a rich set of reconfigurable components that work together to implement custom digital circuits. These building blocks are tightly integrated and connected through a highly flexible routing architecture, allowing designers to tailor the

FPGA to a wide range of applications—from simple logic gates to complex signal processing pipelines.

FPGAs consist of several key components:

- **CLBs** – These form the fundamental processing elements of the FPGA. CLBs typically contain Look-Up Tables (LUTs), flip-flops (FFs), multiplexers, and carry chain logic. They are used to implement both combinational and sequential logic, including state machines, arithmetic units, and control logic.
- **Programmable Interconnects** – These are routing resources that connect CLBs and other logic elements across the FPGA fabric. They include switch matrices and routing channels that provide signal pathways between logic blocks. Efficient routing is critical for achieving timing closure and minimizing delay.
- **IOBs** – These blocks manage the interaction between the FPGA and external devices. IOBs support a variety of I/O standards (e.g., LVCMOS, LVDS, SSTL), programmable impedance control, drive strength, and input termination. They can be configured for single-ended or differential signaling, supporting high-speed communication.
- **Embedded Memory BRAM** – BRAMs are dedicated dual-port memory blocks distributed throughout the FPGA fabric. They are optimized for use in applications requiring high-speed storage such as FIFOs, buffers, and data caches. Some devices also support Error Correction Code (ECC) and initialization through configuration bitstreams.
- **DSP Slices** – DSP slices are specialized resources for implementing high-speed arithmetic operations such as multiplication, accumulation, and multiply-accumulate (MAC) functions. These slices are crucial in applications like FIR filters, FFTs, matrix operations, and machine learning accelerators.
- **Clocking Resources (PLL, MMCM)** – Modern FPGAs include sophisticated clock management units such as Phase-Locked Loops (PLLs) and Mixed-Mode Clock Managers (MMCMs). These components allow frequency synthesis, phase alignment, jitter filtering, and clock domain crossing—supporting both global and regional clock networks.

Together, these components form a highly configurable and scalable platform suitable for implementing complex digital systems. The integration of programmable logic, embedded memory, dedicated arithmetic resources, and flexible I/O interfaces allows FPGAs to serve as a versatile alternative to ASICs and microcontrollers in a broad spectrum of industries, including telecommunications, automotive, aerospace, healthcare, and AI.

As illustrated in Figure 7.1, the FPGA architecture typically consists of configurable logic blocks (CLBs), DSP slices, block RAM (BRAM), routing interconnects, I/O blocks, and a configuration interface. These elements are tightly interconnected to support reprogrammability and high-performance computation.

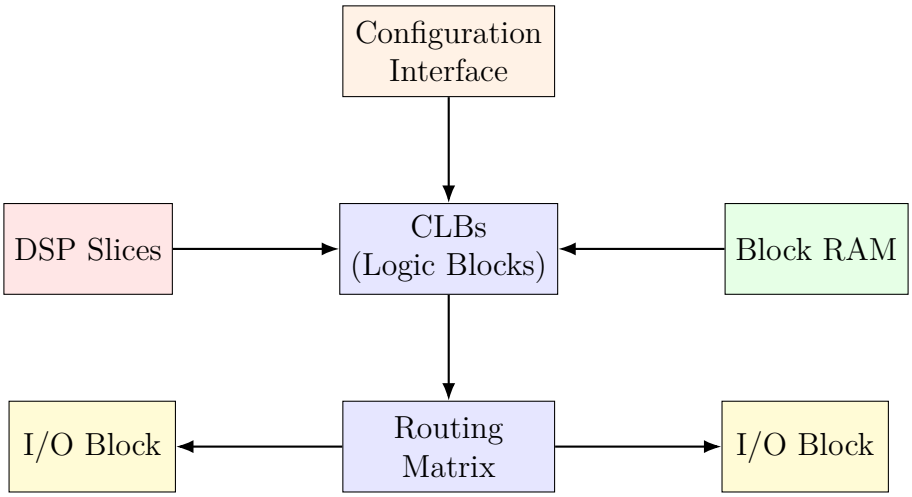


Figure 7.1: Basic FPGA architecture with logic, DSP, BRAM, and I/O blocks

7.3 Configurable Logic Blocks (CLBs)

CLBs form the core fabric of FPGAs. These programmable elements enable the implementation of both combinational and sequential logic functions. CLBs are distributed in a grid-like architecture throughout the FPGA and are connected via programmable interconnects.

7.3.1 Internal Structure of a CLB

Each CLB typically contains:

- One or more **slices**, each with Look-Up Tables (LUTs), Flip-Flops (FFs), multiplexers, and carry-chain logic.
- **LUTs** are SRAM-based components capable of implementing any n-input Boolean function.
- **Flip-Flops** provide storage for sequential logic, enabling designs such as counters and finite state machines (FSMs).
- **Carry-chain logic** accelerates arithmetic operations like addition and subtraction.

7.3.2 Slice Components

A slice is a sub-component of a CLB that contains:

- **LUTs:** Common sizes include 4-input, 6-input, and fracturable LUTs.
- **Flip-Flops:** Used for pipelining and edge-triggered logic.
- **MUXes:** Direct LUT outputs to either FFs or interconnects.
- **Carry chains:** Inter-slice paths for rapid arithmetic.

7.3.3 Arithmetic Operations and Carry Chains

Carry chains are optimized paths between adjacent slices that:

- Bypass general routing to minimize delay.
- Enable fast computation of wide adders and counters.

7.3.4 Verilog Example: Ripple Carry Adder Using CLBs

A 4-bit ripple carry adder written in Verilog demonstrates how CLBs are utilized in real-world FPGA design. This example highlights the use of basic combinational and sequential elements such as full adders and carry propagation, which are efficiently mapped onto LUTs and carry-chain structures within the CLBs.

A **ripple carry adder** computes the sum of two binary numbers by sequentially propagating the carry bit from the least significant bit (LSB) to the most significant bit (MSB). In FPGA design, carry chain logic embedded within *CLBs* enables faster carry propagation by minimizing routing delays, making ripple carry adders efficient for small-width additions.

The following Verilog module illustrates a 4-bit ripple carry adder using a `generate` block to instantiate full-adder behavior in a compact form. The `carry` signal propagates from bit 0 to bit 4, with each sum and carry computed combinationally.

Example Verilog Code: 4-bit Ripple Carry Adder in Verilog

```
module ripple_adder (
    input [3:0] A, B,
    output [3:0] SUM,
    output COUT
);
    wire [4:0] carry;
    assign carry[0] = 0;
    genvar i;
```

```
    generate
        for (i = 0; i < 4; i = i + 1) begin : adder
            assign {carry[i+1], SUM[i]} = A[i] + B[i] + carry[i];
        end
    endgenerate
    assign COUT = carry[4];
endmodule
```

This module takes two 4-bit input vectors `A` and `B` and produces a 4-bit sum `SUM` and a carry-out signal `COUT`. The carry bits are stored in an intermediate wire array `carry`, with `carry[0]` initialized to logic 0.

The `generate` block creates four full adders using structural Verilog. Each iteration computes a single-bit sum and carry using the formula:

$$\texttt{SUM[i]} = A[i] \oplus B[i] \oplus \texttt{carry[i]} \tag{7.1}$$

$$\texttt{carry[i+1]} = (A[i] \& B[i]) + ((A[i] \oplus B[i]) \& \texttt{carry[i]}) \tag{7.2}$$

This structure maps efficiently onto the CLBs due to the presence of fast carry chains, which are optimized for adder circuits in modern FPGAs.

Simulation Tip: When synthesizing this code in Vivado or Quartus, inspect the RTL schematic to confirm the use of carry-chain primitives (e.g., `CARRY4` in Xilinx or `ALTADD` in Intel FPGAs).

Application: Ripple carry adders like this form the basis of larger arithmetic units, such as ALUs and multipliers. Understanding their implementation at the CLB level is fundamental to digital system design using FPGAs.

7.3.5 Distributed RAM and Shift Registers

FPGAs offer flexible ways to implement on-chip memory and delay elements through their CLBs. Two notable features are Distributed RAM and Shift Register LUTs (SRLs), both implemented using the LUTs inside specific slice types (e.g., `SLICEM` in Xilinx FPGAs).

Distributed RAM

Distributed RAM refers to small memory blocks built from LUTs that are configured as static RAM cells. Instead of using BRAM, distributed RAM allows designers to create lightweight, high-speed memory structures near logic for low-latency access.

- Typical usage includes FIFOs, register files, and small lookup tables.
- Can be configured as single-port or dual-port RAM.

- Offers smaller capacity than BRAM but allows more localized memory allocation.

Example Verilog Code: 16x8 Distributed RAM using inferred logic.

```
module dist_ram (
    input clk,
    input [3:0] addr,
    input [7:0] din,
    input we,
    output reg [7:0] dout
);
    reg [7:0] ram [0:15];

    always @(posedge clk) begin
        if (we)
            ram[addr] <= din;
        dout <= ram[addr];
    end
endmodule
```

When synthesized, tools like Vivado or Quartus recognize this structure and map it to LUT-based distributed RAM, avoiding usage of scarce BRAM resources.

Shift Register LUTs

Shift Register LUTs (SRLs) allow LUTs to be configured as variable-length shift registers. These are highly efficient for delay lines, serial-in/parallel-out (SIPO), and FIR filter tap delays.

- Configurable shift lengths (up to 32 stages per LUT).
- Ideal for pipelines, timers, or tapped delay lines in DSP applications.

Example Verilog Code: 16-bit shift register using an always block.

```
module srl16 (
    input clk,
    input din,
    output dout
);
    reg [15:0] shift_reg = 16'b0;

    always @(posedge clk) begin
        shift_reg <= {shift_reg[14:0], din};
```

```
    end

    assign dout = shift_reg[15];
endmodule
```

Synthesis tools automatically map this structure to SRL primitives (e.g., `SRL16E` in Xilinx FPGAs), making it highly area-efficient compared to flip-flop-based chains.

Note

- Distributed RAM enables localized memory built from LUTs, offering lower access latency compared to block RAM for small data buffers.
- Shift Register LUTs provide compact, low-power shift registers, well-suited for pipelining and delay elements in signal processing applications.
- Both features are implemented within CLBs and give designers flexible, fine-grained control over storage and timing.

By leveraging these features, designers can significantly optimize logic utilization and data flow efficiency in their FPGA-based systems.

7.3.6 Device-Specific CLB Characteristics

While the basic concept of CLBs is consistent across FPGA vendors, the implementation details vary based on device family, target applications, and vendor-specific architecture. Understanding these device-specific characteristics is crucial for optimizing resource utilization, timing, and power efficiency in practical designs.

CLB and Slice Composition

Modern FPGAs (e.g., Xilinx 7-Series, Intel Cyclone/Stratix) define CLBs as arrays of logic slices. A typical CLB may contain:

- **Slices:** Each CLB may contain 2 to 4 slices depending on the family.
- **LUTs:** Each slice contains 4 to 8 LUTs capable of 6-input Boolean functions.
- **Flip-Flops:** Each LUT is usually paired with a D flip-flop, enabling combinational or sequential logic per bit.

In Xilinx FPGAs, slices are classified as:

- **SLICEL (Logic Slice):** Supports logic only.

- **SLICEM (Memory Slice):** Supports logic + distributed RAM or shift registers.

Intel (formerly Altera) devices such as Cyclone V and Stratix 10 use Adaptive Logic Modules (ALMs), which are functionally equivalent but structurally different.

CLB Resource Comparison (Xilinx 7-Series)

Table 7.1 compares the CLB resources—slices, LUTs, and flip-flops—available in selected Xilinx 7-Series FPGA families. These values reflect the logic capacity and scalability offered by each device class.

Table 7.1: CLB resources in selected Xilinx 7-series FPGAs.

Device	Slices	LUTs	Flip-Flops
Artix-7 7A35T	5,200	20,800	41,600
Kintex-7 7K70T	10,250	41,000	82,000
Virtex-7 7V585T	91,050	364,200	728,400

CLB Density and Performance

Higher-end devices offer significantly more CLBs per chip, supporting:

- Larger designs with more parallelism.
- Higher operating frequencies due to improved routing and pipelining.
- Advanced functionality such as partial reconfiguration and dynamic power gating.

Additionally, device families differ in maximum supported logic depth, carry chain length, and support for embedded features (e.g., DSP slices, BRAM, SERDES).

Optimizing for Device-Specific Resources

When targeting a specific FPGA:

- Use vendor tools (Vivado for Xilinx, Quartus for Intel) to analyze logic utilization and timing.
- Use `SLICEM` when shift registers or distributed memory is required.
- Enable logic replication, pipelining, and retiming to optimize for critical paths and flip-flop packing.

Device-specific CLB characteristics have a significant impact on design strategies and overall performance. Understanding the architectural distinctions across various FPGA families enables designers to make informed decisions when utilizing slices, LUTs, and

FFs according to their specific design requirements. It also aids in selecting the most suitable FPGA device based on application constraints such as cost, logic density, and performance goals. Moreover, awareness of these characteristics allows for more effective use of synthesis optimizations and floorplanning techniques, ultimately helping to extract the maximum functional and timing efficiency from the programmable fabric.

CLBs are the cornerstone of FPGA flexibility and efficiency. By understanding their internal architecture, designers can effectively map complex combinational logic functions using LUTs, implement pipelined and sequential circuits through the integrated flip-flops, and accelerate arithmetic computations using built-in carry chain logic. Furthermore, the capability of configuring LUTs as distributed RAM or shift registers allows localized memory resources to be embedded within the logic fabric, optimizing both speed and area utilization. These versatile features make CLBs indispensable in designing scalable and high-performance digital systems on FPGAs.

7.4 Programmable Interconnects

Programmable interconnects are a fundamental aspect of FPGA architecture, enabling the flexible and dynamic connectivity between different logic elements such as CLBs, embedded memories, input/output blocks, and DSP slices. The interconnect fabric forms the communication backbone of the FPGA, allowing designers to define how data flows between logic components.

Unlike fixed wiring in ASICs, FPGAs offer a matrix of switchable connections that are configured during the bitstream loading phase. The interconnect system is composed of switch matrices, multiplexers, and routing channels that span different regions of the FPGA.

Types of Interconnects

FPGA interconnects are broadly classified into the following types:

- **Global Routing:** These are long interconnect lines that span large distances across the chip. Global routing is used for signals that need to reach multiple clock regions or distant logic blocks. This includes clock distribution networks, resets, and control signals. Because of their wide reach, global routes are generally buffered and managed to reduce delay and skew.

- **Local Routing:** Local routing consists of short interconnect segments that connect nearby CLBs or slices within a tile or column. It provides fast and resource-efficient connections for fine-grained logic placement. This routing is typically managed by adjacent switch boxes and multiplexers within a logic region.

- **Direct Connections:** These are fixed, non-programmable paths between adjacent logic elements such as between LUTs and flip-flops within a slice or from a slice to a neighbor. They offer extremely low delay and are primarily used for carry chain logic, shift registers, and tightly coupled logic operations.

Impact on Performance

The choice and efficiency of routing directly affect the overall system performance, particularly in terms of:

- **Propagation Delay:** Routing delay is often the dominant contributor to critical path timing, especially in high-speed designs. Longer routes introduce higher delays due to wire capacitance and fanout loading.
- **Clock Skew and Jitter:** Improper or unbalanced routing of clock signals may result in clock skew, leading to setup or hold time violations.
- **Resource Utilization:** Inefficient routing can lead to congestion, which forces the place-and-route tool to use suboptimal paths, increasing power consumption and reducing available routing for other logic.

Routing Architecture Considerations

Modern FPGA devices implement hierarchical routing architectures where routing resources are arranged in grids, organized into:

- *Horizontal and Vertical Channels* for scalable communication.
- *Segmented Routing* to enable optimized switching and reuse.
- *Switch Boxes* and *Connection Boxes* that allow routing paths to be programmed at intersections between tiles and channels.

Tools such as Vivado (Xilinx) and Quartus (Intel) include powerful place-and-route engines that automatically select optimal routing paths based on timing, congestion, and resource constraints.

Programmable interconnects are essential for the reconfigurability of FPGAs. They enable scalable, customizable routing of signals between logic blocks while also presenting critical design trade-offs in terms of speed, area, and power. An in-depth understanding of routing types and their impact on performance is key to achieving optimized, high-speed FPGA designs.

7.5 Input/Output Blocks

Input/Output Blocks (IOBs) form the physical interface between the internal FPGA logic and the external world. These blocks are strategically located along the periphery of the FPGA die and are responsible for handling all digital signals entering or leaving the device. IOBs are highly configurable and play a critical role in enabling communication with other digital systems, such as microcontrollers, memory chips, sensors, and peripheral devices.

Core Features of IOBs

Each IOB typically includes:

- **Input Buffers (IBUFs)** – These receive signals from external sources and condition them for internal FPGA logic levels. They include features such as Schmitt triggers and hysteresis to improve noise immunity.
- **Output Drivers (OBUFs)** – These are used to drive signals from the FPGA onto external pins. They support various drive strengths and slew rate settings, which help to balance signal integrity with power consumption.
- **Bidirectional Buffers (IOBUF)** – These allow a single I/O pin to be used for both input and output operations, controlled by a tri-state enable signal. They are commonly used for data buses and memory interfaces.

Configurable Voltage Standards

IOBs support a wide range of programmable I/O standards to interface with different voltage levels and signaling environments. These standards include:

- **LVCMOS (Low Voltage CMOS)** – A common single-ended standard available in multiple voltage levels (e.g., 1.8V, 2.5V, 3.3V).
- **SSTL (Stub Series Terminated Logic)** – Typically used for interfacing with SDRAM and DDR memory due to its ability to support high-speed signaling with controlled impedance.
- **HSTL (High-Speed Transceiver Logic)** – Suited for fast data transfers in high-performance systems.
- **Differential Signaling Standards** – Including LVDS (Low Voltage Differential Signaling), which offer better noise immunity and higher data rates for serial communication.

Most FPGAs allow voltage banks to be independently powered and configured, meaning different I/O banks can operate with different voltage standards and logic types simultaneously. This feature supports system integration with diverse peripheral requirements.

Advanced IOB Features

Modern IOBs also include advanced capabilities such as:

- **On-chip termination** to match impedance and reduce reflection.
- **Programmable slew rate control** to manage electromagnetic interference (EMI).
- **Input delay and output delay** controls to align timing with external devices.
- **DDR register alignment** for capturing data on both rising and falling clock edges.

Example Use Case

In a typical design, IOBs might be configured as:

- LVCMOS 3.3V inputs for push-button signals.
- LVDS outputs for a high-speed DAC interface.
- Bidirectional IOBs with pull-ups for I^2C communication.

Toolchains such as Vivado (Xilinx) or Quartus (Intel) enable these configurations through constraints files (e.g., XDC, SDC), where I/O standards, voltage levels, and pin locations are defined.

IOBs are essential for reliable and flexible communication between the FPGA and its surrounding environment. Their configurability in terms of voltage standards, signaling types, and directionality allows FPGAs to integrate into a wide variety of electronic systems. A well-planned I/O configuration ensures robust signal integrity, compatibility with external components, and adherence to timing and power requirements.

7.6 Block RAM and Distributed RAM

FPGAs offer two primary forms of on-chip memory: BRAM and Distributed RAM. These memory types differ in size, implementation, access patterns, and use cases. Understanding the distinction between them allows designers to optimize for speed, area, and functional requirements in their systems.

7.6.1 Block RAM

BRAM consists of large, dedicated memory blocks embedded in the FPGA fabric. Unlike memory implemented using LUTs, BRAMs are true SRAM arrays designed to store a substantial amount of data. BRAMs are distributed across the FPGA in columns and are tightly coupled with dedicated read/write ports and control logic.

Key Features of BRAM:

- **Dual-Port Access:** Each BRAM block typically supports independent read and write operations on two ports, with individual clocks, addresses, data buses, and enable signals.
- **Configurable Width and Depth:** BRAMs can be configured in multiple width/depth combinations (e.g., 32K×1, 16K×2, 4K×8), allowing flexible usage depending on the design needs.
- **Initialization:** Most toolchains allow BRAMs to be initialized with predefined values via initialization files (e.g., .mem or .coe), useful for storing lookup tables, fonts, or firmware.
- **Synchronous Operation:** Read and write operations are clocked, allowing deterministic behavior and easy pipelining.
- **Optional Parity or ECC:** Some FPGAs support integrated error detection and correction to enhance reliability.

Example: A 256x8 single-port BRAM (Verilog-inferred)

```
module simple_bram (
    input clk,
    input [7:0] addr,
    input [7:0] din,
    input we,
    output reg [7:0] dout
);
    reg [7:0] mem [0:255];

    always @(posedge clk) begin
        if (we)
            mem[addr] <= din;
        dout <= mem[addr];
    end
endmodule
```

Use Cases:

- Video frame buffers and image storage
- Large lookup tables or waveform memory
- Instruction and data caches
- FIFO and dual-clock domain buffering

7.6.2 Distributed RAM

Distributed RAM is constructed by configuring the LUTs in logic slices to act as small RAM blocks. This memory type is ideal for lightweight, localized storage needs that benefit from proximity to the surrounding logic.

Key Features of Distributed RAM:

- **Implemented using LUTs:** Utilizes SLICEM resources in Xilinx or ALMs in Intel devices.
- **Low Latency Access:** Since it resides near logic elements, distributed RAM offers faster access than BRAM.
- **Flexible Sizing:** Typical depth options include 16x1, 32x1, or larger arrays built from multiple LUTs.
- **Efficient for Small Memories:** Ideal for register files, coefficient tables, or fast cache buffers within control logic.

Example: 16x8 distributed RAM using inferred logic

```
module dist_ram (
    input clk,
    input [3:0] addr,
    input [7:0] din,
    input we,
    output reg [7:0] dout
);
    reg [7:0] ram [0:15];

    always @(posedge clk) begin
        if (we)
            ram[addr] <= din;
        dout <= ram[addr];
```

```
    end
endmodule
```

Comparison and Design Considerations

- **BRAM** is preferable for storing large datasets and centralized memory structures.
- **Distributed RAM** is optimal for compact and speed-critical memory needs close to logic.
- Designs can combine both to achieve area and performance balance.

Block RAM and Distributed RAM provide designers with flexible memory options tailored for varying capacity, latency, and locality needs. Understanding their trade-offs enables efficient memory partitioning and performance optimization in FPGA-based systems.

7.7 DSP Slices

DSP slices are specialized hardware blocks embedded within modern FPGAs to efficiently perform high-speed arithmetic computations. These slices eliminate the need to implement complex arithmetic functions using general-purpose logic, thereby reducing both area usage and power consumption while achieving high throughput. DSP slices are a critical component in signal processing applications such as audio/video processing, wireless communication, image filtering, and artificial intelligence.

Core Functions of DSP Slices

Each DSP slice typically integrates the following key arithmetic functions:

- **Multipliers:** Perform fast integer or fixed-point multiplication operations. Some DSP slices support 18x25, 27x18, or even 48x48-bit multipliers depending on the FPGA family.
- **Accumulators:** Allow the result of a multiplication or addition to be accumulated over time, a fundamental requirement in FIR filters and DSP pipelines.
- **Multiply-Accumulate (MAC) Units:** Execute operations of the form $Y = A \times B + C$ in a single clock cycle, which is central to digital filters, convolution, and neural network inference.

Architectural Overview

A typical DSP slice (e.g., Xilinx `DSP48E1` or Intel `DSP Block`) includes:

- Pre-adder, multiplier, and post-adder/subtractor
- Cascade paths for pipelining long filters or chained MAC operations
- Dedicated input registers for timing control and pipeline depth
- Control logic for signed/unsigned operations, rounding, and saturation

These features allow DSP slices to maintain high clock speeds and throughput even when handling complex dataflows.

Example: FIR Filter MAC Structure

A Finite Impulse Response (FIR) filter uses MAC operations to compute the output:

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k] \tag{7.3}$$

This operation can be implemented efficiently using cascaded DSP slices, with one slice per tap. The data stream and coefficient inputs are pipelined through the FPGA's fabric, making real-time filtering possible at high sample rates.

Verilog Example: DSP-based Multiply-Accumulate

Below is a simplified behavioral Verilog module modeling a DSP MAC unit:

```
module mac_unit (
    input clk,
    input signed [15:0] A, B,
    input signed [31:0] C,
    output reg signed [31:0] Y
);
    always @(posedge clk) begin
        Y <= (A * B) + C;
    end
endmodule
```

In synthesis, this construct is typically mapped to the FPGA's native DSP slice without the need for manual instantiation.

Application Domains

- **Digital Filters (FIR/IIR):** Used in communication systems and audio processing.
- **Fast Fourier Transforms (FFT):** Critical in spectrum analysis, image compression, and OFDM.
- **Matrix Multiplication:** Foundational to AI/ML inference, video processing, and scientific computation.
- **Control Systems and Robotics:** Real-time signal fusion and PID control.

Optimization Tips

- Use vendor IP cores (e.g., Xilinx FIR Compiler, Intel DSP Builder) to simplify development.
- Enable pipelining to achieve timing closure and meet throughput requirements.
- Use cascade connections between DSP slices to build scalable MAC pipelines.

DSP slices are powerful, dedicated hardware resources designed for arithmetic-intensive operations in FPGA-based systems. By leveraging these slices, designers can achieve significant performance gains while saving logic area and improving power efficiency. Their seamless integration into vendor toolchains and support for standard arithmetic operations make them a preferred choice in high-performance digital signal processing applications.

7.8 Clocking Resources

FPGAs provide a comprehensive suite of clocking resources that enable precise control over timing, frequency, and synchronization across the programmable logic fabric. These clocking components are essential for achieving high-speed and reliable operation in complex digital designs involving multiple clock domains and performance-critical datapaths.

Clock Generation and Management

Clocking in FPGAs involves both generation and distribution. To support diverse design requirements, modern FPGAs include advanced clock synthesis and conditioning blocks such as:

- **PLL:** PLLs generate new clock frequencies by locking onto a reference clock and producing one or more output clocks with defined multiplication or division factors.

They are commonly used to boost clock speeds, perform phase alignment, and regenerate clean clock signals from noisy inputs.

- **MMCM:** Exclusive to Xilinx architectures, MMCMs offer even more granular clock manipulation than PLLs. They support dynamic reconfiguration, jitter reduction, fine-grained phase shifting, duty cycle correction, and frequency division. MMCMs are used in systems requiring multiple clock domains or tight clock alignment.

- **Global and Regional Clock Buffers (BUFG/BUFR):** These buffers are used to distribute clock signals efficiently across the FPGA. `BUFG` drives global clock networks that span the entire chip, while `BUFR` and `BUFH` provide lower-skew local distribution within a specific region. Selecting the appropriate buffer ensures optimal resource usage and timing performance.

Clock Domains and Synchronization

FPGA designs often incorporate multiple clock domains operating at different frequencies or phases. Proper synchronization across these domains is critical to avoid timing violations and metastability. Techniques such as dual-rank synchronizers, asynchronous FIFOs, and proper CDC (Clock Domain Crossing) constraints are employed to ensure data integrity.

Clocking Use Cases

- **Frequency Scaling:** Adjusting clock speeds for performance or power optimization.

- **Interface Timing:** Aligning internal clocks with external interfaces such as ADCs, memory, or high-speed serial links.

- **Power Domains:** Isolating or gating clock signals to save dynamic power.

- **Timing Closure:** Distributing clean and low-skew clocks to meet setup and hold constraints.

Design Considerations

Designers must consider:

- Proper placement of clock sources and buffers to reduce skew.

- Use of dedicated clock routing resources.

- Defining timing constraints (e.g., `create_clock`, `set_false_path`) to guide synthesis and place-and-route tools.

Clocking resources such as PLLs, MMCMs, and BUFG/BUFR buffers form the backbone of FPGA timing infrastructure. They enable reliable high-frequency operation, flexible clock manipulation, and precise synchronization across clock domains. Understanding how to leverage these components is essential for building efficient and high-performance digital systems on FPGAs.

7.9 Clock Distribution Network

The Clock Distribution Network is a critical component in FPGA architecture responsible for delivering clock signals uniformly and reliably to all sequential elements (e.g., flip-flops, memory blocks, DSP slices) across the device. Its primary goal is to ensure minimal skew and consistent timing, which are vital for synchronous operation and timing closure in complex designs.

Role of the Clock Network

In synchronous digital systems, the clock signal acts as the heartbeat that triggers data movement and state transitions. Any variation in clock arrival time (skew) across the chip can lead to setup and hold time violations, resulting in functional errors. The FPGA's clock distribution network is engineered to minimize such timing inconsistencies by employing dedicated routing resources and buffering strategies.

Clock Tree Architecture

Modern FPGAs implement hierarchical clock distribution systems organized in tree-like or mesh-like architectures. These include:

- **Global Clock Trees:** Driven by `BUFG` buffers, global clocks are routed through dedicated low-skew channels that span the entire FPGA. These are used for high-fanout clocks like system clocks or core clocks for processors and memory.
- **Regional Clock Networks:** Accessed through `BUFR` or `BUFH` buffers, regional clocks are distributed within specific clock regions (e.g., a set of rows or columns). These allow localized control and reduced power usage.
- **Local Clock Routing:** Inside logic slices and CLBs, local routing channels distribute the clock to nearby flip-flops and functional blocks. This level allows for precise control and timing optimization within a small area.

Design Guidelines

- Use `BUFG` for high-fanout, low-skew global clocks.
- Constrain and route high-speed clocks early in the design to avoid timing violations.
- For region-specific functions, utilize `BUFR` to save global resources and reduce congestion.
- Avoid excessive logic on clock paths; clocks should only be routed through dedicated clock buffers.

Skew and Jitter Management

Clock skew (difference in arrival times of the clock at various endpoints) and jitter (random variation in clock edge timing) can severely affect circuit reliability. FPGA clock networks manage these using:

- Balanced routing and matched path lengths.
- PLLs and MMCMs to regenerate clean clock signals.
- Placement-aware routing algorithms during place-and-route (PAR).

The Clock Distribution Network in an FPGA ensures reliable delivery of clock signals with minimal skew and jitter across all parts of the device. A well-designed clock tree is essential for meeting timing requirements, supporting multiple clock domains, and enabling high-speed synchronous operation in complex FPGA-based systems.

7.10 I/O Standards and Interfaces

FPGAs are equipped with highly configurable Input/Output (I/O) blocks that support a wide variety of electrical standards and communication protocols. This flexibility enables seamless integration of FPGAs with external components such as sensors, memory modules, analog-to-digital converters, and other digital systems. Configurable I/O banks allow the designer to tailor voltage levels, impedance, and signaling modes to match application-specific requirements.

Supported Electrical Standards

FPGA I/O blocks support both single-ended and differential signaling standards, catering to various voltage, speed, and noise tolerance needs:

- **Single-Ended Standards:**
  - **LVCMOS (Low-Voltage CMOS):** Widely used for general-purpose digital I/O. Supported in multiple voltage levels (e.g., 1.2V, 1.8V, 2.5V, 3.3V).
  - **TTL (Transistor-Transistor Logic):** Compatible with legacy 5V systems. Supported in a limited capacity due to modern device constraints.
- **Differential Standards:**
  - **LVDS (Low-Voltage Differential Signaling):** Offers high-speed, low-noise communication over twisted pair. Commonly used in camera interfaces, ADCs, and memory buses.
  - **HSTL (High-Speed Transceiver Logic):** Designed for fast communication in systems such as SDRAM and FPGAs-to-FPGAs connections.
  - **SSTL (Stub Series Terminated Logic):** Often used for DDR memory interfaces due to impedance control and signal integrity.

Each I/O bank on an FPGA can typically be configured independently, allowing the use of mixed signaling standards across the device.

Supported Communication Interfaces

Beyond electrical levels, FPGAs also support the implementation of common digital communication protocols. These protocols can be implemented via:

- **Hard IP Cores:** Pre-built logic blocks available in some FPGA families (e.g., DDR controllers, PCIe).
- **Soft IP Cores:** Synthesizable Verilog/VHDL modules instantiated by the designer.

Common supported protocols include:

- **UART (Universal Asynchronous Receiver/Transmitter):** Simple serial communication for debugging and low-speed control links.
- **SPI (Serial Peripheral Interface):** Synchronous protocol widely used for communication with flash memory, ADCs, and sensors.
- **I^2C (Inter-Integrated Circuit):** Two-wire serial bus used for low-speed peripheral communication in control applications.
- **DDR (Double Data Rate):** High-speed memory interface requiring precise I/O timing and differential signaling (via SSTL or HSTL).

Design Considerations

- Use vendor constraint files (e.g., XDC for Xilinx) to define I/O standards, drive strengths, and pin locations.
- Match voltage levels and impedance with external device specifications.
- Apply proper termination for high-speed differential interfaces.
- Use timing constraints and static timing analysis to ensure interface reliability.

FPGAs provide extensive support for industry-standard I/O signaling formats and communication protocols, enabling seamless integration with a wide range of external components. Designers can fine-tune electrical parameters and implement both standard and custom interfaces to meet the performance, power, and compatibility requirements of their target systems.

7.11 Specialized Blocks and IPs

Beyond the basic programmable fabric and logic elements, modern FPGAs incorporate a range of specialized hardware blocks and vendor-provided intellectual property (IP) cores. These elements enable high-performance and standards-compliant system integration without requiring the designer to develop complex functionality from scratch. Such capabilities significantly accelerate development cycles and enhance system capabilities in domains like networking, video processing, storage, and communication.

7.11.1 PCIe, Ethernet, HDMI Cores

FPGAs frequently include hardened or soft IP cores that support high-speed interface standards. These cores are either embedded directly into the silicon (hardened) or implemented in programmable logic (soft cores).

- **PCI Express (PCIe):**
  - Widely used for high-speed communication between FPGAs and CPUs, GPUs, or NVMe storage devices.
  - Most mid-to-high-end FPGAs include PCIe Gen2/Gen3/Gen4 endpoints as hardened IP blocks.
  - IP cores often include DMA engines, AXI interfaces, and configuration registers to streamline implementation.
- **Ethernet:**

  - Integrated 10/100/1000 Mbps, 2.5G, 10G, or 25G Ethernet MAC/PHY cores are available depending on device family.
  - Suitable for implementing networked embedded systems, industrial automation, or software-defined networking (SDN).
  - Support for time synchronization (IEEE 1588 PTP) is also often included in high-performance Ethernet cores.

- **HDMI (High-Definition Multimedia Interface):**
  - Used for video transmission in display and multimedia systems.
  - IP cores manage encoding/decoding, control signaling (DDC, CEC), and audio data.
  - Paired with TMDS drivers for differential signal transmission in video pipelines.

Design Integration and Licensing

Vendors such as Xilinx and Intel provide a rich catalog of IP cores via tools like:

- Xilinx Vivado IP Catalog (e.g., Aurora, Video PHY, AXI DMA)
- Intel Quartus IP Library (e.g., Transceiver Toolkit, DDR4 controller, JESD204B interface)

Some IPs are free, while others require licensing depending on their complexity and commercial usage. Most IP cores are configurable through GUI interfaces and are delivered with HDL wrappers and example designs.

Use Cases

- **PCIe:** FPGA co-processors, NVMe offloading, data center acceleration.
- **Ethernet:** Real-time data acquisition, industrial fieldbus interfaces, and IoT gateways.
- **HDMI:** Camera pipelines, display controllers, and FPGA-based media processing.

Specialized blocks and IP cores significantly expand the capability of FPGAs, enabling them to interface with high-speed peripherals and adhere to complex communication standards. By leveraging pre-validated IP for PCIe, Ethernet, and HDMI, designers can focus on core application logic while meeting performance and compliance requirements efficiently.

7.11.2 Embedded Soft/Hard Processors

Modern FPGAs support embedded processing through two main types of processor implementations: soft processors instantiated in the programmable fabric, and hard processors physically embedded into the silicon. These processing elements enable tight integration of hardware acceleration and software control, making FPGAs suitable for heterogeneous computing, real-time control, and system-on-chip (SoC) applications.

Soft Processors

Soft processors are implemented using the FPGA's logic fabric and can be configured as part of the synthesis process. These offer flexibility in customization and integration:

- **MicroBlaze (Xilinx):**
  - A 32-bit RISC soft processor core developed by Xilinx.
  - Configurable features include instruction/data caches, MMU, FPU, exception handling, and AXI interfaces.
  - Ideal for control-centric tasks, embedded software, and real-time operating systems (e.g., FreeRTOS).
- **Nios II (Intel):**
  - A configurable 32-bit RISC processor soft core provided by Intel (formerly Altera).
  - Supported through Platform Designer in Intel Quartus, with system interconnection over Avalon bus.
  - Suitable for cost-sensitive embedded applications with moderate performance requirements.

Soft processors offer a software-programmable path to manage peripherals, debug logic, and execute control algorithms without needing external MCUs. They are especially useful in tightly-coupled hardware/software systems.

Hard Processors

Hard processors are physically embedded in the FPGA silicon and offer higher performance and lower power consumption than their soft counterparts. They are commonly paired with programmable logic to form hybrid SoC platforms.

- **ARM Cortex-A9/A53 in Xilinx Zynq:**

  - The Xilinx Zynq-7000 and Zynq UltraScale+ families integrate dual-core or quad-core ARM Cortex processors alongside FPGA logic.
  - The processing system (PS) and programmable logic (PL) communicate via high-speed AXI interconnects.
  - Enables full Linux OS, real-time control, and acceleration offload in a single chip.

- **ARM Cortex-M in Intel SoC FPGAs:**
  - Intel SoC devices integrate Cortex-A9 or Cortex-M cores with FPGA fabric, enabling embedded system integration with peripherals and accelerators.

Applications of Embedded Processors

- **Control Systems:** Motor controllers, PID loops, and sensor fusion.
- **Embedded OS Execution:** Running Linux, FreeRTOS, or Baremetal applications.
- **Hardware/Software Co-Design:** Partitioning functionality between firmware and programmable logic.
- **Accelerated Systems:** Using FPGA logic for data-intensive compute (e.g., image processing) while control logic resides in software.

Embedded processors—whether soft cores like MicroBlaze and Nios II or hard cores like ARM Cortex—enable hybrid FPGA designs that combine the flexibility of programmable logic with the versatility of software programmability. This architecture supports powerful, tightly integrated SoC platforms suitable for control, signal processing, and AI workloads.

7.12 Resource Estimation and Floorplanning

Efficient FPGA design begins with accurate estimation and planning of available resources. As designs grow in complexity, it becomes critical to anticipate the consumption of logic elements, memory, DSP blocks, and routing capacity in order to ensure optimal performance, meet timing constraints, and avoid routing congestion. Floorplanning complements resource estimation by organizing physical layout and guiding placement tools to improve design quality.

Resource Estimation

Before implementation, designers must evaluate how much of each FPGA resource is likely to be consumed. Key resource types include:

- **Logic Cells (LUTs and FFs):**
  - Look-Up Tables (LUTs) implement combinational logic.
  - Flip-Flops (FFs) implement sequential logic and are often tightly coupled with LUTs in slices.
  - Estimation involves assessing logic depth, fan-out, and pipeline stages.
- **Memory (BRAM and Distributed RAM):**
  - Used for buffers, FIFOs, LUTs, and on-chip storage.
  - Estimated based on data width, depth, dual/single port requirements.
  - Tools like synthesis reports and IP configuration GUIs help predict usage.
- **Arithmetic (DSP Slices):**
  - Required for multiply-accumulate, filtering, FFTs, and AI operations.
  - Some tools provide macro usage estimations from arithmetic-heavy Verilog modules.
- **Routing and Timing:**
  - Even if logic fits, routing congestion or long delays may break timing.
  - High-fanout signals, long buses, or poor modularity increase pressure on routing resources.
  - Static Timing Analysis (STA) tools reveal critical paths and slack.

Floorplanning

Floorplanning is the process of assigning logic regions and I/O locations before place-and-route begins. Effective floorplanning results in:

- **Improved Timing Performance:** Reduces wire lengths and improves signal timing margins.
- **Reduced Routing Congestion:** Avoids overloading routing channels in dense regions.

- **Predictable Design Layout:** Helpful in partially reconfigurable or safety-critical systems.
- **Optimized Power Usage:** Closer placement of logic blocks minimizes switching capacitance.

Tools and Techniques

- **Synthesis Reports:** Tools like Vivado and Quartus provide logic and memory utilization summaries.
- **Floorplanning GUI:** Interactive floorplanners allow manual region definition and constraint assignment.
- **XDC/SDC Constraints:** Used to fix logic regions, guide routing, and define timing budgets.
- **Incremental Compilation:** Enables reuse of floorplan-aware modules to reduce build time and improve repeatability.

Best Practices

- Break large designs into smaller, modular RTL blocks to enable easier analysis.
- Use resource-aware IP cores and analyze post-synthesis resource usage early.
- Define physical constraints for critical blocks (e.g., DSP pipelines, memory controllers).
- Regularly inspect timing reports and floorplan congestion maps during iteration.

Resource estimation and floorplanning are foundational to high-quality FPGA development. Estimating usage of logic, memory, and DSP elements ensures the design fits and performs as expected, while strategic floorplanning improves timing, routability, and power efficiency. Together, they help create scalable and robust FPGA systems that meet functional and performance goals.

7.13 Resource Utilization Reports

After synthesizing and implementing an FPGA design, detailed reports are generated that quantify how the design maps onto the available hardware resources. These resource utilization reports are critical for verifying that the design fits within the target device, analyzing performance, and guiding optimization efforts in subsequent development iterations.

Key Metrics in Utilization Reports

The following key metrics are commonly presented in post-synthesis and post-implementation reports:

- **Slice Usage:**
  - Slices are the basic physical units of programmable logic, consisting of LUTs and flip-flops.
  - Utilization is reported as the number of slices used vs. available, often presented as a percentage.
- **LUTs and Flip-Flops:**
  - LUTs implement combinational logic, while flip-flops provide sequential logic storage.
  - Reports distinguish between LUTs used for logic, arithmetic, or memory (i.e., distributed RAM).
  - Flip-flop counts help assess the depth and complexity of pipelining in the design.
- **Block RAM (BRAM):**
  - Shows number of BRAM tiles used and their configuration (depth $\times$ width).
  - Differentiates between true dual-port and single-port usage.
- **DSP Slices:**
  - Reflects usage of hardware multipliers and accumulators.
  - Important in signal processing, neural network, or image processing workloads.
- **Power Estimates:**
  - Tools provide static and dynamic power breakdowns by logic, memory, clocking, and I/O.
  - Helps identify high-consumption blocks that may benefit from clock gating or optimization.
- **Timing Analysis:**
  - Includes worst negative slack (WNS), total negative slack (TNS), and achieved clock frequencies.
  - Essential for verifying that all timing constraints (setup and hold) are met.

Types of Reports

- **Post-Synthesis Reports:** Show initial resource estimates based on mapped netlists before physical placement.
- **Post-Implementation Reports:** Provide actual usage after place-and-route, including routing congestion and clocking overheads.
- **Power Reports:** Include thermal estimates and suggest optimizations to reduce switching activity.
- **Timing Reports:** Generated by Static Timing Analysis (STA) engines, including hold/setup violations and path delays.

Tool Outputs (Examples)

- `Vivado Utilization Report:`
  - Displays categories like Logic LUTs, LUTRAM, Flip-Flops, DSP48, and BRAM36.
  - Can be exported as text, XML, or JSON for automation and scripting.
- `Quartus Resource Usage Summary:`
  - Offers graphical breakdowns of Adaptive Logic Modules (ALMs), memory blocks, and MLABs.
  - Reports available both in the GUI and as command-line output.

Resource utilization reports are indispensable for analyzing how efficiently a design fits and performs on a target FPGA. By reviewing slice, LUT, flip-flop, BRAM, DSP, power, and timing statistics, designers can make informed decisions to optimize architecture, adjust logic partitioning, and refine timing paths. These reports ultimately help close timing and reduce resource bottlenecks, ensuring a successful and sca

7.14 Power Management in FPGAs

Power efficiency is a key design consideration in FPGA-based systems, especially in battery-operated, portable, or thermally constrained environments. Power consumption in FPGAs can be broadly classified into dynamic and static components. Understanding and managing these components is essential for creating energy-efficient designs.

Dynamic Power Consumption

Dynamic power arises primarily from the switching activity within the FPGA. It is affected by the following factors:

- **Switching Activity:** Power is consumed when logic gates toggle states. Higher toggle rates and more transitions lead to greater power dissipation.
- **Clocking:** Clock trees are among the highest contributors to dynamic power due to their wide fanout and continuous toggling.
- **I/O Drivers:** High-speed and high-drive-strength I/O standards (e.g., LVDS, SSTL) consume significant power during signal transitions, especially with high capacitive loading.

Dynamic power can be reduced through:

- Clock gating unused logic blocks.
- Reducing toggle rates with pipelining or lower operating frequencies.
- Using lower-power I/O standards or reducing external capacitance.

Static Power Consumption

Static power is the power drawn when the FPGA is powered on but not actively switching. It is primarily caused by leakage currents in transistors and includes:

- **Subthreshold Leakage:** Current that flows even when a transistor is off.
- **Gate Leakage:** Due to tunneling effects in thin oxide layers.

Static power is impacted by the process technology (e.g., 28nm vs. 7nm), temperature, and supply voltage. While harder to mitigate through logic design, it can be managed by:

- Powering down unused blocks or banks.
- Selecting FPGAs with lower static power specifications.
- Using programmable power modes or partial reconfiguration features.

Power-Aware Design Techniques

Designers can adopt several best practices for power-aware synthesis and implementation:

- **Power-Aware Synthesis:** Enable synthesis options that optimize for both area and power.
- **Clock Gating:** Introduce enable-controlled clock logic to disable clocks for idle logic regions.
- **Voltage Scaling:** Use lower core voltages when available (e.g., 0.9V instead of 1.0V for core logic).
- **Floorplanning:** Physically group logic to reduce interconnect switching distances.

Tool Support and Reports

FPGA design tools such as Xilinx Vivado and Intel Quartus provide detailed power estimation and analysis features:

- Post-implementation power reports based on actual toggle rates and routing.
- Early-stage power estimators using spreadsheets or IP configuration GUIs.
- Breakdown by category: logic, BRAM, DSP, I/O, and clocking.

Effective power management in FPGAs involves understanding both dynamic and static power contributors and applying design strategies such as clock gating, power-aware synthesis, and efficient floorplanning. Leveraging vendor tools for early and post-implementation power estimation allows designers to meet power budgets while maintaining performance and functionality.

7.15 Vendor Architectures

FPGA architecture varies significantly across manufacturers and families. Each vendor offers a range of device series optimized for different application domains, including low-power embedded systems, high-speed signal processing, and SoC-level integration. The two leading vendors in the FPGA industry are Xilinx (now part of AMD) and Intel (formerly Altera), each with well-defined architecture families.

7.15.1 Xilinx Architecture

Xilinx offers a broad portfolio of FPGAs under different product families tailored for specific use cases. These devices share a consistent toolchain (Vivado) and similar architectural features such as CLBs, BRAM, DSP slices, and advanced clocking resources.

- **Artix Series:**
  - Optimized for low power and cost-sensitive applications.
  - Ideal for embedded systems, motor control, and low-end communication systems.
- **Spartan Series:**
  - Entry-level FPGA line offering basic functionality at minimal cost.
  - Popular in education, prototyping, and simple control applications.
- **Kintex Series:**
  - Mid-range family offering a balance of performance, power, and logic capacity.
  - Suited for image processing, wireless communication, and high-speed transceivers.
- **Virtex Series:**
  - High-end FPGAs with maximum performance, bandwidth, and integration.
  - Used in aerospace, defense, high-frequency trading, and data centers.
- **Zynq SoC and Zynq UltraScale+:**
  - Combines ARM Cortex-A9 or Cortex-A53 cores with programmable logic.
  - Supports full Linux systems and hardware/software co-design.
  - Ideal for applications requiring real-time processing, such as autonomous systems and industrial control.

7.15.2 Intel (Altera) Architecture

Intel's FPGA families target a similar range of application needs and are supported by the Quartus Prime toolchain.

- **Cyclone Series:**
  - Designed for low-cost, low-power applications.
  - Used in consumer electronics, automotive, and IoT.

- **MAX Series:**
  - Combines CPLD-like instant-on features with basic FPGA capabilities.
  - Used for system control, configuration, and low-density logic integration.
- **Arria Series:**
  - Mid-tier devices offering good performance with lower power.
  - Often used in wireless infrastructure, broadcast, and edge computing.
- **Stratix Series:**
  - High-performance FPGAs for advanced processing and transceiver integration.
  - Targeted at high-bandwidth applications such as 5G, cloud acceleration, and real-time analytics.

Comparison Highlights

- **Toolchains:** Xilinx uses Vivado and Vitis; Intel uses Quartus Prime.
- **Soft Processors:** Xilinx supports MicroBlaze; Intel supports Nios II.
- **Hard SoCs:** Xilinx integrates ARM Cortex in Zynq; Intel uses ARM cores in its SoC FPGAs.
- **Transceivers and HBM:** High-end families from both vendors offer transceivers exceeding 100 Gbps and integrated high-bandwidth memory (HBM).

Both Xilinx and Intel offer FPGA architectures that span from cost-sensitive, low-power devices to performance-driven SoC-class platforms. Understanding the features and strengths of each family helps designers select the most appropriate device for their system requirements, development workflow, and long-term scalability.

7.16 Example Application Mapping

Mapping real-world applications onto FPGA resources involves identifying how functional blocks are implemented using the available logic, memory, and arithmetic primitives. This section illustrates how various digital designs leverage core FPGA components such as CLBs, BRAM, and DSP slices. Understanding this mapping is crucial for writing synthesizable hardware descriptions and estimating resource requirements.

7.16.1 FFT Processor

A Fast Fourier Transform (FFT) processor is commonly used in signal processing applications to analyze frequency components of digital signals. Its implementation on an FPGA efficiently utilizes:

- **BRAM:** Used to store input samples, intermediate results, twiddle factors, and output values. Dual-port access facilitates parallel memory operations.
- **DSP Slices:** Perform complex multiplication and butterfly operations. Using dedicated DSP blocks allows higher throughput and pipelining.
- **CLBs:** Implement control logic and finite-state machines (FSMs) to manage data flow and FFT stage transitions.

Mapping Summary: The FFT processor combines memory and arithmetic resources to perform high-speed transforms. The use of pipelining and data reuse across BRAM and DSP units enhances efficiency and timing closure.

7.16.2 UART Engine

A UART is a fundamental communication block used in embedded systems. Its FPGA implementation typically uses:

- **CLBs:** Handle protocol control logic such as baud rate generation, start/stop bit detection, parity checking, and finite state machines.
- **BRAM:** Acts as a transmit/receive buffer (FIFO) to store characters during burst communication. Reduces data loss in case of CPU latency.
- **I/O Blocks:** Used to drive and receive external serial lines (TX, RX).

Mapping Summary: The UART engine is a lightweight IP core that benefits from the flexibility of CLBs and on-chip BRAM for buffering, providing reliable serial communication at moderate data rates.

7.16.3 Simple CPU Core

A basic CPU core for educational or soft-processor applications maps well onto FPGA primitives:

- **ALU (Arithmetic Logic Unit):** Implemented using CLBs for bitwise and arithmetic operations such as add, subtract, AND, OR, XOR, and shift.

- **Register File:** Mapped to BRAM or distributed RAM depending on size. Dual-port BRAM enables concurrent access to multiple registers.
- **Control Unit:** Designed as an FSM to decode opcodes and control pipeline stages (fetch, decode, execute, write-back).
- **Instruction Memory:** Stored in BRAM with optional pre-initialization.
- **Program Counter and Branch Logic:** Implemented using flip-flops and multiplexers in CLBs.

Mapping Summary: The CPU core showcases hardware/software co-design, enabling instruction execution from BRAM and logic sequencing through CLBs. With modular design, additional components such as UART or SPI controllers can be mapped and interfaced with ease.

These application examples demonstrate how different FPGA resources are utilized based on functional requirements. Complex arithmetic tasks benefit from DSP slices, memory-intensive operations leverage BRAM, and control and decision logic are best mapped to CLBs. Proper application-resource mapping leads to optimized performance, reduced congestion, and efficient power usage.

7.17 Summary

FPGAs are versatile digital devices built from configurable logic blocks, memory elements, DSP slices, routing fabric, and I/O resources. They also support advanced features like embedded processors and high-speed interfaces.

Understanding FPGA architecture helps designers effectively map applications to hardware, optimize performance, and manage power and timing. Tools for synthesis, floorplanning, and resource analysis further support efficient development.

As FPGAs evolve with more integration and AI capabilities, a solid grasp of their architecture enables designers to build scalable and high-performance digital systems across various domains.

The laboratory exercises for Chapter 7: *FPGA Architecture and Resources* draw on Lab 2: Nexys A7 Setup and Lab 8: Simple ALU Design, both of which are provided in the Appendix section.

7.18 Exercises

1. **Identify FPGA Resources:** List and describe the primary components of a target FPGA (e.g., Artix-7 or Cyclone V), including CLBs, BRAM, DSPs, and I/O

blocks. Use a vendor datasheet or tool (Vivado, Quartus) to extract actual resource counts.

2. **Logic Block Utilization:** Design a 4-bit multiplier in Verilog. Synthesize and report its LUT, FF, and DSP usage. Discuss any optimizations to reduce logic consumption.

3. **Implement a RAM Module:** Write a synthesizable dual-port RAM module using inferred block RAM. Verify its synthesis result and compare with distributed RAM implementation.

4. **DSP Slice Integration:** Implement an 8-bit MAC (Multiply-Accumulate) unit in Verilog. Ensure that the synthesis tool maps it to dedicated DSP blocks. Analyze the resulting timing and area.

5. **Clocking Resources:** Create a clock divider circuit and implement it using PLL or MMCM primitives available in your FPGA toolchain. Describe clock constraints and derived frequencies.

6. **BRAM vs. Distributed RAM:** Compare two versions of a FIFO design: one using BRAM, the other using LUT-based distributed RAM. Evaluate performance and resource utilization trade-offs.

7. **I/O Block Constraints:** Define timing and voltage constraints for input/output pins using an XDC or SDC file. Assign appropriate IOSTANDARD and drive strength for different peripherals.

8. **FPGA Floorplanning (Advanced):** Explore the floorplanning interface in Vivado or Quartus. Manually assign a small module to a specific region and observe its placement impact.

Bibliography

[1] C. Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes, 2004.

[2] D. A. Pucknell and K. Eshraghian, *Basic VLSI Design*. New Delhi, India: Prentice Hall, 2005.

[3] I. Kuon and J. Rose, "FPGA architecture: Survey and challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2007.

[4] K. Roy and C. Chakrabarti, Eds., *Low-Energy FPGAs: Architecture and Design*. Boston, MA, USA: Springer, 2009.

[5] M. Manohar and J. Bhasker, *Digital System Design Using FPGA: Implementation with Verilog and VHDL*. New York, NY, USA: McGraw-Hill, 2017.

[6] R. Merrick, *Getting Started with FPGAs: Digital Circuit Design, Verilog, and VHDL for Beginners*. San Francisco, CA: No Starch Press, 2023.

[7] U. Farooq, W. Luk, A. Tumeo, and C. Ebeling, "FPGA architectures: An overview," Univ. of South Florida, 2012.

[8] G. Borriello, C. Ebeling, S. Hauck, and S. Burns, "The Triptych FPGA architecture," *IEEE Transactions on VLSI Systems*, vol. 3, no. 4, pp. 473–482, Dec. 1995, doi: https://doi:10.1109/92.475968.

[9] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Hoboken, NJ, USA: Wiley, 2007.

[10] M. H. Quraishi, "A survey of system architectures and techniques for FPGA virtualization," *arXiv preprint*, arXiv:2001.01234, 2020. [Online]. Available: https://arxiv.org/abs/2011.09073

[11] A. Boutros, A. Arora, and V. Betz, "Field-programmable gate array architecture for deep learning: Survey & future directions," *arXiv preprint*, arXiv:2404.12345, Apr. 2024. [Online]. Available: https://arxiv.org/abs/2404.10076

[12] Intel Corporation, "FPGA architecture overview," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683152/22-2/fpga-architecture-overview.html.

[13] Xilinx Inc., *Vivado Design Suite User Guide*, 2023. [Online]. Available at: https://www.xilinx.com/products/design-tools/vivado.html

[14] Siemens EDA (Mentor Graphics), *ModelSim User's Manual*, 2021. [Online]. Available at: https://eda.sw.siemens.com/en-US/ic/model/

[15] Stephen Williams and contributors, *Icarus Verilog and GTKWave Open Source Tools*, 2024. [Online]. Available at: http://iverilog.icarus.com and http://gtkwave.sourceforge.net

Chapter 8

FPGA Design Flow and Toolchains

Chapter Objectives

- Learn the steps in FPGA design, from coding to hardware testing.
- Use FPGA tools for simulation, synthesis, and implementation.
- Apply constraints and debug FPGA designs using built-in tools.

8.1 Introduction to FPGA Design Flow

The FPGA design flow is a structured process that transforms a digital hardware concept into a working circuit implemented on an FPGA device. It begins with defining the design specifications, including system requirements, interfaces, performance goals, and the target FPGA platform. The next step involves writing HDL code, typically in Verilog or VHDL, to describe the behavior and structure of the circuit. This code is functionally verified through simulation using testbenches and tools like ModelSim, Vivado Simulator, or Icarus Verilog with GTKWave. After verifying correctness, the design is synthesized into a gate-level netlist, which maps the logic to available FPGA resources. The implementation phase follows, where the synthesized logic is placed and routed within the physical architecture of the FPGA, ensuring that timing constraints such as setup and hold times are met. Once the design passes timing and resource checks, a bitstream file is generated. This bitstream is then loaded onto the FPGA hardware for real-world testing using input/output peripherals like LEDs, switches, or communication interfaces. Throughout this flow, debugging tools like Integrated Logic Analyzers (ILA) and timing reports aid in refining the design. Popular EDA tools supporting this process include **Xilinx Vivado** for Xilinx devices and **Intel Quartus Prime** for Intel FPGAs. Mastery of the FPGA design flow is essential for producing reliable, high-performance digital systems in both prototyping and production environments.

8.2 Overview of Design Steps

Designing a digital system on an FPGA involves a sequence of structured steps, from defining the system requirements to testing the implemented hardware. Each step ensures that the design is functionally correct, optimized for the target FPGA, and ready for deployment. Below is a typical FPGA design flow:

1. **Design Specification:** Define the system functionality, performance requirements, input/output behavior, and constraints. This includes creating block diagrams, timing diagrams, and use-case scenarios.

2. **HDL Coding (Verilog or VHDL):** Write the digital logic using a hardware description language (HDL), such as Verilog or VHDL. Modules are written to describe behavior and structure of the hardware system.

3. **Functional Simulation:** Simulate the HDL code using testbenches to verify correct functionality before hardware synthesis. This step helps detect logical and syntax errors early.

4. **Synthesis:** Convert the HDL code into a netlist of logic gates and flip-flops that can be mapped to the FPGA's logic elements. Tools like Xilinx Vivado or Intel Quartus perform this step.

5. **Implementation (Place and Route):** Map the synthesized design to physical FPGA resources, place logic blocks, and route the connections between them. Timing analysis is also performed to ensure clock constraints are met.

6. **Bitstream Generation:** Generate the final binary file (bitstream) used to program the FPGA. This file contains the complete configuration data that maps logic and interconnects.

7. **Programming and Hardware Testing:** Load the bitstream onto the FPGA using a hardware programmer or JTAG interface. Test the system in real hardware using inputs and outputs to ensure correct operation.

Each of these steps plays a critical role in ensuring the design's reliability, performance, and correctness. Mastery of this workflow is essential for successful FPGA-based development.

8.3 Design Entry

The design entry phase is the starting point for implementing digital systems on an FPGA. This is where the desired system behavior is described using a suitable input

method, most commonly an HDL.

Verilog HDL is widely used for design entry due to its concise syntax, simulation support, and suitability for both behavioral and structural modeling. Designers typically follow a modular approach, breaking down large systems into smaller, reusable components. This modularity improves code organization, simplifies debugging, and promotes design reuse across multiple projects.

In hierarchical modeling, top-level modules instantiate lower-level submodules, creating a tree-like design structure. This approach enhances readability and allows teams to divide work among multiple designers efficiently.

In addition to HDL-based entry, modern design tools offer graphical design entry options. Tools such as Xilinx IP Integrator and Intel Platform Designer allow developers to visually connect IP blocks, processors, peripherals, and custom logic. These block-based systems are especially useful in system-on-chip designs where embedded processors and complex interconnects are involved.

Design entry may also include:

- Constraint files (e.g., XDC for Xilinx) to specify timing and pin assignments.
- Parameter files or generics to control configurable module behavior.
- Simulation testbenches to verify the correctness of the HDL modules.

Effective design entry is critical for successful synthesis, simulation, and system-level integration. Clarity, modularity, and correctness at this stage ensure a smoother overall design flow.

8.4 Functional Simulation

Functional simulation is essential for verifying the correctness of digital logic before synthesis. It allows designers to test the behavior of HDL modules under various input conditions without needing physical hardware.

Popular simulation tools include:

- **Xilinx Vivado Simulator** — integrated into the Vivado design suite.
- **ModelSim / QuestaSim** — commonly used with Intel (Altera) Quartus.
- **GTKWave** — an open-source waveform viewer for visualizing VCD files.

Testbenches are used to apply input stimuli and observe the design's output. A testbench is written in Verilog (or VHDL) and includes procedural code that simulates various operating scenarios.

Important Verilog simulation tasks:

```
$display()    // Print output to the console
$monitor()    // Automatically display changes in signal values
$dumpfile()   // Specify a waveform output file (e.g., .vcd)
$dumpvars()   // Define which variables to record
```

These tasks help generate simulation results for debugging and waveform inspection.

Example: Simple Testbench Snippet

```
initial begin
  $dumpfile("test.vcd");
  $dumpvars(0, uut);  // 'uut' is the design module
  A = 4'b0011; B = 4'b0101;
  #10 A = 4'b1111; B = 4'b0001;
  #10 $finish;
end
```

Waveform viewers like GTKWave can load the generated '.vcd' file to display signal transitions over time.

Functional simulation ensures that design logic operates as intended and identifies bugs before moving to synthesis. This step is critical for saving time and improving design quality.

8.5 Synthesis

SynthesisindexSynthesis is the process of converting RTL code—typically written in Verilog or VHDL—into a gate-level netlist that represents the digital logic in terms of logic gates and flip-flops. This netlist is then mapped to the specific resources available on the target FPGA device, such as LUTs, FFs, DSP slices, and BRAMs.

Modern synthesis tools, such as Xilinx Vivado and Intel Quartus, perform several tasks during synthesis:

- Optimize logic expressions and reduce resource usage.
- Map high-level HDL constructs to FPGA primitives.
- Check for coding errors, timing violations, and design rule violations.

The output of the synthesis step typically includes:

- **Netlist File:** A structural description of the design, representing gates and their connections.

- **Constraint File Report:** A summary of user-defined constraints such as clock frequencies, pin assignments, and timing requirements.
- **Utilization Summary:** A report showing how many logic resources (LUTs, FFs, BRAM, DSPs) the design will use.

This information is critical for evaluating whether the design will fit within the target FPGA and for guiding subsequent steps such as implementation, floorplanning, and power optimization.

Synthesis is a key stage in the digital design flow, bridging the abstract RTL description with physical hardware realizability.

8.6 Constraints and I/O Planning

Constraints and input/output (I/O) planning are essential components of FPGA design, ensuring that the design can meet performance requirements and integrate correctly with external hardware.

Constraints define critical physical and timing characteristics of the design. These include:

- **Clock Timing:** Specifies frequency, duty cycle, clock uncertainty, and timing margins. Proper clock constraints are vital to ensure reliable synchronous operation and to help the synthesis and implementation tools achieve timing closure.
- **I/O Assignments (Pin Mapping):** Maps logical I/O ports from the HDL code to specific physical pins on the FPGA package. Incorrect mapping may result in malfunctioning or damaged devices, especially when connected to external peripherals.
- **I/O Standards:** Define electrical characteristics such as voltage levels and signaling types (e.g., LVCMOS33, SSTL, LVDS). These settings ensure signal compatibility with other system components and avoid damage or signal integrity issues.

Most FPGA vendors use dedicated constraint file formats:

- **Xilinx:** Uses XDC (Xilinx Design Constraints) files, which are based on Synopsys Design Constraints (SDC) syntax but customized for Xilinx toolchains (e.g., Vivado).
- **Intel:** Uses SDC (Synopsys Design Constraints) files, compatible with the Intel Quartus software suite.

Constraint files are typically edited manually or generated using graphical I/O planning tools provided by the FPGA design suite. For example, Xilinx Vivado includes an I/O Planning view where users can drag-and-drop ports onto specific FPGA pins while visualizing bank voltages and I/O standard compatibility.

Proper constraint definition and I/O planning are essential for successful synthesis, timing analysis, and implementation. They ensure correct operation of the final design in a real-world environment and are critical to achieving optimal performance and reliability.

8.7 Implementation: Place and Route

The implementation phase, also referred to as "place and route," translates the synthesized netlist into a physical layout on the FPGA. This step is crucial for optimizing the design to meet timing, area, and power constraints.

Implementation involves the following key steps:

- **Logic Mapping:** The synthesized netlist, consisting of logic gates and flip-flops, is mapped onto the available CLBs, DSP slices, BRAMs, and other primitives in the FPGA fabric.
- **Placement:** Logic blocks are placed within the FPGA fabric in a way that minimizes signal delays and satisfies area constraints. Optimal placement reduces interconnect length and supports better timing performance.
- **Routing:** Interconnections between placed blocks are established through the FPGA's programmable routing fabric. The router ensures that all signals are connected as specified in the design netlist and that timing constraints are respected.

After implementation, the tools generate detailed reports to help evaluate the quality of the result:

- **Timing Summary:** Includes Worst Negative Slack (WNS) and Total Negative Slack (TNS), which indicate how far the design is from meeting required timing.
- **Routing Congestion:** Shows how efficiently the routing resources are utilized and highlights areas where signal congestion may lead to timing or placement issues.
- **Design Rule Checks (DRCs):** Verifies that all physical and electrical design rules have been met, such as I/O standard compliance, pin usage, and clock domain integrity.

Successful implementation ensures that the design meets timing, resource, and connectivity requirements, laying the foundation for bitstream generation and final hardware testing.

8.8 Bitstream Generation

After successful implementation and timing closure, the next step is to generate the bitstream — a binary configuration file that programs the FPGA hardware. This file encodes the complete physical design, including logic placement, routing, clocking, and I/O assignments.

The bitstream format varies by vendor:

- **Xilinx:** `.bit` or `.bin` file
- **Intel (Altera):** `.sof` (SRAM Object File) or `.pof` (Programmer Object File) for non-volatile storage

The bitstream is generated after a design passes all implementation checks, including timing, placement, and design rule verification. Most vendor toolchains provide bitstream customization options such as encryption, compression, and CRC protection.

8.9 Programming the FPGA

Programming involves transferring the generated bitstream into the FPGA device. This step can be performed through several hardware interfaces depending on the FPGA vendor and board setup:

- **JTAG Programmer:** A common method for debugging and development, using tools like Vivado Hardware Manager (Xilinx) or Quartus Programmer (Intel).
- **USB Blaster (Intel):** Intel's proprietary interface for FPGA configuration and device communication.
- **Platform Cable USB (Xilinx):** High-speed cable for JTAG-based programming of Xilinx FPGAs.

FPGAs can be categorized as:

- **Volatile FPGAs:** Require reconfiguration on every power-up. The bitstream must be loaded from an external source (e.g., flash memory or microcontroller).
- **Non-Volatile or Flash-based FPGAs:** Retain configuration across power cycles (e.g., Xilinx Spartan-6Q with integrated flash or Intel Max 10).

In production environments, the bitstream is often preloaded into external flash memory. Upon power-up, the FPGA automatically loads the bitstream into its configuration logic, allowing for autonomous and reliable deployment.

8.10 In-System Debugging

Once an FPGA design is deployed on hardware, functional verification often requires real-time observation of internal signals and states. In-system debugging tools enable designers to probe, capture, and analyze signal activity while the FPGA is running in its actual operating environment.

Common in-system debugging techniques include:

- **Logic Analyzer Cores (e.g., ILA in Xilinx):** Integrated Logic Analyzer (ILA) cores allow designers to tap internal nets and observe signal transitions in real time. Signals are sampled and displayed as waveforms using a software interface (e.g., Vivado Hardware Manager).

- **Virtual I/O (VIO):** VIO cores enable bi-directional communication between the FPGA and the host PC. Designers can dynamically toggle inputs or read signal states without changing the HDL code or recompiling the bitstream.

- **UART Print Statements:** For debugging FSMs or embedded soft processors, UART can be used to transmit internal states or variable values to a terminal on the host PC. It is simple but effective for embedded software and control-path debugging.

These tools are particularly valuable for diagnosing issues such as timing errors, logic glitches, initialization faults, or communication failures that may not manifest during simulation.

Using debug cores requires inserting them during synthesis or implementation, so planning for debug instrumentation early in the design cycle is recommended. Most FPGA toolchains provide automation features to insert and connect these cores with minimal disruption to the primary logic.

In-system debugging significantly accelerates the bring-up phase of FPGA-based systems and enables iterative refinement of hardware designs.

8.11 Toolchains

Modern FPGA development relies on powerful toolchains provided by vendors. These tools manage every stage of the design flow, from HDL entry and simulation to synthesis, implementation, and programming. Toolchains also support debugging, constraint management, and IP integration.

8.11.1 Xilinx Vivado

Xilinx Vivado is the primary development environment for Xilinx 7-series and UltraScale FPGAs. It offers a unified, feature-rich platform for designing, simulating, and implementing digital systems.

Key features of Vivado include:

- **IP Integrator for Block Design:** A graphical tool that allows designers to integrate intellectual property (IP) cores, including processors, peripherals, and custom modules, using a drag-and-drop interface.
- **Schematic and RTL View:** Enables visualization of synthesized designs in schematic or RTL-level formats to verify signal flow, hierarchy, and design structure.
- **Vivado Simulator:** A built-in simulation engine that supports functional and timing simulation with waveform analysis, compatible with Verilog, VHDL, and mixed-language projects.
- **Hardware Manager for Programming:** Provides a JTAG interface for downloading bitstreams, monitoring debug cores (e.g., ILA), and performing in-system testing of FPGAs.

Vivado also supports Tcl scripting for automation, IP packaging, floorplanning, power analysis, and integration with third-party tools. It is the standard toolchain for academic, industrial, and embedded FPGA applications using Xilinx devices.

8.11.2 Intel Quartus Prime

Intel Quartus Prime is Intel's flagship FPGA design suite, supporting a wide range of devices including the Cyclone, MAX, Arria, and Stratix families. It provides a full development flow from design entry to bitstream generation and in-system debugging.

Key features include:

- **Platform Designer:** A visual integration tool for connecting IP cores, memory interfaces, and processors such as the Nios II. It enables rapid development of SoC-style architectures.
- **TimeQuest Timing Analyzer:** A powerful static timing analysis engine for validating timing constraints, analyzing clock domains, and ensuring reliable operation under all timing scenarios.
- **Programmer Tool:** Used to upload the compiled bitstream (.sof or .pof) to the FPGA via JTAG or Active Serial (AS) interface. It also supports device chain programming and flash configuration.

Quartus Prime is available in three editions:

- **Lite Edition:** Free for smaller FPGA devices (e.g., Cyclone IV/V), ideal for education and prototyping.
- **Standard Edition:** Supports mid-range devices and offers full functionality for most commercial applications.
- **Pro Edition:** Targeted at high-end Stratix and Agilex devices with support for hyper-retiming, advanced debugging, and fast compilation.

Tool Comparison: Vivado vs. Quartus Prime

Table 8.1 compares the key features of Xilinx Vivado and Intel Quartus Prime toolchains, including device support, IP integration, simulation tools, debugging capabilities, and scripting options—essential considerations when selecting an FPGA development environment.

Table 8.1: Comparison of Xilinx Vivado and Intel Quartus Prime toolchains

Feature	**Xilinx Vivado**	**Intel Quartus Prime**
Target Devices	7-series, UltraScale, Zynq	Cyclone, MAX, Arria, Stratix
Graphical IP Integration	IP Integrator	Platform Designer
Timing Analysis Tool	Vivado Timing Analyzer	TimeQuest Timing Analyzer
Simulation Tool	Vivado Simulator	ModelSim (or external)
Programming Utility	Vivado Hardware Manager	Quartus Programmer Tool
In-System Debugging	ILA, VIO	Signal Tap Logic Analyzer
Design Entry Options	HDL, Block Diagram, HLS	HDL, Block Diagram
Licensing Options	WebPACK (free), HLx editions	Lite, Standard, Pro
Scripting Support	Tcl-based scripting	Tcl and QSF/QIP scripting

8.11.3 Other Tools

In addition to vendor-specific toolchains, several open-source and third-party tools support FPGA design, simulation, synthesis, and implementation. These tools offer flexibility, portability, and advanced analysis capabilities, particularly for academic and research use.

- **Open-source Tools:**
  - **Yosys:** A popular open-source synthesis tool that supports Verilog RTL and generates gate-level netlists. It is widely used in conjunction with open FPGA flows.

  - **nextpnr:** An open-source place-and-route tool compatible with several FPGAs (e.g., Lattice iCE40, ECP5). It works with netlists generated by Yosys and supports custom constraint files.
  - **GHDL:** An open-source VHDL simulator that also supports mixed-language simulation when used with tools like Yosys and GTKWave.
  - **Verilator:** An open-source simulator that compiles synthesizable Verilog into C++ or SystemC for cycle-accurate simulation.

- **Third-party Commercial Tools:**
  - **ModelSim / QuestaSim:** Industry-standard simulators from Siemens EDA (formerly Mentor Graphics), supporting VHDL, Verilog, and SystemVerilog. Often used with both Xilinx and Intel designs.
  - **Aldec Riviera-PRO / Active-HDL:** Simulation and verification tools supporting mixed-language projects, assertions, coverage analysis, and waveform viewing.
  - **Synplify Pro:** A high-performance synthesis tool used for optimized RTL-to-gate conversion, especially in complex designs targeting multiple vendors.

These tools are commonly integrated into educational, research, and commercial design workflows where cross-platform compatibility or custom back-end flows are needed.

8.12 Timing Analysis and Optimization

Timing analysis is a critical step in FPGA design that ensures signals propagate correctly between registers and logic blocks within the required clock periods. The goal is to meet all setup and hold time requirements so that the design operates reliably at the desired frequency.

FPGA toolchains provide STA tools to analyze all timing paths in a design based on constraints specified in the timing constraint file (e.g., XDC for Xilinx or SDC for Intel).

8.12.1 Timing Closure

Timing closure refers to the condition when all timing paths in the design satisfy their respective setup and hold time requirements across all clock domains and operating conditions.

Key concepts:

- **Setup Time:** The minimum time before the clock edge by which data must arrive at a register input.

- **Hold Time:** The minimum time after the clock edge during which data must remain stable at the register input.
- **Worst Negative Slack (WNS):** Indicates how much a path fails to meet the setup requirement. Positive WNS means all paths meet timing.
- **Total Negative Slack (TNS):** The sum of all negative slack values across failing paths, providing an aggregate measure of timing violations.

Common strategies to achieve timing closure:

- Pipeline long combinational paths to reduce logic delay.
- Use retiming and register balancing during synthesis.
- Apply multicycle path and false path constraints accurately.
- Optimize floorplanning and placement to minimize routing delay.
- Adjust clock constraints and leverage clock management resources.

Successfully achieving timing closure ensures the design will run at the intended clock speed without timing-related errors, which is crucial for performance-critical and real-time applications.

8.12.2 Techniques

To meet timing constraints and close timing on high-speed or complex designs, several optimization techniques are employed throughout the synthesis and implementation phases. These techniques aim to reduce critical path delays, improve setup and hold margins, and ensure stable operation at the desired clock frequency.

- **Pipelining to Shorten Critical Paths:** By inserting additional registers between logic stages, pipelining reduces the amount of combinational logic between flip-flops. This shortens the critical path and allows the design to run at higher clock frequencies. Although pipelining increases latency, it significantly improves throughput.
- **Logic Restructuring:** Optimizing Boolean expressions, reordering computations, and balancing logic depth help reduce propagation delay. Synthesis tools can perform these optimizations automatically, but designers can further refine critical blocks manually.

- **Clock Constraint Tuning:** Accurate and well-defined clock constraints are essential for effective timing analysis. Designers can tune clock periods, define multicycle paths, and specify false paths to inform the timing engine and prevent unnecessary violations.

Combining these techniques allows designers to achieve a balance between performance, area, and power while ensuring the design functions reliably at its target frequency.

8.13 Design Verification

Design verification ensures that the implemented hardware behaves according to the original specifications and functional requirements. This critical phase helps detect logical errors, synthesis mismatches, and integration issues before final deployment on hardware.

Key stages of verification include:

- **RTL Simulation:** Simulate the RTL code using testbenches to confirm functional correctness at the behavioral level. This step helps detect issues early before synthesis.
- **Post-Synthesis Simulation:** Run simulations on the synthesized netlist to check for functional correctness and timing accuracy after logic optimization. It accounts for gate delays and resource mapping.
- **Hardware Testing:** Program the FPGA with the bitstream and test the system using real-world inputs and peripherals. This step verifies integration, I/O behavior, and system-level operation.

Formal Verification

In addition to traditional simulation, formal verification techniques may be applied:

- **Equivalence Checking:** Compares the RTL design with the synthesized netlist to ensure functional equivalence using mathematical proof methods.
- **Assertion-Based Verification:** Embeds properties and constraints directly in the HDL to automatically check expected conditions during simulation or formal analysis.

Comprehensive verification ensures a high degree of confidence in the final implementation, reducing the likelihood of costly hardware bugs and ensuring system reliability.

8.14 Using Intellectual Property Cores

IP cores are pre-designed, pre-verified blocks of logic provided by FPGA vendors or third-party developers. These cores significantly reduce development time, improve reliability, and offer optimized implementations of complex functionalities.

FPGA vendors such as Xilinx and Intel offer extensive libraries of IP cores through their design suites (Vivado IP Catalog and Quartus Platform Designer, respectively). These cores are designed to be configurable and easily integrable into HDL-based or graphical designs.

Common types of IP cores include:

- **Memory Controllers:** Interfaces for DDR3/DDR4, QDR, SRAM, and Flash memory. These IPs manage complex timing protocols and data alignment internally.
- **Mathematical Functions:** Implementations of multipliers, dividers, FIR/IIR filters, FFT engines, and other signal processing primitives optimized for FPGA resources like DSP slices.
- **Protocol Stacks:** Communication interfaces such as UART, SPI, I2C, CAN, Ethernet, PCIe, and AXI are provided as configurable IPs with standard interface signals and software drivers.

Most IP cores include graphical configuration tools that allow users to:

- Select data widths, buffer sizes, and number of channels.
- Enable optional features (e.g., parity, burst modes).
- Automatically generate wrapper code, simulation models, and documentation.

Benefits of IP Cores

- Save design and verification time.
- Increase performance by using vendor-optimized implementations.
- Improve portability and system integration in complex projects.

Using IP cores effectively allows designers to focus on high-level architecture and application-specific logic rather than reinventing common infrastructure components.

8.15 Project Management in EDA Tools

Efficient project management is critical for developing and maintaining scalable, reusable, and error-free FPGA designs. EDA tools like Xilinx Vivado and Intel Quartus provide structured workflows to manage files, configurations, and automation scripts.

Key practices include:

- **Use Project Directories and Version Control:** Organize source files, simulation models, constraint files, and scripts into a well-structured project directory. Use version control systems like Git to track changes, manage branches, and collaborate with teams.

- **Use Constraints, Scripts, and Automation:** Leverage scripting (TCL in Vivado, Quartus scripting interface) to automate repetitive tasks such as compilation, simulation, synthesis, and implementation. Maintain centralized constraint files (XDC or SDC) to manage I/O locations, clock frequencies, and timing requirements.

- **Leverage Build Reports and Warnings:** Analyze synthesis and implementation reports for resource usage, timing violations, and optimization suggestions. Heed warnings and critical messages, and use reports to guide optimization and verification.

Effective use of project management features reduces errors, improves repeatability, and ensures that FPGA development flows remain scalable and maintainable across design iterations and team collaboration.

8.16 Summary

The FPGA design flow converts Verilog-based hardware descriptions into functioning digital systems through steps such as simulation, synthesis, implementation, and programming. Each phase ensures correct logic, timing, and resource use.

Proficiency in tools like Xilinx Vivado or Intel Quartus, along with proper constraint and report handling, is key to creating efficient and reliable FPGA designs. A solid grasp of the design process enables engineers to build optimized, reusable hardware systems.

The laboratory exercise for Chapter 8: *FPGA Design Flow and Toolchains* expands on Lab 1: Introduction to Xilinx Vivado, which is provided in the Appendix section.

8.17 Exercises

1. **Complete FPGA Flow:** Design a simple 4-bit counter in Verilog. Perform simulation, synthesis, implementation, and bitstream generation using a selected FPGA toolchain (e.g., Vivado or Quartus Prime).

2. **Toolchain Comparison:** Compare two FPGA development environments (e.g., Vivado vs. Quartus) in terms of user interface, simulation capabilities, synthesis reports, and resource utilization.

3. **Constraint Definition:** Write a basic timing constraint file (XDC or SDC) to define a 100 MHz system clock, input delays, and output delays for a sample design.

4. **Analyze Timing Reports:** Given the post-synthesis timing report of a design, identify the worst negative slack and explain how to resolve timing violations.

5. **IP Core Integration:** Integrate an IP core (e.g., FIFO, UART, or PLL) into your design using the toolchain's IP catalog. Modify your top module and constraints accordingly.

6. **Run a Simulation Workflow:** Create a testbench for a 2-to-1 multiplexer and run behavioral simulation. Analyze waveforms and verify functional correctness using ModelSim or Vivado Simulator.

7. **Bitstream Programming:** Program your synthesized design onto an FPGA development board and verify its operation using onboard switches and LEDs.

8. **Understand Project Files:** Identify and explain the purpose of key project files generated during an FPGA design flow: e.g., `.v`, `.xdc`, `.bit`, `.sdf`, and `.rpt`.

Bibliography

[1] C. Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes, 2004.

[2] M. Manohar and J. Bhasker, *Digital System Design Using FPGA: Implementation with Verilog and VHDL*. New York, NY, USA: McGraw-Hill, 2017.

[3] R. Merrick, *Getting Started with FPGAs: Digital Circuit Design, Verilog, and VHDL for Beginners*. San Francisco, CA: No Starch Press, 2023.

[4] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+nextpnr: An open-source framework from Verilog to bitstream for commercial FPGAs," in *Proc. 2019 IEEE 27th Annual Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, San Diego, CA, USA, 2019, pp. 1–4, doi: https://doi:10.1109/FCCM.2019.00010.

[5] A. Taylor, "Intel FPGA Tool Chain Introduction," *Adiuvo Engineering Blog*, May 2, 2022. [Online]. Available: https://www.adiuvoengineering.com/post/intel-fpga-tool-chain-introduction

[6] "FPGA Design Flow," *CircuitFever*, Jan. 17, 2023. [Online]. Available: https://circuitfever.com/fpga-design-flow

[7] "FPGA Design Flow: 7 Essential Steps to Implementing a Circuit on an FPGA," *FPGATek*, Nov. 27, 2023. [Online]. Available: https://fpgatek.com/fpga-design-flow/

[8] "opensource-toolchain-fpga," GitHub repository by cjacker. [Online]. Available: https://github.com/cjacker/opensource-toolchain-fpga

[9] Xilinx Inc., *Vivado Design Suite User Guide*, 2023. Available at: https://www.xilinx.com/products/design-tools/vivado.html

[10] Intel Corporation, *Intel Quartus Prime Pro Edition User Guide*, 2023. Available at: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/download.html

[11] Siemens EDA (Mentor Graphics), *ModelSim User's Manual*, 2021. Available at: https://eda.sw.siemens.com/en-US/ic/model/

[12] Stephen Williams and contributors, *Icarus Verilog and GTKWave Open Source Tools*, 2024. Available online: http://iverilog.icarus.com and http://gtkwave.sourceforge.net

Chapter 9

System Design and IP Integration

Chapter Objectives

- Learn system design using Verilog and IP cores.
- Connect custom modules with IP blocks and bus interfaces.
- Build and test FPGA systems with memory-mapped I/O.

9.1 Introduction to System-Level Design

System-level design in FPGA development refers to the process of building complete digital systems by combining various functional components such as custom Verilog HDL modules, pre-verified IP (Intellectual Property) cores, soft-core processors (e.g., MicroBlaze or Nios II), and standard communication interfaces (e.g., UART, SPI, I2C). Rather than focusing solely on the internal logic of individual blocks, system-level design emphasizes the structure, connectivity, and functionality of the entire digital system.

This approach enables the rapid development of complex applications such as embedded systems, DSP pipelines, communication subsystems, and sensor fusion platforms. Designers can integrate hardware accelerators with general-purpose processors and memory controllers to implement both control and datapath logic within a single FPGA.

Key benefits of system-level design include:

- **Modularity:** Allows reuse of verified components, reducing development time and errors.
- **Abstraction:** Simplifies system complexity by using block diagrams and interconnect models.
- **Scalability:** Facilitates expansion by adding or modifying subsystems without overhauling the entire design.

- **Rapid Prototyping:** Enables quick testing of architectural concepts and functional validation before deployment.

EDA tools such as **Xilinx Vivado IP Integrator** and **Intel Platform Designer (Qsys)** streamline the creation of system-level designs. These tools provide graphical interfaces to instantiate, connect, and configure IP blocks, automatically generate HDL wrappers and address maps, and facilitate simulation and bitstream generation.

Overall, system-level design is an essential methodology in modern FPGA-based development, bridging hardware, firmware, and software to implement complete and flexible embedded solutions.

9.2 Hierarchical Design in Verilog

Hierarchical design is a fundamental approach in Verilog that organizes a project into multiple levels of modules and submodules. This modular structure improves clarity, reusability, and maintainability of the code.

In this method, complex systems are broken down into smaller components—each encapsulated in its own Verilog module. A top-level module instantiates these submodules and manages signal interconnections between them. For example, a digital system like a processor might have separate modules for the ALU, control unit, register file, and memory interface.

Each module can be developed and tested independently using simulation, then integrated into a larger system. This approach also allows teams to work in parallel on different parts of the design, streamlining the development process.

Hierarchical modeling also supports parameterization, enabling the reuse of the same module for different configurations. This is useful for scalable systems such as N-bit adders, configurable counters, or memory blocks.

Example:

```
// Top-level module
module top_module(input clk, input [3:0] a, b, output [4:0] sum);
    adder4bit u1 (.A(a), .B(b), .SUM(sum));
endmodule

// Submodule
module adder4bit(input [3:0] A, B, output [4:0] SUM);
    assign SUM = A + B;
endmodule
```

This structure demonstrates how a simple adder can be reused and integrated within a larger system using hierarchical design in Verilog.

9.2.1 Benefits

Hierarchical design in Verilog provides several advantages that enhance both development and maintainability of complex digital systems:

- **Easier debugging and simulation:** Each module can be developed and tested independently, allowing designers to isolate and verify functionality at a granular level before full system integration.
- **Scalability and reuse:** Modules can be parameterized and reused across different projects or system levels. This reduces development time and improves consistency in design practices.
- **Clear separation of functionality:** Logical partitioning of design components improves readability and simplifies teamwork. Designers can assign different modules to different team members, enhancing collaboration and modular integration.

9.2.2 Example

Hierarchical design enables complex systems to be built from reusable and independently testable building blocks. In this example, a top-level module instantiates four submodules: an ALU, a register file, memory interface and a FSM for control logic.

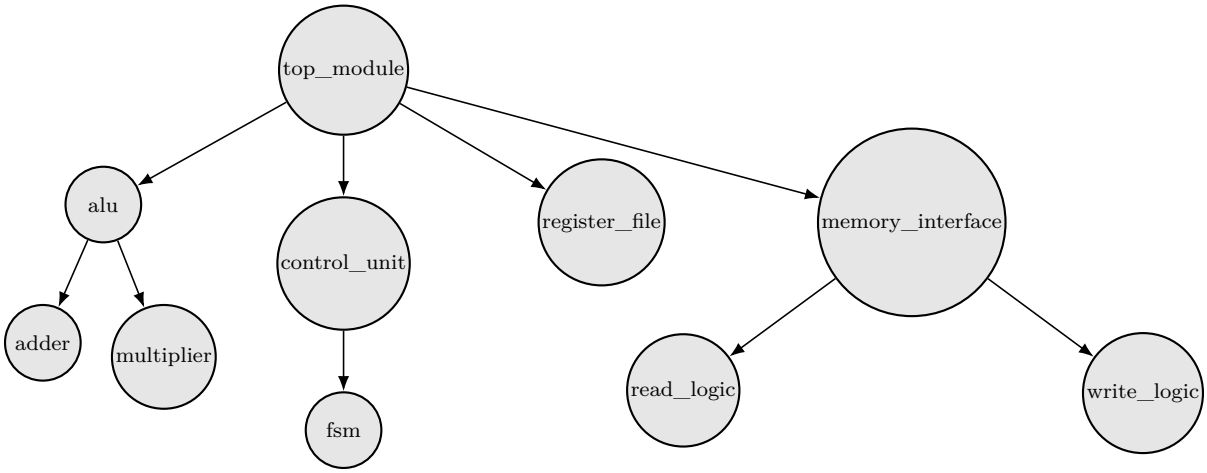


Figure 9.1: Hierarchical design tree

As illustrated in Figure 9.1, the top-level design integrates these components into a coherent architecture that separates datapath and control logic for better modularity and scalability.

```
1 // Top-level module
2 module processor_top(input clk, reset, input [7:0] in_data,
    output [7:0] out_data);
```

```
    wire [7:0] alu_result;
    wire [7:0] reg_data;
    wire [3:0] control_signal;

    // Instantiate ALU
    alu u_alu (
        .clk(clk),
        .control(control_signal),
        .data_in(reg_data),
        .result(alu_result)
    );

    // Instantiate Register File
    register_file u_regfile (
        .clk(clk),
        .reset(reset),
        .data_in(in_data),
        .data_out(reg_data)
    );

    // Instantiate FSM Control
    control_fsm u_fsm (
        .clk(clk),
        .reset(reset),
        .control_out(control_signal)
    );

    assign out_data = alu_result;
endmodule
```

9.3 Modular Design with Parameters

Parameterization in Verilog allows designers to create highly reusable and configurable modules. By defining parameters, a single module can be instantiated with different widths, sizes, or behaviors to suit various design needs without rewriting code.

```
// Parameterized shift register
module shift_reg #(parameter WIDTH = 8)(
    input clk,
    input rst,
    input [WIDTH-1:0] din,
```

```
    output reg [WIDTH-1:0] dout
);
    always @(posedge clk or posedge rst)
        if (rst)
            dout <= 0;
        else
            dout <= din;
endmodule
```

In this example, the `WIDTH` parameter defines the bit-width of the shift register, making the module scalable for different use cases.

Verilog also supports the use of `generate` blocks to instantiate multiple copies of logic elements based on parameters or loops. This is especially useful in building scalable arithmetic units, multiplexers, and register banks.

```
// Generate block to instantiate N flip-flops
generate
    genvar i;
    for (i = 0; i < WIDTH; i = i + 1) begin : gen_ff
        dff u_dff (.clk(clk), .rst(rst), .d(din[i]), .q(dout[i]))
            ;
    end
endgenerate
```

These techniques allow modular design with high flexibility and reduce code duplication across projects, contributing to efficient and scalable hardware development. Table 9.1 compares hardcoded and parameterized Verilog modules in terms of reusability, scalability, code maintenance, and clarity.

Table 9.1: Comparison between hardcoded and parameterized Verilog modules

Aspect	**Hardcoded Module**	**Parameterized Module**
Reusability	Low	High
Scalability	Poor	Excellent
Code Maintenance	Difficult	Simplified
Clarity	Redundant	Concise

9.4 Bus and Interface Design

FPGA-based systems often incorporate multiple functional blocks that need to communicate efficiently. To facilitate structured and scalable interconnections, standardized bus

protocols are commonly used. These bus systems define how data is transferred, how peripherals are accessed, and how system resources are coordinated.

- **AXI (Advanced eXtensible Interface):** Part of the ARM AMBA standard, AXI is widely used for high-performance, memory-mapped communication. It supports burst transfers, separate address and data channels, and is suitable for SoC designs.
- **APB (Advanced Peripheral Bus):** Also from the AMBA standard, APB is optimized for low-bandwidth, low-power communication with peripherals such as UARTs, timers, and GPIO. It uses a simple protocol and is ideal for configuration and control operations.
- **Wishbone:** An open-source interface standard designed for use in FPGA and ASIC designs. It enables the connection of cores in a flexible and platform-independent way. Wishbone is known for its simplicity and modular design.

These interfaces promote modularity and reuse by enabling IP cores to be integrated with minimal adaptation. Most vendor tools, such as Xilinx Vivado and Intel Platform Designer, offer support for these buses through ready-made IP cores and automated interconnection tools.

Understanding these protocols is essential for integrating memory controllers, processors, DMA engines, and custom logic in a cohesive and synchronized system architecture.

9.4.1 Designing Memory-Mapped Interfaces

Memory-mapped interfaces are a foundational technique for organizing communication between a processor (master) and peripherals (slaves) in an FPGA system. In this approach, each peripheral is assigned a unique address range within the processor's memory space. The CPU can then access peripheral registers using standard read and write operations as if they were regular memory locations.

During operation:

- The **master** (typically a CPU or DMA controller) initiates a transaction by providing an address and specifying whether it is a read or write operation.
- The **slave** (e.g., UART, GPIO, timer) responds if the address matches its assigned range, either returning data (in a read) or capturing the input (in a write).

The memory-mapped interface simplifies integration by unifying memory and I/O access under a single protocol. It also supports modular expansion—new peripherals can be added by assigning additional address space and integrating their logic.

A typical memory-mapped slave interface includes:

- Address input
- Write enable and read enable signals
- Data input (for writes) and output (for reads)
- Acknowledge or ready signal

Designers often use bus protocols like AXI4-Lite or APB to implement memory-ma

9.5 Using IP Cores

IP cores are pre-designed, pre-verified digital blocks provided by FPGA vendors or third-party developers. These cores are used to accelerate development by offering ready-to-use implementations of common hardware components, eliminating the need to design complex logic from scratch.

Common examples of IP cores include:

- **Communication Controllers:** UART, SPI, I^2C, CAN
- **Signal Processing Units:** FFT blocks, FIR filters, MAC engines
- **Memory Interfaces:** DDR3/DDR4 controllers, SRAM interfaces

Using IP cores helps ensure correctness, reduces development time, and leverages vendor-optimized performance. These cores are highly configurable, allowing designers to tailor parameters such as data width, clock frequency, FIFO depth, and protocol settings to meet application-specific requirements.

Integration Steps:

1. **Instantiate the IP core:** Use the IP catalog provided by tools like Xilinx Vivado or Intel Quartus Prime to select and generate the desired core.
2. **Configure Parameters:** Adjust core-specific settings using graphical configuration wizards or IP parameter files.
3. **Integrate into HDL Design:** Connect the generated wrapper module to user-defined Verilog code, using defined ports, interfaces, or bus connections (e.g., AXI or Avalon).

IP cores are typically delivered with documentation, timing models, simulation files, and example designs. Designers can integrate them hierarchically within larger systems using block design tools or manually within structural HDL code.

Leveraging IP cores is essential for developing complex FPGA systems such as embedded processors, communication interfaces, and real-time signal processing pipelines.

Example: UART IP Core Integration

To illustrate the usage of an IP core, consider integrating a UART core into an FPGA design. The following Verilog module shows how a UART IP wrapper is instantiated and connected with user logic:

```
module top_uart_system (
    input clk,
    input rst,
    input rx,
    output tx
);
    // Wires for connecting to the UART IP core
    wire [7:0] tx_data;
    wire tx_valid, tx_ready;
    wire [7:0] rx_data;
    wire rx_valid;

    // Instantiate UART IP core (auto-generated wrapper)
    uart_ip_core uart_inst (
        .clk        (clk),
        .resetn     (~rst),
        .rx         (rx),
        .tx         (tx),
        .tx_data    (tx_data),
        .tx_valid   (tx_valid),
        .tx_ready   (tx_ready),
        .rx_data    (rx_data),
        .rx_valid   (rx_valid)
    );

    // Example control logic (user-defined)
    assign tx_data  = 8'hA5;
    assign tx_valid = 1'b1; // transmit continuously

endmodule
```

The UART IP core handles low-level serial communication. The designer only needs to send and receive data through its high-level interface signals such as `tx_data`, `rx_data`, `tx_valid`, and `rx_valid`.

9.6 Soft-Core Processor Integration

Modern FPGAs support the instantiation of embedded soft-core processors, enabling the development of complete system-on-chip (SoC) architectures within the programmable fabric. Soft-core processors are synthesized from HDL and occupy FPGA resources such as LUTs, flip-flops, and block RAM.

Two widely used soft-core processors include:

- **MicroBlaze** – a 32-bit RISC soft processor provided by Xilinx, configurable via the Vivado IP Integrator.
- **Nios II** – Intel's customizable soft-core processor supported within Quartus Platform Designer.

These processors are typically used in applications requiring moderate computational performance, embedded control, and software programmability alongside custom hardware acceleration.

9.6.1 SoC Architecture

A soft-core-based SoC architecture implemented on an FPGA generally includes the following components:

- **Processor Core:** Executes software routines and controls system logic.
- **Memory Interface:** Connects to on-chip Block RAM or off-chip DDR/flash memory for program and data storage.
- **Custom Peripherals:** HDL-based modules providing application-specific logic, such as timers, UARTs, or GPIOs.
- **Interconnect Fabric:** Structured communication bus like AXI (used by MicroBlaze) or Avalon (used by Nios II) enables memory-mapped transactions between processor and peripherals.

These components are connected using vendor-provided tools such as the Xilinx Vivado IP Integrator or Intel Platform Designer, allowing designers to visually construct and configure complex SoC systems. Embedded software can be developed using SDKs provided by the vendors and programmed directly into internal or external memory.

This integration provides a flexible platform for implementing mixed hardware/software systems and supports rapid prototyping of embedded applications.

9.7 Memory-Mapped I/O Implementation

Memory-mapped I/O (MMIO) is a technique used in FPGA-based embedded systems to control peripherals by assigning specific memory addresses to registers. These registers may store control flags, input/output data, or status indicators. The processor or custom logic interacts with peripherals by reading from or writing to these predefined addresses.

Each peripheral is given a unique address space. The host system (e.g., a soft-core processor or memory controller) performs read or write operations over an interconnect bus (e.g., AXI, Avalon) to access the functionality of the peripheral.

9.7.1 Verilog Example

The following Verilog code snippet illustrates a memory-mapped write to a register associated with an LED output:

```
reg [7:0] led_reg;

always @(posedge clk) begin
  if (wr_en && addr == 8'h10)
    led_reg <= wr_data;
end
```

In this example:

- `led_reg` is an 8-bit register used to drive LED outputs.
- The register is written to only when `wr_en` (write enable) is asserted and the target address matches `8'h10`.
- `wr_data` is the data value being written to the register.

This method allows a CPU or a controller to easily manipulate peripheral behavior via simple memory operations. MMIO is a fundamental concept in SoC integration and embedded FPGA development.

9.8 Integrating Verilog with IP-Based Designs

Modern FPGA design flows often combine hand-written Verilog modules with vendor-provided IP blocks using graphical system integration tools. Tools like **Xilinx Vivado IP Integrator** and **Intel Quartus Platform Designer** allow users to build system architectures visually by dragging and connecting IP components, including soft processors, memory controllers, and communication peripherals.

Key features of these tools include:

- **Block-based system construction:** Designers can instantiate IP cores and custom modules as blocks and visually connect them via standard buses like AXI or Avalon.

- **Interface auto-connection:** Tools automatically manage signal naming, handshaking, and arbitration when compatible interfaces are connected. This reduces errors and speeds up integration.

- **Export of HDL wrappers for synthesis:** Once the block design is complete, a top-level HDL wrapper is generated. This file instantiates and connects all system components and can be synthesized along with additional Verilog modules.

To integrate custom Verilog modules into these environments, users typically:

1. Package the module as an IP block with defined ports and metadata.

2. Add the module to the project IP catalog.

3. Instantiate the module graphically and connect it to system components.

This hybrid methodology leverages both the automation of IP-based design and the flexibility of custom HDL coding, enabling efficient development of complex FPGA-based systems.

9.9 Clock and Reset Synchronization

In FPGA-based systems, it is common to have custom modules operating under different clock domains. Proper synchronization techniques are necessary to ensure reliable communication and system stability, particularly in designs involving IP cores, soft processors, and custom logic blocks.

- **Use FIFO or handshake protocols for data transfer:** When crossing from one clock domain to another, data can become corrupted due to metastability. To address this, designers often use dual-clock FIFOs or handshake mechanisms (such as ready/valid protocols) to safely transmit data between modules running at different frequencies.

- **Synchronize reset signals across domains:** Reset signals must be synchronized to the destination clock domain to prevent glitches or partial resets. This is typically done using synchronizer flip-flop chains to reduce metastability risk. Failure to synchronize reset lines may lead to unpredictable system behavior.

Effective clock and reset synchronization is critical for system reliability, especially in larger SoC-style designs where multiple timing domains are present. FPGA design tools often offer IP blocks such as CDC wizards and reset synchronizers to automate and validate these design practices.

9.10 Debugging and Validation Techniques

Debugging and validation are critical stages in the FPGA development cycle, especially when integrating custom Verilog modules with IP cores in complex systems. A combination of real-time observation, internal probing, and communication-based diagnostics is often used to isolate and correct issues.

- **ILA/VIO cores for real-time monitoring:** The Integrated Logic Analyzer (ILA) and Virtual Input/Output (VIO) cores provided by tools like Xilinx Vivado allow internal signal probing and stimulus generation during live operation. These IPs are inserted into the design and accessed through the JTAG interface, providing visibility into register contents, state machine transitions, and timing events.

- **JTAG and UART-based testing:** Using JTAG for boundary scan and in-system programming allows low-level access to internal hardware. UART interfaces can be used for diagnostic printouts, real-time status updates, or sending test commands to the FPGA during validation.

- **Bus analyzers for AXI, SPI:** For systems utilizing standard bus protocols, bus analyzers and protocol checkers help verify the correctness of transactions. These tools check for compliance, monitor traffic patterns, and assist in detecting timing or arbitration issues.

Integrating these debugging tools into the design flow significantly reduces bring-up time and improves system reliability. In practice, designers often maintain debug ports and selectively instantiate ILAs during early hardware testing phases.

9.11 Case Study: UART-Controlled LED Module

This case study demonstrates how to integrate IP cores with custom Verilog logic to build a simple system where an LED array is controlled via UART communication from a PC terminal. This small system illustrates modular design, memory-mapped I/O, and hardware-software integration using FPGA resources.

9.11.1 Components

- **UART Receiver (IP core):** A vendor-supplied UART IP block receives serial data transmitted from a terminal application over USB. The core handles low-level signal synchronization, baud rate timing, and serial-to-parallel conversion.
- **Decoder (Verilog):** This module parses incoming UART data to interpret simple control commands (e.g., ASCII codes). For example, receiving the character 'A' could be interpreted as a command to toggle LED[0].
- **LED Register (Verilog):** The decoded data is stored in a register mapped to an LED array. Any update to this register immediately reflects on the output LEDs, providing visual feedback for received commands.

This modular design enables the UART interface to act as a bridge between human-readable input and FPGA-controlled output hardware, demonstrating a practical and beginner-friendly application of system integration in FPGA development.

9.11.2 Workflow

The following steps outline the design and data flow for controlling LEDs using UART input:

1. **Instantiate UART IP:** The vendor-provided UART receiver is instantiated and configured for the desired baud rate (e.g., 9600 bps). It receives serial input from a terminal through the FPGA's USB-to-UART bridge.
2. **Receive data and decode command:** The received byte from the UART is passed to a custom Verilog decoder module. This module interprets ASCII or binary commands to determine which LED(s) to toggle or update.
3. **Write to LED control register:** Based on the decoded command, a control register is updated. This register directly drives the FPGA's output pins connected to LEDs, thus illuminating or toggling them according to the user input.

Verilog Example: UART Decoder and LED Register

```
module uart_decoder (
    input wire clk,
    input wire rst,
    input wire [7:0] rx_data,
    input wire rx_valid,
    output reg [7:0] led_cmd
```

```
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            led_cmd <= 8'b0;
        else if (rx_valid)
            led_cmd <= rx_data;  // Direct command passthrough
    end
endmodule
```

```
module led_register (
    input wire clk,
    input wire rst,
    input wire [7:0] led_cmd,
    output reg [7:0] led_out
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            led_out <= 8'b0;
        else
            led_out <= led_cmd;  // Drive LEDs based on received
               command
    end
endmodule
```

9.12 Case Study: FPGA-Based Calculator

This case study explores the implementation of a simple calculator system on an FPGA. The design incorporates core digital building blocks such as arithmetic units, a register file, and a controller, with inputs and outputs interfaced through either a UART or an LCD display.

- **ALU (Arithmetic Logic Unit):** The ALU performs fundamental operations such as addition, subtraction, and multiplication. It receives operand inputs from the register file and executes operations based on control signals from the FSM.
- **Register File:** A small multi-register storage unit used to hold operands and results. It allows the controller to load, read, and write data used in computation.
- **UART or LCD Display:** For input/output operations, the system can use a UART interface for serial communication or an LCD module to display operands

and results. UART provides flexibility during simulation and debugging, while LCD is useful for embedded, standalone use cases.

- **FSM-based Controller:** The FSM governs the operation flow of the calculator. It manages states such as data input, operation decoding, ALU execution, result output, and register update.

This calculator project demonstrates the hierarchical integration of multiple functional modules, the use of parameterization for flexible design scaling, and the coordination of datapath and control units in digital design. It is an excellent example of an FPGA-based embedded system with practical user interaction and custom arithmetic computation.

ALU Module

```
module alu (
    input wire [7:0] A,
    input wire [7:0] B,
    input wire [1:0] op,  // 00: Add, 01: Sub, 10: Mul
    output reg [15:0] result
);
    always @(*) begin
        case (op)
            2'b00: result = A + B;
            2'b01: result = A - B;
            2'b10: result = A * B;
            default: result = 16'h0000;
        endcase
    end
endmodule
```

Register File Module

```
module reg_file (
    input wire clk,
    input wire rst,
    input wire we,
    input wire [7:0] data_in,
    output reg [7:0] reg_a,
    output reg [7:0] reg_b
);
    always @(posedge clk or posedge rst) begin
```

```
        if (rst) begin
            reg_a <= 8'd0;
            reg_b <= 8'd0;
        end else if (we) begin
            reg_a <= data_in;
            reg_b <= reg_a;  // Shift previous A to B
        end
    end
endmodule
```

Top-Level Calculator FSM Controller

```
module calculator_top (
    input wire clk,
    input wire rst,
    input wire [7:0] rx_data,
    input wire rx_valid,
    output wire [15:0] result
);
    wire [7:0] A, B;
    reg [1:0] opcode;
    reg we;

    reg_file u_reg (
        .clk(clk),
        .rst(rst),
        .we(we),
        .data_in(rx_data),
        .reg_a(A),
        .reg_b(B)
    );

    alu u_alu (
        .A(A),
        .B(B),
        .op(opcode),
        .result(result)
    );

    always @(posedge clk or posedge rst) begin
        if (rst) begin
```

```
            opcode <= 2'b00;
            we <= 0;
        end else if (rx_valid) begin
            case (rx_data)
                8'hAA: opcode <= 2'b00;  // Add
                8'hBB: opcode <= 2'b01;  // Sub
                8'hCC: opcode <= 2'b10;  // Mul
                default: we <= 1;        // Write data
            endcase
        end else begin
            we <= 0;
        end
    end
endmodule
```

Testbench for FPGA-Based Calculator

The following Verilog testbench simulates the calculator module by applying test vectors through the UART data interface. It performs addition, multiplication, and subtraction using encoded input sequences.

```
`timescale 1ns / 1ps

module calculator_tb;
    reg clk;
    reg rst;
    reg [7:0] rx_data;
    reg rx_valid;
    wire [15:0] result;

    // Instantiate the top calculator module
    calculator_top uut (
        .clk(clk),
        .rst(rst),
        .rx_data(rx_data),
        .rx_valid(rx_valid),
        .result(result)
    );

    // Clock generation
    always #5 clk = ~clk;

```

```
22      initial begin
23          // Initialize inputs
24          clk = 0;
25          rst = 1;
26          rx_data = 0;
27          rx_valid = 0;
28
29          // Reset pulse
30          #10;
31          rst = 0;
32
33          // Load 8 into reg_a
34          send_byte(8'h08);
35          #20;
36
37          // Load 3 into reg_b
38          send_byte(8'h03);
39          #20;
40
41          // Perform Add (Opcode 0xAA)
42          send_byte(8'hAA);
43          #20;
44
45          // Load 5 into reg_a
46          send_byte(8'h05);
47          #20;
48
49          // Load 2 into reg_b
50          send_byte(8'h02);
51          #20;
52
53          // Perform Multiply (Opcode 0xCC)
54          send_byte(8'hCC);
55          #20;
56
57          // Load 9 and 4, then Subtract (Opcode 0xBB)
58          send_byte(8'h09);
59          #20;
60          send_byte(8'h04);
61          #20;
62          send_byte(8'hBB);
```

```
        #20;

        $display("Final Result: %d", result);
        $finish;
    end

    task send_byte(input [7:0] data);
        begin
            rx_data = data;
            rx_valid = 1;
            #10;
            rx_valid = 0;
        end
    endtask

endmodule
```

9.13 Design Guidelines

Effective FPGA design relies on disciplined coding practices and architectural clarity. The following guidelines help create scalable, maintainable, and efficient hardware systems:

- **Use modular coding and naming conventions:** Break down the design into functional blocks, such as ALUs, registers, or controllers. Use consistent naming (e.g., `alu.v`, `reg_file.v`, `fsm_ctrl.v`) to improve code clarity.

```
// Top-level instantiation
alu u_alu (...);
reg_file u_reg (...);
fsm_ctrl u_ctrl (...);
```

- **Separate datapath and control:** Isolate computation logic (datapath) from decision-making logic (control). This enhances reuse and allows separate simulation of each part.

```
// Datapath: ALU performs computation
result = (opcode == ADD) ? A + B :
         (opcode == SUB) ? A - B :
         (opcode == MUL) ? A * B : 16'h0000;

// Control FSM: selects operation
```

```
  case (state)
    IDLE: if (start) next_state = LOAD;
    LOAD: next_state = EXEC;
    EXEC: next_state = DONE;
  endcase
```

- **Use state machines for coordination:** FSMs coordinate complex control flows such as UART transactions or memory-mapped I/O. Define named states using `localparam` and use one-hot or binary encoding.

```
  localparam IDLE = 2'b00, LOAD = 2'b01, EXEC = 2'b10, DONE =
      2'b11;

  always @(posedge clk or posedge rst) begin
    if (rst)
      state <= IDLE;
    else
      state <= next_state;
  end

  always @(*) begin
    case (state)
      IDLE:  if (start) next_state = LOAD;
      LOAD:  next_state = EXEC;
      EXEC:  next_state = DONE;
      DONE:  next_state = IDLE;
    endcase
  end
```

These best practices promote structured design, simplify integration and debugging, and improve overall project scalability for both individual and team-based FPGA development.

9.14 Verification Strategy

A systematic verification strategy ensures that each component of an FPGA design functions correctly and integrates seamlessly into the final system. Key verification steps include:

- **Test each module individually:** Develop unit testbenches for isolated components like the ALU, register file, or decoder. Use assertions and waveform inspection to confirm expected outputs.

```
// Testbench for ALU
initial begin
  A = 8'd10; B = 8'd5; op = 2'b00; #10; // Add
  $display("Result: %d", result);
end
```

- **Use behavioral simulation for top-level tests:** Create a top-level testbench that applies realistic input sequences and monitors the full system behavior. Simulators such as ModelSim or Vivado can be used to visualize signal transitions and validate logic.

```
// Stimulus for top-level FSM
initial begin
  rx_valid = 1; rx_data = 8'hAA; #10; // Select ADD
  rx_data = 8'd7; #10;
  rx_data = 8'd3; #10;
end
```

- **Implement post-synthesis and hardware tests:** After synthesis, verify timing with post-synthesis simulation. Once programmed onto the FPGA, validate design operation using real-world inputs, logic analyzers (ILA), and UART output.
  - Use ILA cores to capture internal signals.
  - Validate GPIO outputs (e.g., LEDs).
  - Monitor serial outputs for correct computation or response.

Combining simulation, synthesis-level validation, and in-system debugging ensures high design confidence and robustness against functional or timing failures.

9.15 Project Organization

Maintaining a well-organized project directory structure is essential for collaboration, scalability, and ease of debugging. A recommended organization is:

- `src/` – Contains all Verilog or VHDL source files, including top-level modules and submodules. Each module should be placed in its own file for clarity.
- `ip/` – Stores vendor-generated IP cores, such as UART controllers, PLLs, or memory blocks. These may include wrapper files, configuration XMLs, or synthesized netlists.

- `tb/` – Contains all testbenches and simulation stimuli files. Subfolders can be used to separate unit tests and top-level system tests.
- `scripts/` – Includes TCL scripts for automating synthesis, simulation, bitstream generation, and project setup. Helps ensure reproducibility and efficient builds.

This hierarchy supports modularity, facilitates version control (e.g., Git), and helps teams maintain consistency across different stages of the FPGA design lifecycle.

9.16 Summary

Verilog and IP core integration enables the efficient design of scalable FPGA systems. Hierarchical modules, parameterization, and memory-mapped interfaces support modularity and reuse. Design environments like Vivado and Quartus streamline IP integration and automation.

Debugging with tools like ILAs and structured project organization enhances verification and maintainability. Overall, mastering these techniques is key to building reliable FPGA-based SoC architectures.

The laboratory exercises for Chapter 9: *System Design and IP Integration* build upon Lab 8: Simple ALU Design and Lab 11: Multiplier Design, both of which are provided in the Appendix section.

9.17 Exercises

1. **Modular Design:** Design a 4-bit ALU in Verilog with modules for AND, OR, ADD, and SUB operations. Use a control input to select the operation.
2. **IP Core Integration:** Use Vivado or Quartus to instantiate a built-in IP core (e.g., a multiplier or UART). Connect it to a simple top-level design and simulate the behavior.
3. **Bus Interface:** Implement a Verilog module that interfaces with a simple bus protocol (e.g., read/write strobe, address, data). Simulate data transfer between a master and a slave module.
4. **Clock Domain Crossing:** Create a Verilog system with two modules operating at different clock frequencies. Use synchronization techniques (e.g., dual flip-flop synchronizer) to safely transfer a control signal across the domains.
5. **Custom IP Packaging:** Package a Verilog module (e.g., a counter or PWM generator) as a reusable IP block in your FPGA design tool. Document its ports and parameterization options.

6. **System Integration:** Build a top-level Verilog design that instantiates and connects a controller module, datapath unit, and memory interface. Use testbench simulation to verify correct operation under various input scenarios.

7. **AXI or Avalon Bus Exploration (Optional):** Research and summarize the basic structure of the AXI or Avalon interface. Describe how IP blocks use these interfaces for communication within an SoC.

Bibliography

[1] C. Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes, 2004.

[2] M. Manohar and J. Bhasker, *Digital System Design Using FPGA: Implementation with Verilog and VHDL*. New York, NY, USA: McGraw-Hill, 2017.

[3] R. Merrick, *Getting Started with FPGAs: Digital Circuit Design, Verilog, and VHDL for Beginners*. San Francisco, CA: No Starch Press, 2023.

[4] S. D. Brown and Z. G. Vranesic, *Fundamentals of Digital Logic with Verilog Design*. New York, NY, USA: McGraw-Hill, 2003.

[5] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[6] A. Tumanov, "IP Reuse: 10 Best Practices for SoC, IC & FPGA Design," IC Manage white paper, Sep. 10, 2024. [Online]. Available: https://www.icmanage.com/ip-reuse-best-practices-soc-ic-fpga-design/

[7] S. Lee, "Mastering FPGA Design with IP Integration," Number Analytics blog, Jun. 23, 2025. [Online]. Available: https://www.numberanalytics.com/blog/ultimate-guide-ip-integration-fpga

[8] Digilent, "Getting Started with IP Integrator in Vivado," Vivado IP Integrator Guide. [Online]. Available: https://docs.amd.com/r/en-US/ug994-vivado-ip-subsystems/Getting-Started-with-Vivado-IP-Integrator

[9] Intel Corporation, "Introduction to Intel® FPGA IP Cores", Quartus Prime Documentation, Sep. 30, 2019. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683102/21-3/introduction-to-cores.html

Chapter 10

Real-World FPGA Projects and Applications

Chapter Objectives

- Learn how FPGAs are used in real-world applications like DSP, AI, and Communication.
- Build practical projects using Verilog and FPGA tools.
- Test and debug complete FPGA-based systems.

10.1 Introduction

FPGAs have evolved from niche academic research tools and development prototypes into indispensable components in a wide array of commercial, industrial, and embedded systems. Initially appreciated for their reconfigurability and utility in digital logic education, FPGAs now serve as high-performance, parallel-processing platforms across domains such as telecommunications, aerospace, automotive, artificial intelligence, and medical instrumentation.

The increasing demand for real-time data processing, low-latency control, and adaptable hardware has positioned FPGAs as a compelling alternative to both ASICs and general-purpose processors. Their architecture, built from CLBs, embedded memory, DSP slices, and programmable I/O, enables tailored acceleration of specific tasks with a high degree of concurrency and determinism.

Modern FPGA development is further empowered by powerful EDA tools and IP core libraries, allowing developers to design complex systems using both hardware description languages like Verilog and graphical SoC integration tools. Additionally, the emergence of hybrid SoC platforms such as Xilinx Zynq and Intel SoC FPGAs has bridged the

gap between software and hardware development by integrating ARM processors with programmable logic fabric.

This chapter presents an in-depth exploration of real-world FPGA applications through a series of case studies. These include DSP, image processing pipelines, motor control systems, network packet processing, and hardware acceleration for AI workloads. Each example highlights not only the architectural advantages of using FPGAs but also design considerations such as timing closure, resource management, and verification.

By examining these practical implementations, readers will gain a deeper understanding of how FPGAs can be leveraged to develop efficient, scalable, and domain-specific hardware solutions in the modern engineering landscape.

10.2 Application Domains of FPGAs

FPGAs are reconfigurable logic devices that offer a unique blend of hardware-level speed and software-like flexibility. Their capacity for parallel processing, deterministic timing, and in-system programmability makes them suitable for a wide range of real-world applications across various industries. Key domains include:

- **Embedded Systems:** FPGAs are often deployed as the core of custom embedded platforms, enabling high-speed control loops, low-latency data processing, and hardware offloading. They integrate seamlessly with microcontrollers or soft-core processors to build real-time systems used in robotics, smart appliances, industrial controllers, and IoT gateways. Their low power consumption and support for partial reconfiguration are advantageous in resource-constrained devices like drones or wearable tech.

- **Telecommunications:** In 5G infrastructure, FPGAs are central to physical-layer (PHY) and MAC-layer acceleration, handling complex modulation, beamforming, and channel coding tasks. Their use extends to optical network switching, baseband processing, packet classification, and hardware acceleration in Software-Defined Networking (SDN) and Network Function Virtualization (NFV). FPGAs' parallelism enables line-rate throughput for protocols like Ethernet, PCI Express (PCIe), and Time-Sensitive Networking (TSN).

- **Automotive and Industrial Automation:** Modern vehicles and factory systems demand reliable, real-time computation. FPGAs support automotive features such as Adaptive Cruise Control, lane detection, and sensor fusion in ADAS systems. In industrial environments, they enable deterministic control for motors, drives, and machine vision systems, while also allowing predictive maintenance through data-driven monitoring. Compliance with ISO 26262 and IEC 61508 standards reinforces

their role in safety-critical applications.

- **Aerospace and Defense:** FPGAs are often deployed in radar systems, avionics, and secure communications. Their support for space-grade features such as Single Event Upset (SEU) mitigation and redundancy make them suitable for satellites and high-radiation environments. Reconfigurable logic also facilitates mission-specific adaptation and secure cryptographic operations during runtime, aligning with cybersecurity and defense protocols.

- **Medical Devices:** Real-time diagnostics and monitoring systems benefit from the low-latency and high-throughput characteristics of FPGAs. Applications include portable ultrasound machines, ECG/EEG monitoring systems, endoscopy imaging, and medical robotics. FPGAs ensure precise signal timing, accurate imaging, and secure patient data processing, often conforming to IEC 60601 standards.

- **Artificial Intelligence and Machine Learning:** As AI moves toward edge and embedded applications, FPGAs are increasingly used to accelerate inference tasks in neural networks. Their reconfigurable fabric supports parallel multiply-accumulate operations, fixed-point optimization, and sparsity-aware computation. In edge devices such as smart cameras, autonomous robots, and surveillance systems, FPGAs enable real-time object detection, image classification, and audio analysis while maintaining energy efficiency.

10.3 Case Study 1: Digital Signal Processing

An illustrative implementation of FIR filter design on FPGA is presented in the work by *Zheng and Wei (2018)*, which demonstrates how parallelism and efficient utilization of DSP slices can significantly enhance performance in digital signal processing applications.

10.3.1 Objective

The goal is to implement a real-time Finite Impulse Response (FIR) filter using FPGA hardware. Target applications include audio equalization, wireless communications, and biomedical signal filtering. FPGAs offer significant advantages such as deterministic timing, pipelining, and parallelism, making them ideal for high-throughput signal processing.

10.3.2 Design Architecture

The design of an efficient FIR filter on FPGA integrates multiple components, each contributing to performance and flexibility. As shown in Figure 10.1, the architecture consists of delay elements, multipliers using DSP slices, and an adder chain.

- **Parallel Multiplication using DSP Slices:**
  Modern FPGAs (e.g., Xilinx and Intel) provide DSP slices that support high-speed multiply-accumulate (MAC) operations. FIR filters use these slices to perform parallel multiplications between incoming samples and filter coefficients, significantly boosting throughput.

- **Shift Registers for Delay Line:**
  FIR filters rely on previous samples. These are stored using shift registers that form a delay line, allowing new samples to enter each cycle and older ones to shift down. This delay structure is aligned with the coefficient set for convolution.

- **Coefficient Storage using BRAM:**
  Filter coefficients (taps) are stored in BRAMs, which allow fast, deterministic access. Coefficients can be dynamically reloaded, supporting adaptive filtering or runtime reconfiguration (e.g., switching between low-pass and band-pass modes).

- **Pipeline Stages for Performance:**
  Pipelining is applied between stages of multiplication and accumulation to meet timing closure at high clock speeds. This approach allows deeper filters without compromising system throughput.

- **Fixed-Point Optimization:**
  To save area and increase performance, fixed-point arithmetic is used instead of floating-point. Word-length is carefully selected to avoid overflow and maintain precision. Saturation and rounding logic are included as needed.

- **Interface Logic:**
  The design interfaces with streaming input data, typically using a clock-synchronous protocol such as AXI4-Stream or valid-ready handshake. Filtered output is produced with minimal latency.

10.3.3 Concept and Structure

An FIR filter produces an output $y[n]$ by performing a weighted sum of the current and past input samples:

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k] \tag{10.1}$$

where:

- $x[n]$ is the input signal,

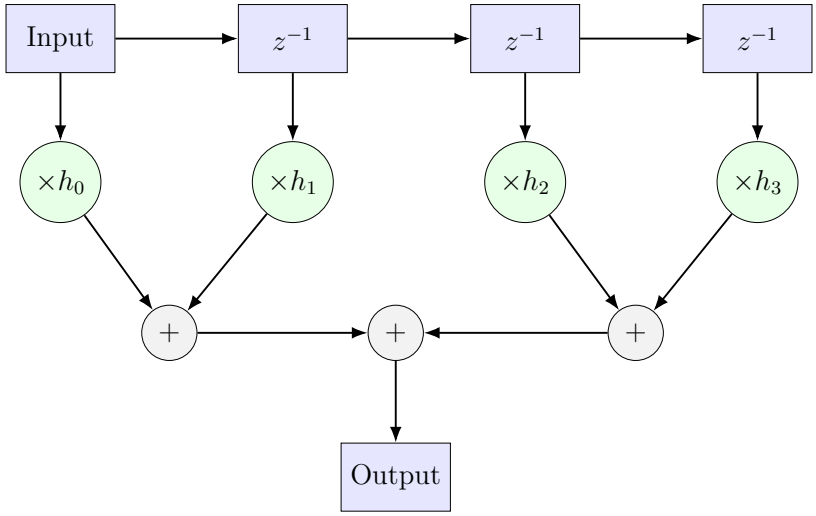


Figure 10.1: FIR filter architecture with delay line and adder chain

- $h[k]$ are the filter coefficients (taps),
- N is the number of taps (filter order).

In FPGA, this structure maps naturally to hardware as a pipeline of multipliers and adders. Each coefficient multiplication and accumulation can be implemented in parallel using DSP slices or logic elements, significantly accelerating performance compared to software-based filtering.

10.3.4 Verilog Example

```
module fir_filter #(
    parameter N = 8  // Number of taps
)(
    input wire clk,
    input wire rst,
    input wire signed [15:0] x_in,              // Input sample
    input wire signed [15:0] coeffs [0:N-1],    // Coefficient
       array
    output reg signed [31:0] y_out              // Output result
);
    reg signed [15:0] shift_reg [0:N-1];        // Delay line
    integer i;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            for (i = 0; i < N; i = i + 1)
                shift_reg[i] <= 0;
```

```
17              y_out <= 0;
18          end else begin
19              // Shift the register
20              for (i = N-1; i > 0; i = i - 1)
21                  shift_reg[i] <= shift_reg[i-1];
22              shift_reg[0] <= x_in;
23
24              // FIR filtering computation
25              y_out <= 0;
26              for (i = 0; i < N; i = i + 1)
27                  y_out <= y_out + shift_reg[i] * coeffs[i];
28          end
29      end
30  endmodule
```

10.3.5 Performance

- **Low Latency:** The FIR filter is designed using a fully pipelined architecture where each stage performs a part of the convolution operation. This enables the system to produce a new output at every clock cycle once the pipeline is filled. The absence of sequential dependencies between filter taps ensures that output generation is not delayed by previous computations, making the filter suitable for real-time and high-speed applications.

- **High Sampling Rate:** By leveraging parallel DSP slices and efficient resource mapping, the FIR filter can operate at clock frequencies in the hundreds of MHz. This makes it capable of handling high-bandwidth signals in communication systems, such as RF and audio processing. The ability to process samples at such rates ensures real-time performance, which is critical in latency-sensitive environments.

- **Configurable Filter Taps:** The design supports parameterized filter length (number of taps), allowing it to be customized for different use cases without structural code changes. For example, increasing the number of taps improves frequency selectivity at the cost of more hardware, while shorter filters are more resource-efficient. This scalability is achieved through the use of Verilog parameters or ‘generate‘ blocks that adjust the internal structure during synthesis.

- **Resource-Efficient Coefficient Management:** Coefficients are stored in Block RAM (BRAM), enabling dynamic reconfiguration. This supports features such as mode switching (e.g., from low-pass to band-pass) and adaptive filtering in systems where coefficients are updated based on external feedback or signal analysis.

- **Timing Closure and Throughput:** The use of pipelining and register balancing ensures that the critical path delay is minimized, which helps achieve timing closure even for filters with a large number of taps. High throughput is maintained due to consistent sample processing and overlapping operations within the pipeline.

10.4 Case Study 2: Image Processing Pipeline

Recent advancements in image processing pipelines implemented on FPGA platforms have explored both traditional and stochastic approaches for edge detection. *Aygün et al. (2017)* proposed a Sobel filter implementation using a stochastic arithmetic-logic unit to improve fault tolerance and power efficiency in hardware-based image processing, while *Huang et al. (2023)* demonstrated the effectiveness of the Canny algorithm for robust edge analysis and its application in real-time visual systems. These studies highlight the growing interest in diverse architectural choices for FPGA-based image enhancement and feature extraction.

10.4.1 Objective

Design and implement a real-time edge detection system on an FPGA using a Sobel filter applied to live video input. The system identifies image boundaries by detecting regions of high spatial frequency—typically corresponding to edges in visual scenes.

10.4.2 System Modules

The real-time edge detection system implemented on FPGA consists of modular blocks that process live video input through a convolution filter and output the enhanced image to a display. Figure 10.2 illustrates the overall pipeline architecture.

- **Camera Interface (DVP/HDMI):** Acquires pixel data from a live video source (e.g., CMOS sensor or HDMI input) and handles synchronization signals (HSYNC, VSYNC, PCLK). The interface converts the serial data stream into frame-buffered pixel data for downstream processing.
- **Line Buffers and Window Generator:** Stores multiple rows of pixel data using FIFO-based shift registers or BRAM to form a sliding 3×3 window. This window is continuously updated and passed to the convolution core.
- **Sobel Filter (Convolution Module):** Applies Sobel operators in horizontal (G_x) and vertical (G_y) directions:

$$G = \sqrt{G_x^2 + G_y^2} \quad \text{or} \quad G \approx |G_x| + |G_y| \tag{10.2}$$

The output emphasizes edges by highlighting regions with high gradient magnitude. The computation is pipelined to process one pixel per clock cycle.

- **Video Output Driver (VGA/HDMI):** Formats the processed image data into RGB signals along with proper sync pulses (HSYNC, VSYNC) for monitor output. Double buffering ensures stable display without flicker or tearing.

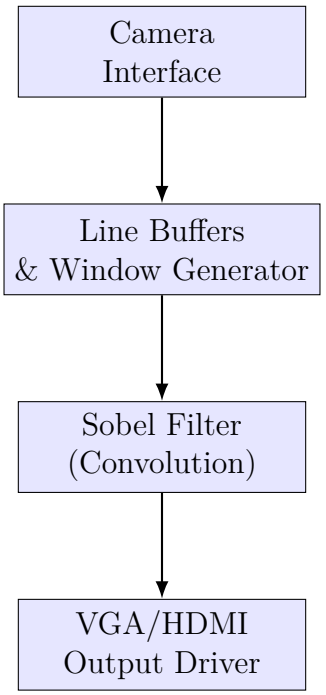


Figure 10.2: Real-time edge detection pipeline on FPGA

10.4.3 Design Challenges

- **Memory Throughput:** High-resolution video requires efficient memory access. Line reuse, partial sum caching, and streaming architectures reduce BRAM bandwidth pressure.
- **Signal Synchronization:** Pixel data must be correctly aligned with frame control signals. If different clock domains exist, CDC mechanisms such as asynchronous FIFOs must be employed.
- **Pipelining for Real-Time Performance:** To sustain high frame rates, the system is deeply pipelined to process a pixel per cycle. Timing closure is achieved by balancing DSP usage with register placement.
- **Edge Accuracy in Noisy Input:** Sobel filters are sensitive to noise. Optional pre-processing with Gaussian filtering improves edge quality but adds complexity to the pipeline.

10.4.4 Verilog Implementation Example

The following Verilog module implements a simplified Sobel filter that computes the gradient using a 3×3 sliding window and approximates the gradient magnitude using $|G_x| + |G_y|$:

```
// sobel_filter.v
module sobel_filter (
    input wire clk,
    input wire rst,
    input wire [7:0] pixel_in,
    input wire valid_in,
    output reg [7:0] pixel_out,
    output reg valid_out
);

    reg [7:0] row0[0:2];
    reg [7:0] row1[0:2];
    reg [7:0] row2[0:2];

    integer i;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            for (i = 0; i < 3; i = i + 1) begin
                row0[i] <= 0;
                row1[i] <= 0;
                row2[i] <= 0;
            end
            valid_out <= 0;
            pixel_out <= 0;
        end else if (valid_in) begin
            row0[0] <= row0[1]; row0[1] <= row0[2]; row0[2] <=
               row1[2];
            row1[0] <= row1[1]; row1[1] <= row1[2]; row1[2] <=
               row2[2];
            row2[0] <= row2[1]; row2[1] <= row2[2]; row2[2] <=
               pixel_in;

            integer gx, gy;
            gx = -row0[0] + row0[2] - 2*row1[0] + 2*row1[2] -
               row2[0] + row2[2];
```

```
33              gy =  row0[0] + 2*row0[1] + row0[2] - row2[0] - 2*
                    row2[1] - row2[2];
34
35              integer g = (gx < 0 ? -gx : gx) + (gy < 0 ? -gy : gy)
                    ;
36              if (g > 255) g = 255;
37              pixel_out <= g[7:0];
38              valid_out <= 1;
39          end else begin
40              valid_out <= 0;
41          end
42      end
43  endmodule
```

10.4.5 Challenges

- **Memory bandwidth optimization:** Real-time video processing requires efficient buffering and pipelining to handle large amounts of pixel data without stalling the system. Techniques such as line buffering using dual-port RAM and data reuse minimize memory contention.
- **Synchronization of data streams:** Proper timing control is needed to align input, processing, and output stages, especially when dealing with multiple clock domains (e.g., camera input clock vs. display output clock). Clock domain crossing (CDC) techniques and handshaking protocols help ensure data integrity.

10.5 Case Study 3: Motor Control System

Recent advancements in FPGA-based motor control systems emphasize the integration of adaptive algorithms and reusable IP cores to enhance real-time responsiveness and design modularity. The work by *Ngo et al. (2024)* presents an FPGA-based adaptive PID controller capable of dynamically tuning control parameters to improve performance under varying load conditions. Complementarily, the studies by *C et al. (2024)* introduce a dedicated FPGA IP core for DC motor control, demonstrating efficient implementation of closed-loop control with minimal latency and high reliability.

10.5.1 Objective

This project implements a PWM-based motor control system for a Brushless DC (BLDC) motor using an FPGA. The main goal is to achieve real-time control of motor speed

using closed-loop feedback from an encoder. This architecture is applicable in robotics, industrial automation, electric vehicles, and precision mechatronics.

10.5.2 Control Strategy Overview

- **PWM Generation:** A high-frequency digital signal modulates voltage applied to the motor. Duty cycle variations control the average power delivered. In FPGA, PWM is implemented with counters and comparators.
- **PID Controller:** Continuously computes a correction based on the error between the desired and actual motor speeds. The PID output dynamically adjusts the PWM duty cycle.
- **Encoder Feedback:** A quadrature encoder tracks shaft movement by producing two signals 90° out of phase. This allows direction detection and high-resolution speed estimation.

10.5.3 PID Controller Formula

Continuous domain:

$$u(t) = K_p e(t) + K_i \int e(t)\,dt + K_d \frac{de(t)}{dt} \tag{10.3}$$

Discrete-time (digital implementation):

$$u[n] = K_p e[n] + K_i \sum_{i=0}^{n} e[i]T + K_d \cdot \frac{e[n] - e[n-1]}{T} \tag{10.4}$$

where:

- $e[n] = r[n] - y[n]$: error between reference input and encoder output
- K_p, K_i, K_d: PID gain constants
- T: sampling interval

10.5.4 FPGA Architecture

- **PWM Generator:** Counter compares against a duty register. Output toggles high when $counter < duty\ value$
- **Encoder Interface:** Decodes quadrature signals (A and B) to determine direction and count. Speed is computed by:
  - Measuring time between pulses, or
  - Counting pulses in a fixed time window

- **PID Block:** Implemented using fixed-point arithmetic for low-latency response. Operates synchronously with PWM updates.
- **Reference Input Block:** Accepts desired speed/position via external switches, UART, or SPI.

10.5.5 Hardware Features

- **PWM Precision:** Adjustable bit-width for fine control (e.g., 8-bit → 256 steps)
- **Quadrature Decoder:** Uses edge detection and XOR logic to interpret encoder direction and steps
- **Velocity Estimator:** Computes motor speed using pulse timing or accumulation counters

10.5.6 Control System Diagram

A basic feedback control system for motor control implemented on an FPGA is illustrated in Figure 10.3. The closed-loop system consists of several key components that work together to regulate the motor's behavior based on a reference input.

- **Ref Input:** Sets the desired motor speed or position.
- **Summing Block (Σ):** Calculates the error between the desired and actual value.
- **PID Controller:** Adjusts the control signal to reduce the error.
- **PWM Generator:** Creates a signal to control the motor's power.
- **Motor:** Moves according to the PWM signal.
- **Encoder:** Measures actual speed or position and sends feedback.

This feedback loop allows the system to automatically correct and maintain accurate motor control.

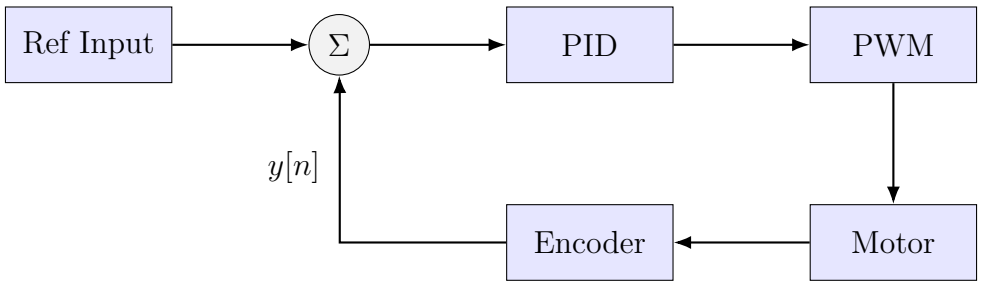


Figure 10.3: Closed-loop FPGA-based motor control with PID and PWM

10.5.7 Benefits

- **Deterministic Timing:** FPGA hardware ensures fixed sampling and update intervals.
- **High-Resolution Control:** Fine-grained PWM and encoder resolution support smooth speed transitions.
- **Hardware Parallelism:** Multiple PID loops can run in parallel for multi-axis systems.
- **Tunable Performance:** Gains and filters can be adjusted at runtime.

Advantages

- **Low latency:** Fast feedback minimizes control lag
- **Precision:** Accurate speed/position tracking from encoders
- **Scalable:** Supports multiple motors in parallel
- **Customizable:** Logic is programmable for unique motor profiles

10.5.8 Implementation: FPGA-Based Motor Control System

10.5.8.1 PID Controller

The PID module calculates the control signal based on setpoint and feedback:

```
module pid_controller (
    input clk, rst,
    input signed [15:0] setpoint, feedback,
    output reg signed [15:0] control_out
);
    parameter signed Kp = 16'sd2, Ki = 16'sd1, Kd = 16'sd1;
    reg signed [15:0] error, prev_error, derivative;
    reg signed [31:0] integral;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            error <= 0; prev_error <= 0; integral <= 0;
               control_out <= 0;
        end else begin
            error <= setpoint - feedback;
            integral <= integral + error;
```

```
            derivative <= error - prev_error;
            control_out <= (Kp * error) + (Ki * integral[15:0]) +
                (Kd * derivative);
            prev_error <= error;
        end
    end
endmodule
```

10.5.8.2 PWM Generator

Generates a PWM signal based on the control signal:

```
module pwm_generator (
    input clk, rst,
    input [7:0] duty_cycle,
    output reg pwm_out
);
    reg [7:0] counter;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            counter <= 0; pwm_out <= 0;
        end else begin
            counter <= counter + 1;
            pwm_out <= (counter < duty_cycle);
        end
    end
endmodule
```

10.5.8.3 Encoder Interface

Tracks motor position from quadrature encoder signals:

```
module encoder_interface (
    input clk, rst,
    input enc_a, enc_b,
    output reg signed [15:0] position
);
    reg [1:0] prev_state;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
```

```
            position <= 0; prev_state <= 2'b00;
        end else begin
            prev_state <= {enc_a, enc_b};
            case ({enc_a, enc_b, prev_state})
                4'b0001, 4'b0111, 4'b1110, 4'b1000: position <=
                    position + 1;
                4'b0010, 4'b0100, 4'b1101, 4'b1011: position <=
                    position - 1;
            endcase
        end
    end
endmodule
```

10.5.8.4 Top-Level Motor Control

Integrates all components into a complete control system:

```
module top_motor_control (
    input clk, rst,
    input enc_a, enc_b,
    input [15:0] setpoint,
    output pwm_out
);
    wire signed [15:0] position, control;

    encoder_interface encoder (
        .clk(clk), .rst(rst), .enc_a(enc_a), .enc_b(enc_b), .
            position(position)
    );

    pid_controller pid (
        .clk(clk), .rst(rst),
        .setpoint(setpoint), .feedback(position),
        .control_out(control)
    );

    pwm_generator pwm (
        .clk(clk), .rst(rst),
        .duty_cycle(control[15:8]),
        .pwm_out(pwm_out)
    );
endmodule
```

10.6 Case Study 4: FPGA as Co-Processor for AI

The role of FPGAs as co-processors for accelerating AI workloads, particularly in training deep convolutional neural networks (CNNs), has gained momentum due to their parallelism and reconfigurability. *Liu et al. (2017)* propose an FPGA-based reconfigurable co-processor architecture tailored for deep CNN training, enabling efficient hardware reuse and on-the-fly adaptability. Similarly, *Clere et al. (2018)* present a dedicated FPGA-based processor that optimizes training performance through customized memory hierarchies and parallel compute pipelines, showcasing the potential of FPGAs in high-performance AI training tasks.

10.6.1 Goal

The objective of this project is to accelerate compute-intensive operations such as matrix multiplications and convolution layers commonly found in Convolutional Neural Networks (CNNs). These operations are offloaded to FPGA hardware, where parallelism and pipelining can significantly outperform general-purpose CPUs for specific tasks.

10.6.2 Architecture

- **Parallel MAC Units:** Multiply-Accumulate (MAC) operations are the core of CNN convolution layers. The FPGA fabric instantiates multiple MAC units in parallel to compute several dot products simultaneously. This architecture allows the execution of multiple output feature maps in one clock cycle, significantly boosting throughput. Pipelining is used to further improve performance.

- **Tiled Memory Access Patterns:** Due to the limited on-chip memory (BRAM), full input images or filters cannot always be stored entirely on the FPGA. Instead, the input data and filter weights are divided into smaller sub-blocks (tiles) that can fit into local memory. These tiles are loaded sequentially while overlapping data reuse is optimized through careful scheduling, reducing the need for costly DRAM accesses.

- **Parameterized Kernel Generator:** A generic and reusable Verilog module is designed to generate convolution kernels based on design-time or runtime parameters such as kernel size (3×3, 5×5), stride, padding, and dilation. This modularity enables flexible integration into a larger AI accelerator framework and supports different CNN models without hardware redesign or re-synthesis.

- **AXI4-Stream Interface:** The data input and output for the co-processor are managed through AXI4-Stream interfaces, allowing seamless integration with ARM

processors (e.g., Zynq SoC) or other AXI-compliant DMA engines. This also enables effective data streaming from external memory to the convolution pipeline.

- **BRAM and DSP Utilization:** The convolution co-processor makes full use of FPGA resources, such as BRAMs for storing input tiles and weights, and DSP slices for fast MAC computation. Resource allocation is optimized for maximum throughput while balancing latency and power consumption.

Figure 10.4 illustrates the architectural design of an FPGA-based CNN co-processor, where input feature maps and weights are streamed via AXI4-Stream interfaces, temporarily buffered in BRAM tiles, processed by a parameterized kernel generator, and computed through a highly parallel MAC array utilizing pipelined DSP slices for accelerated convolution operations.

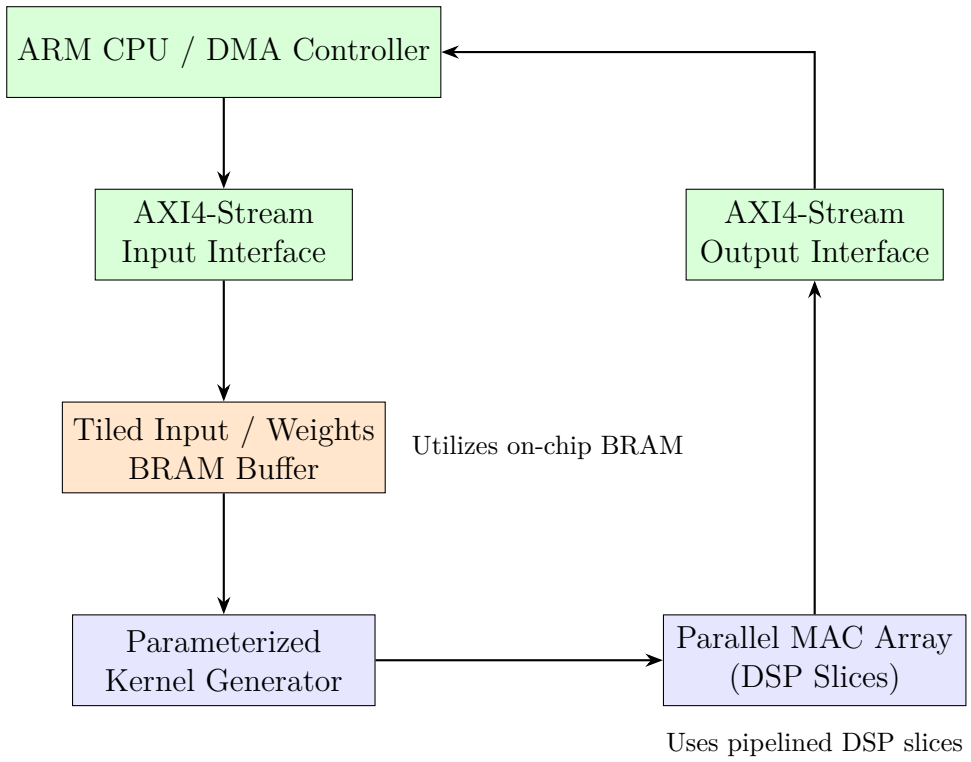


Figure 10.4: Block diagram of FPGA-based CNN co-processor architecture

10.6.3 Performance Advantages

- **High Throughput:** Due to fine-grained parallelism and pipelining, the FPGA implementation achieves higher performance per watt compared to CPU-only processing.

- **Low Latency:** Data is processed as it streams through the pipeline, making it ideal for real-time applications such as video analytics or embedded AI.

- **Flexibility:** Parameterized Verilog allows support for multiple neural network layers without re-synthesizing for each new model, enabling adaptive acceleration.

10.6.4 3×3 Convolution in CNNs

In CNNs, a 3×3 convolution refers to a kernel of size 3 rows by 3 columns that slides over the input feature map. At each position, the kernel performs element-wise multiplication with the overlapping 3×3 region of the input, followed by summation. The result is stored in the output feature map at the corresponding position. This operation is crucial for extracting local spatial patterns such as edges or textures.

A typical 3×3 convolution without padding reduces the spatial dimensions of the input, while with padding (e.g., same padding), the output retains the same size. The stride controls how much the kernel shifts per step, affecting the downsampling rate.

- **Input Feature Map:** The 2D grid of values (e.g., pixel intensities).
- **3×3 Kernel:** The small filter that moves across the input.
- **Output Feature Map:** The result of convolving the kernel over the input.

Figure 10.5 illustrates how a 3×3 kernel convolves with a 5×5 input patch to produce a single output value in the resulting feature map.

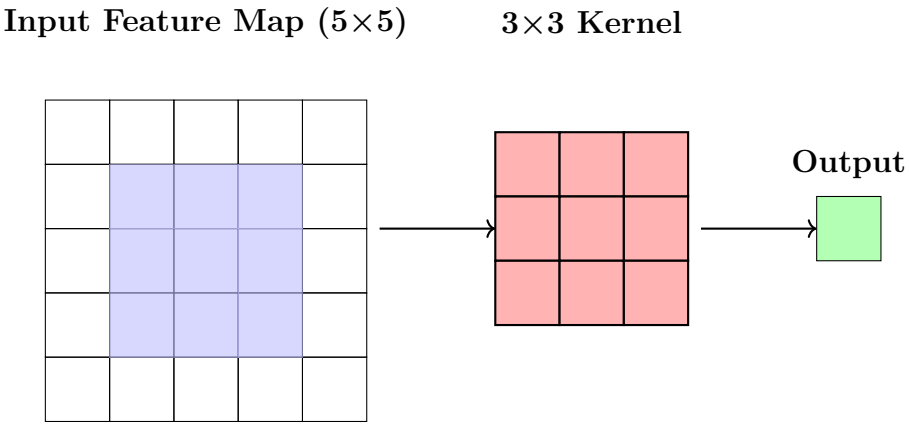


Figure 10.5: A 3×3 convolution applied to a 5×5 input region.

10.6.5 Use Case

This architecture is ideal for edge-based AI applications, such as smart cameras, drones, and robotics, where power and size constraints limit the use of GPUs. The FPGA acts as a co-processor to an embedded CPU, handling compute-heavy CNN layers while the CPU manages scheduling and high-level control.

The following Verilog example demonstrates a parameterized 3×3 convolution kernel designed for use in an FPGA-based CNN accelerator, operating on 8-bit input pixels and weights.

```
module conv2d_3x3 #(
    parameter DATA_WIDTH = 8
)(
    input wire clk,
    input wire rst,
    input wire [DATA_WIDTH-1:0] pixel[0:8],      // Flattened 3x3
       input window
    input wire [DATA_WIDTH-1:0] weight[0:8],     // Flattened 3x3
       kernel weights
    output reg [2*DATA_WIDTH+3:0] result         // Accumulated
       result
);
    integer i;
    reg [2*DATA_WIDTH-1:0] mac_result [0:8];
    reg [2*DATA_WIDTH+3:0] sum;

    always @(*) begin
        for (i = 0; i < 9; i = i + 1) begin
            mac_result[i] = pixel[i] * weight[i];
        end
    end

    always @(posedge clk or posedge rst) begin
        if (rst)
            result <= 0;
        else begin
            sum = 0;
            for (i = 0; i < 9; i = i + 1) begin
                sum = sum + mac_result[i];
            end
            result <= sum;
        end
    end
endmodule
```

10.7 FPGA-Based Wireless Communication Systems

FPGAs have emerged as a crucial technology in wireless communication systems, owing to their inherent reconfigurability, high parallel processing capability, and suitability for low-latency, real-time signal processing. These attributes are particularly advantageous in modern and next-generation wireless standards such as 5G, LTE, Wi-Fi 6/7, and beyond. The increasing complexity and performance requirements of wireless protocols demand flexible hardware platforms capable of supporting multi-standard operations, and FPGAs serve this role exceptionally well.

- **Software-Defined Radio (SDR):** FPGAs are central to SDR implementations, enabling programmable transceivers where digital modulation schemes, filtering operations, and channel coding techniques can be dynamically altered without changing the underlying hardware. This adaptability supports multi-band and multi-protocol functionalities, making FPGAs ideal for cognitive radio and universal wireless platforms.

- **Massive MIMO Systems:** In advanced wireless communication like 5G and 6G, massive MIMO (Multiple Input Multiple Output) systems involve arrays of dozens or even hundreds of antennas. FPGAs efficiently perform tasks such as real-time channel estimation, matrix inversion, beamforming, and precoding by exploiting parallel and pipelined computation architectures.

- **OFDM Modulation and Demodulation:** Orthogonal Frequency Division Multiplexing (OFDM) is a fundamental technique in wireless standards. FPGAs can implement highly optimized FFT/IFFT cores to handle large subcarrier sets. Their pipelined processing nature allows simultaneous handling of multiple OFDM symbols, critical for maintaining throughput and minimizing latency in high-speed communication.

- **Forward Error Correction (FEC):** Robust error correction is crucial in wireless environments prone to fading and interference. FPGA-based implementations of FEC schemes, such as Low-Density Parity-Check (LDPC) codes and Turbo codes, exploit hardware parallelism to deliver high-throughput decoding with minimal delay, making them suitable for real-time applications in 5G and satellite communication.

- **Beamforming and Directional Antenna Control:** Smart antenna technologies require real-time adaptation of radiation patterns based on channel conditions. FPGAs facilitate the implementation of beamforming algorithms, including LMS and RLS adaptive filters, enabling real-time computation of antenna weights and dynamic beam steering for enhanced spatial selectivity and interference suppression.

Benefits of FPGA-Based Wireless Systems:

- **Low Latency:** FPGA architectures allow signal processing operations to be executed with minimal delay, which is critical for time-sensitive wireless applications like ultra-reliable low-latency communication (URLLC) in 5G.
- **Reconfigurability:** Designers can update or switch communication protocols on-the-fly without hardware replacement, enhancing versatility in research and development.
- **High Throughput:** Parallel data paths and pipelined modules allow FPGAs to sustain high data rates required by bandwidth-intensive applications such as video streaming and virtual reality.
- **System Integration:** FPGAs can interface directly with high-speed ADCs/DACs, RF front-ends, and other system components, enabling compact and efficient baseband processing architectures.

10.7.1 Architecture of Wireless System

Evaluating the trade-off between power consumption and performance is crucial in FPGA-based wireless communication systems. *Lorandel et al. (2016)* present a methodology for fast power and performance estimation, enabling efficient design space exploration and optimization of wireless system architectures implemented on FPGAs.

A wireless system architecture using FPGA enables real-time signal processing and flexible control through programmable logic blocks. Figure 10.6 illustrates a detailed architecture of a wireless communication system implemented on an FPGA platform. The design highlights the complete signal flow from reception at the RF antenna, through digital baseband processing, and back to transmission via another RF front-end. This layered representation demonstrates how different FPGA components collaboratively handle real-time signal processing tasks essential for modern wireless communication protocols such as 5G, Wi-Fi, and IoT radio standards.

The system begins with the RF Front End (RX Antenna), which captures analog electromagnetic signals from the air interface. These signals are then converted into digital samples using an Analog-to-Digital Converter (ADC), enabling further processing inside the digital domain. Once digitized, the signal is routed into the FPGA fabric, where high-speed, low-latency computation takes place.

Within the FPGA, several dedicated hardware resources are utilized:

- **DSP Slices** perform MAC operations and filtering functions, which are essential for implementing modulation, demodulation, and channel estimation algorithms.

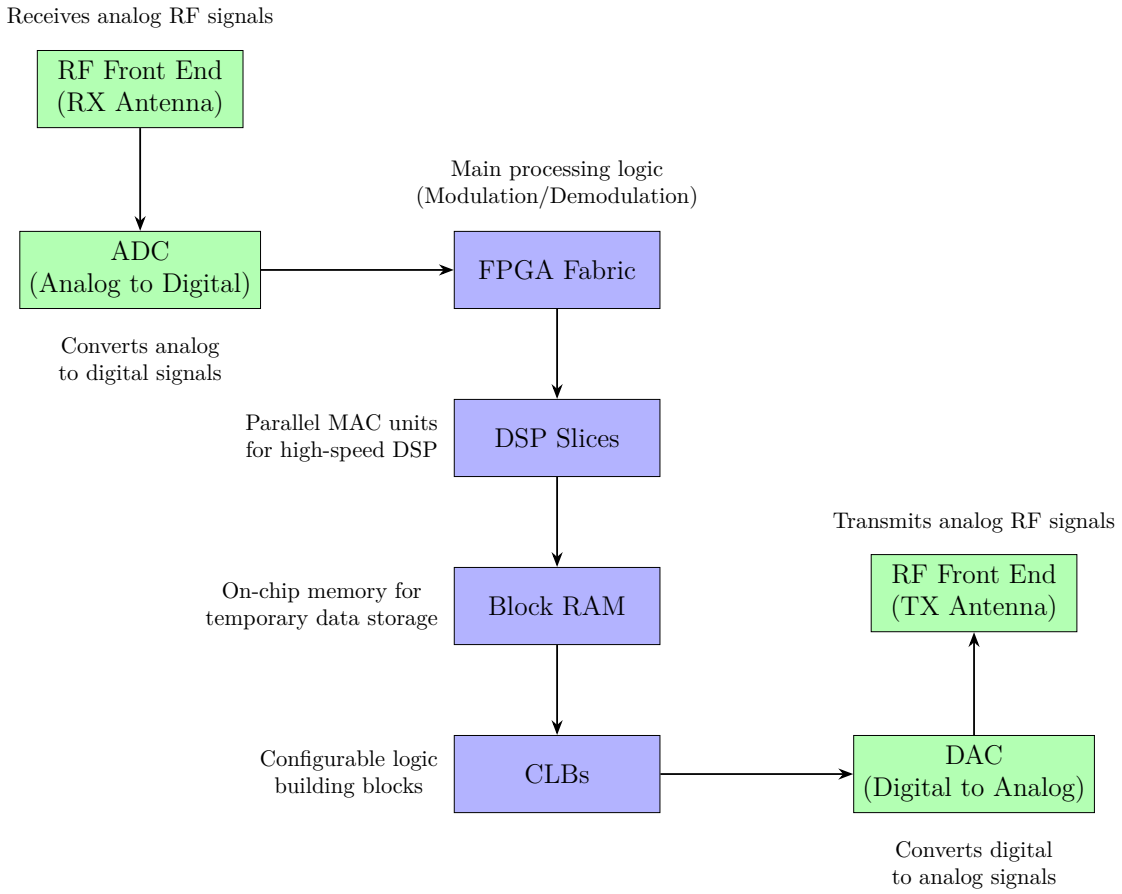


Figure 10.6: FPGA-based wireless communication architecture

- **BRAM** provides on-chip memory for buffering intermediate data, look-up tables, or channel state information.
- **CLBs** enable flexible implementation of control logic, finite-state machines, and custom signal pipelines.

After completing digital baseband processing, the output signal is sent to a Digital-to-Analog Converter (DAC), which reconstructs the analog waveform. This signal is then transmitted via the RF Front End (TX Antenna) into the wireless medium.

This architecture emphasizes the parallel and pipelined nature of FPGA systems, allowing for real-time, low-latency processing of complex wireless protocols. It is also highly modular and reconfigurable, making it ideal for rapid prototyping, SDR, and adaptive systems that must support multiple standards and frequency bands.

10.7.2 Digital-to-Physical Coding Technique for RF Front-End

Advances in programmable electromagnetic structures have enabled the realization of reconfigurable RF front-ends through digital control. *Wan et al. (2016)* demonstrated a field-programmable beam reconfiguring technique based on digitally-controlled cod-

ing metasurfaces, laying the groundwork for intelligent radiation pattern manipulation. With the advent of antenna digitalization, the two-dimensional (2-D) digitally coded pattern for planar antennas, as analyzed by *Tantipiriyakul and Kanjanasit (2023)*, illustrates that a binary chessboard configuration can be represented using bitwise logic operations, where horizontal and vertical stripe arrangements correspond to exclusive OR (XOR) and exclusive NOR (XNOR) functions, respectively. These are realized using an artificial magnetic conductor (AMC)-based unit cell configuration *(Kanjanasit et al., 2023)*. Building on this concept, *Tantipiriyakul and Kanjanasit (2025)* introduced a coded chessboard metasurface antenna with binary defects, which further enhances anomalous beam-steering capabilities for dynamic RF applications.

The digital-to-physical coding technique for RF front-end design represents a modern design paradigm where digital logic directly controls or defines the physical configuration and electromagnetic behavior of reflector and antenna structures. Unlike traditional methods that rely on fixed geometries and passive components, this approach enables reconfigurable, programmable, and intelligent antenna systems that respond dynamically to input signals or environmental conditions.

In this concept, digital inputs—often in the form of binary or multi-bit codes—are used to define the operational state of physical elements such as patches, metasurface unit cells, or impedance-tunable components. These digital control signals are processed by embedded systems using FPGA, and are then mapped to physical configurations capable of modifying antenna behavior in real time.

To illustrate the fundamental structure of a digitally programmable coding metasurface, Figure 10.7 presents a standard 5×5 binary chessboard-coded layout. In this physical configuration, the digital states alternate systematically between 1 and 0, forming a high-contrast grid pattern that reflects binary phase coding. Each square represents a unit cell with a discrete coding value, forming the basis for programmable electromagnetic responses.

Figure 10.8 presents four representative physical configurations of a 5×5 binary chessboard-coded metasurface, each exhibiting a deliberate flipped-bit defect. The top row shows defects introduced at positions $(0, 1)$ and $(0, 3)$, while the bottom row illustrates similar defects at positions $(4, 1)$ and $(4, 3)$. These defect locations simulate structural irregularities and serve as key design variations for evaluating the influence of local coding perturbations on far-field radiation behavior.

System Layers

1. Digital Control Layer

- Implements control logic using FPGA or digital processor.

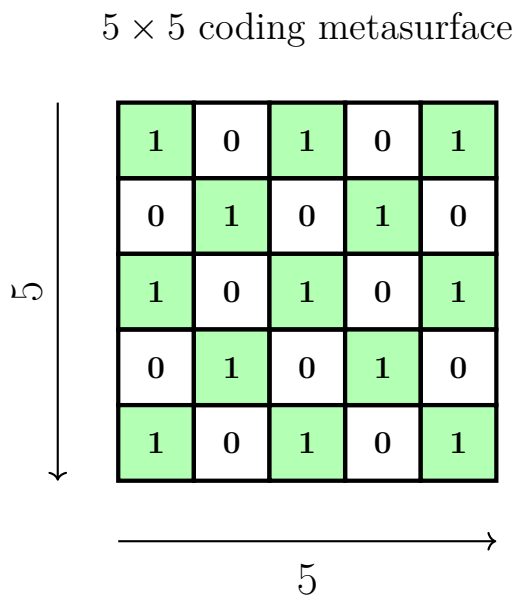


Figure 10.7: Binary chessboard-coded metasurface with alternating digital states (1s and 0s)

1	1	1	0	1
0	1	0	1	0
1	0	1	0	1
0	1	0	1	0
1	0	1	0	1

(a) Defect at (0,1)

1	0	1	1	1
0	1	0	1	0
1	0	1	0	1
0	1	0	1	0
1	0	1	0	1

(b) Defect at (0,3)

1	0	1	0	1
0	1	0	1	0
1	0	1	0	1
0	1	0	1	0
1	1	1	0	1

(c) Defect at (4,1)

1	0	1	0	1
0	1	0	1	0
1	0	1	0	1
0	1	0	1	0
1	0	1	1	1

(d) Defect at (4,3)

Figure 10.8: Four representative cases of chessboard-coded metasurfaces with binary defects

- Generates discrete control signals (e.g., 0 or 1) based on user-defined logic or AI algorithms.

2. Interface and Mapping Layer

- Digital codes are translated into physical control actions using switching circuits or DACs.
- Lookup tables or digital control algorithms determine how each digital input maps to a physical configuration.

3. Physical Configuration Layer

- Includes antenna elements (e.g., patches, dipoles, metasurface cells) with reconfigurable features.
- Control mechanisms such as PIN diodes, varactors, MEMS switches, or tunable substrates alter the structure's radiation properties.

Example Use Case: Binary-Coded Metasurface

A typical application involves a binary-coded metasurface where each unit cell is assigned a digital value:

- Code '0': Unit cell reflects incident wave with 0° phase shift.
- Code '1': Unit cell reflects with 180° phase shift.

By programming an array of such elements using digital logic, designers can create phase gradients that steer or shape beams without physical motion.

Advantages of Digital-to-Physical Approach

The digital-to-physical approach offers several key advantages for reflector and antenna design. It enables on-demand reconfigurability of radiation characteristics, allowing the antenna to adapt to varying communication needs. This method supports compact and integrated implementations by combining digital logic and antenna elements on a single PCB. Additionally, it facilitates intelligent behavior through machine-learning-based adaptation and cognitive radio functions. The system's programmability also allows real-time control and updates via software or firmware, making it highly flexible and future-ready.

The digital-to-physical system connects digital control with physical behavior, making RF front-end design more flexible and software-driven. It enables smart, adaptable, and compact wireless systems for modern communication and sensing needs.

10.8 System Integration Projects

10.8.1 SoC-Based Audio Recorder

This project integrates digital signal acquisition, embedded control, and non-volatile storage into a cohesive audio recording system using an FPGA.

- **Soft-core processor (MicroBlaze/Nios II):** Acts as the main controller of the system. It runs a lightweight embedded software that handles control flow, user commands, file management, and communication with peripheral devices.
- **ADC interface:** An analog-to-digital converter (ADC) module is used to digitize incoming analog audio signals. The sampled audio data is buffered in block RAM and passed to the processor or DMA controller.
- **SD card storage using SPI:** The system stores digitized audio data into an SD card using a Serial Peripheral Interface (SPI). A FAT filesystem IP or custom logic can be used to manage file access and data logging.

This project demonstrates how a soft processor and hardware logic can work together for data acquisition and storage in embedded systems.

10.8.2 Gesture-Controlled Robot

This application showcases real-time control using sensor-based input and FPGA-implemented motor logic.

- **Accelerometer sensor input:** A 3-axis accelerometer (e.g., ADXL345) provides gesture data. The FPGA reads acceleration values over I2C or SPI and decodes specific movement patterns like tilting or shaking.
- **FSM-based motor control logic:** A Finite State Machine (FSM) implemented in Verilog interprets decoded gestures and generates PWM signals to control motor direction and speed for robot movement.
- **Bluetooth module interface:** A Bluetooth module (e.g., HC-05) connects the gesture sensor to the FPGA wirelessly. The received data is parsed and used to trigger robot actions, enabling remote and intuitive control.

This project exemplifies multi-interface integration and real-time responsiveness in FPGA-based robotics.

```
// Simple FSM-based Motor Control Module (used in Gesture-Controlled Robot)
module motor_fsm(
    input wire clk,
    input wire rst,
    input wire [1:0] gesture_cmd,  // 2-bit command: 00-stop, 01-forward, 10-left, 11-right
    output reg motor_left_en,
    output reg motor_right_en
);
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            motor_left_en <= 0;
            motor_right_en <= 0;
        end else begin
            case (gesture_cmd)
                2'b00: begin // Stop
                    motor_left_en <= 0;
                    motor_right_en <= 0;
                end
                2'b01: begin // Forward
                    motor_left_en <= 1;
                    motor_right_en <= 1;
                end
                2'b10: begin // Turn Left
                    motor_left_en <= 0;
                    motor_right_en <= 1;
                end
                2'b11: begin // Turn Right
                    motor_left_en <= 1;
                    motor_right_en <= 0;
                end
                default: begin
                    motor_left_en <= 0;
                    motor_right_en <= 0;
                end
            endcase
        end
    end
endmodule

```

```
// SPI-Based SD Card Write Interface (simplified write trigger interface)
module sd_spi_writer(
    input wire clk,
    input wire rst,
    input wire start_write,
    input wire [7:0] audio_data,
    output reg spi_mosi,
    output wire spi_clk,
    output reg spi_cs
);
    // Simplified SPI writer (details like timing, buffer, FSM omitted)
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            spi_cs <= 1;
            spi_mosi <= 0;
        end else if (start_write) begin
            spi_cs <= 0;
            // Assume SPI shift register exists here
            spi_mosi <= audio_data[7]; // Example only
        end else begin
            spi_cs <= 1;
        end
    end
endmodule
```

10.9 Design Challenges in Real Projects

FPGA development in real-world scenarios extends far beyond writing Verilog code and simulating logic. Designers must navigate a range of practical engineering challenges to meet design goals related to timing, power, resource utilization, manufacturability, and cost. This section outlines several of the most common design hurdles encountered in FPGA projects.

10.9.1 Timing Closure

Timing closure refers to the process of ensuring that all combinational logic paths meet setup and hold timing requirements at the target clock frequency. In large or high-speed designs, closing timing is one of the most difficult tasks. Critical considerations include:

- **Critical Path Optimization:** Identify paths with the longest delays and optimize logic depth, pipelining, or technology mapping to reduce propagation time.
- **Floorplanning:** Manually constrain or guide placement of logic blocks to minimize interconnect delays. Use Pblocks in Vivado to localize critical modules.
- **Clock Domain Crossing (CDC):** Synchronize signals crossing clock domains using FIFOs or double-flop synchronizers to avoid metastability.
- **Constraint Management:** Apply proper constraints in XDC/SDC files, including clock definitions, false paths, and multicycle paths.

10.9.2 Power Management

Power efficiency is increasingly important for FPGA applications in mobile, automotive, and industrial domains. Managing dynamic and static power helps extend battery life and reduce thermal concerns. Techniques include:

- **Clock Gating:** Disable or mask clocks for unused or idle blocks to reduce switching activity.
- **Resource Sharing:** Time-multiplex arithmetic units (e.g., DSP slices) across multiple operations to lower utilization.
- **Low-Power Modes:** Use vendor-specific low-power IP and sleep states (e.g., Xilinx Power-Down control).
- **Voltage Scaling:** Implement multi-voltage domains if supported, and operate logic blocks at the minimum required voltage.

10.9.3 Hardware Debugging

Post-synthesis and in-system debugging are critical for resolving timing violations, logic errors, and functional mismatches. Real hardware behaves differently from simulations due to delays and asynchronous events. Effective strategies include:

- **ILA (Integrated Logic Analyzer):** Insert ILA cores in Vivado to capture internal signal activity during runtime.
- **VIO (Virtual I/O):** Dynamically drive or monitor signals such as resets, flags, and configuration bits.
- **UART/USB Debug Prints:** Send debug messages over serial to assist in state machine or protocol verification.

- **Oscilloscopes and Logic Probes:** Use physical tools for monitoring high-speed I/Os or verifying signal integrity.

10.9.4 IP Licensing and Toolchain Constraints

When using IP cores provided by FPGA vendors or third parties, licensing models can introduce restrictions in development, simulation, and deployment:

- **License Types:** Some IPs are available only in encrypted form or evaluation mode (time-limited or feature-limited).
- **Synthesis Limitations:** Unlicensed IP may not allow bitstream generation, preventing physical implementation.
- **Tool Compatibility:** Ensure IP cores are compatible with specific versions of Vivado or Quartus; upgrading toolchains may require IP regeneration.
- **Export Compliance:** In some cases, using or distributing certain IPs may be restricted by export laws or regional limitations.

Successfully navigating these challenges requires not only proficiency in HDL design but also in constraint management, floorplanning, verification methodology, and the nuances of FPGA development toolchains.

10.10 Collaboration with Software

FPGAs are increasingly used in systems where hardware accelerators interact closely with embedded or host software. This tight coupling requires effective hardware-software collaboration, allowing software to configure, monitor, and manage the operation of FPGA-based subsystems. The design methodology that facilitates this integration is known as hardware-software co-design.

10.10.1 Hardware-Software Co-design

Hardware-software co-design refers to the concurrent development of hardware components (e.g., accelerators or peripherals) and the software that interfaces with them. It ensures seamless communication between a processor (e.g., ARM core, MicroBlaze, Nios II) and custom FPGA logic.

Key integration mechanisms include:

- **AXI-Lite Interface:** This lightweight subset of the AXI protocol is commonly used to expose internal control/status registers to software. Through AXI-Lite, a

processor can write to or read from specific memory-mapped registers to control FPGA modules.

- **Memory-Mapped Control:** Custom logic often includes internal registers accessible via mapped addresses in the processor's memory space. Software can configure parameters (e.g., matrix size, filter coefficients), initiate operations, or poll for status.
- **AXI4-Stream for Dataflow:** In high-throughput systems, AXI4-Stream enables continuous data transfer between logic blocks and software-managed DMA engines without handshaking overhead.
- **Shared BRAM or DDR Interface:** FPGA and processor can share access to a memory block (e.g., DDR or on-chip RAM), enabling bulk data exchange such as video frames, sensor data, or AI tensors.
- **Interrupt Signaling:** Hardware can raise interrupts to inform the processor about task completion, errors, or status changes. This improves responsiveness without constant polling.

10.10.2 Driver Development

Software drivers are necessary to abstract the hardware functionality and expose a usable interface to applications. Drivers ensure safe and correct operation of the hardware modules by handling initialization, data transfer, and synchronization.

Types of drivers include:

- **Bare-Metal Drivers:** These are directly coded firmware routines that access hardware registers without an OS. Often used in real-time or minimal systems, such drivers provide precise control over timing and latency.
- **Linux Device Drivers:** For systems running embedded Linux (e.g., Xilinx Zynq or Intel SoC FPGAs), custom kernel modules or user-space drivers interact with FPGA peripherals through /dev interfaces. These drivers manage memory mapping, interrupts, and I/O control.
- **Auto-Generated HAL:** Tools like Xilinx Vitis or Intel Platform Designer often provide templates or generate hardware abstraction layers (HALs) that simplify the development of application code. These libraries encapsulate register operations in high-level functions.
- **DMA Controller Drivers:** When dealing with large datasets, such as in video or AI pipelines, direct memory access (DMA) is used to move data between RAM and FPGA accelerators. Drivers manage DMA buffers and transaction sequences.

- **Interrupt Handling:** Drivers implement interrupt service routines (ISRs) to respond to events generated by the FPGA logic (e.g., computation done, error condition). This event-driven architecture enhances performance and efficiency.

10.10.3 Best Practices for Integration

- Design clean register maps and document them thoroughly.
- Use simulation and co-simulation tools to test driver-hardware interaction early.
- Validate the software interface with post-implementation simulations or in-system debugging (e.g., using ILA).
- Maintain consistent naming conventions and modularize both HDL and software for maintainability.
- Employ standardized interfaces like AXI4, Avalon, or AMBA to ensure compatibility with IP blocks and tools.

10.10.4 Example Scenario: Hardware Accelerator for Image Filtering

- FPGA implements a 2D convolution module.
- Control registers (filter size, enable bit, status flag) are exposed over AXI-Lite.
- Image frames are streamed via DMA into on-chip buffers.
- A Linux driver maps the AXI-Lite interface into user space and initiates the operation.
- Upon completion, the FPGA raises an interrupt and the driver copies results to user memory.

Conclusion: Effective hardware-software collaboration maximizes FPGA utility, enabling the creation of intelligent, flexible, and high-performance systems. Whether in edge devices, real-time control systems, or embedded AI platforms, mastering co-design and driver development is crucial for system-level success.

10.11 Educational FPGA Projects

Educational FPGA projects offer hands-on experience with digital design concepts, HDL coding, simulation, synthesis, and debugging. These projects are ideal for students,

hobbyists, and beginners to bridge theory and practice. Below are several classic and engaging FPGA project examples with brief descriptions:

- **Reaction Timer Game:** This project challenges the user to press a button as quickly as possible after an LED lights up. It teaches finite state machine (FSM) design, clock division, debouncing input switches, and timing control. Useful for understanding latency measurement and human interface interaction.

- **Digital Stopwatch:** A digital stopwatch design involves counting time units and displaying the output on 7-segment LEDs or an LCD screen. The project covers binary counters, multiplexed display driving, and modular Verilog coding. Optional extensions include pause/resume and split-time features.

- **Traffic Light Controller:** This classic FSM-based design simulates a real-world intersection with red, yellow, and green lights. Students learn about timing sequences, state transitions, and conditional logic. Advanced versions may incorporate pedestrian crosswalk buttons or sensors.

- **VGA Drawing Pad:** A user-controlled interface (buttons, switches, or PS/2 mouse) allows drawing pixels on a VGA monitor. This project introduces VGA timing signals, video memory buffering, coordinate mapping, and signal synchronization, offering an excellent primer on video signal generation and display systems.

- **MIDI Synthesizer:** This project receives MIDI messages via UART and produces digital audio tones via PWM or DAC output. It highlights serial communication decoding, wave generation, tone synthesis, and real-time response. It's great for introducing digital signal generation and audio interfacing.

Learning Outcomes

These projects help reinforce fundamental concepts such as:

- Clocking and timing constraints
- FSM and control logic design
- Verilog-based modular development
- Signal interfacing (GPIO, UART, VGA)
- Simulation and testbench development

Recommended Tools and Platforms

- Xilinx Vivado with Basys 3 / Nexys A7
- Intel Quartus Prime with DE10-Lite or DE1-SoC
- Simulation tools: ModelSim, Vivado Simulator, GTKWave

These FPGA projects not only build technical skills but also enhance problem-solving, creativity, and confidence in digital hardware development.

10.12 Prototyping with Development Boards

Prototyping with FPGA development boards is a vital step in bridging simulation with real-world deployment. These boards offer a practical hardware platform where students, researchers, and engineers can test and debug digital designs in real time. Equipped with programmable FPGAs, standard peripherals, and debugging interfaces, they enable rapid development, iteration, and integration of both academic and industrial projects.

Popular FPGA Development Boards

- **Digilent Basys 3 (Xilinx Artix-7):** Designed for educational purposes, Basys 3 features the Artix-7 FPGA (XC7A35T). It includes onboard components such as 16 switches, 16 LEDs, 5 buttons, and 4-digit seven-segment displays. It is ideal for learning combinational and sequential logic, FSMs, and basic processor design using Vivado Design Suite.
- **DE10-Lite (Intel MAX10):** This board includes the MAX10 FPGA and is supported by Intel Quartus Prime. It offers onboard SRAM, LEDs, switches, an accelerometer, and support for Arduino shields. DE10-Lite is suitable for low-power applications, real-time interfacing, and basic signal processing designs.
- **ZedBoard (Xilinx Zynq-7000 SoC):** Combines a dual-core ARM Cortex-A9 processor with programmable logic, enabling true hardware-software co-design. It includes peripherals such as DDR3, HDMI, USB, Ethernet, and GPIO headers. ZedBoard supports Linux-based embedded system development and is widely used in advanced robotics, AI acceleration, and multimedia applications.

Onboard Peripherals and Interfaces

Most development boards include standard I/O and multimedia peripherals, allowing direct interfacing with the environment:

- **Switches and LEDs:** Used for basic I/O interaction, testing logic states, or simulating control inputs.
- **Seven-Segment Displays:** Useful for displaying numerical results, status codes, or timer outputs driven by HDL modules.
- **VGA/HDMI Output:** Enables video signal generation for projects involving image processing, GUI development, or gaming systems.
- **UART and SPI Interfaces:** Facilitate serial communication with PCs or other embedded devices for debugging or data transfer.
- **Pmod and Arduino Headers:** Allow expansion using sensors, actuators, and third-party modules for IoT, robotics, and control applications.

Educational and Industrial Relevance

- **Academic Projects:** Ideal for lab experiments, course projects, and capstone designs. Students can implement counters, FSMs, ALUs, microprocessors, and peripheral interfaces on actual hardware.
- **Research Prototyping:** Researchers prototype algorithms such as machine learning accelerators, digital filters, and control systems before moving to ASIC or SoC design.
- **Industrial Applications:** FPGAs on development boards enable rapid prototyping of control systems, communication interfaces, and signal processing pipelines before deployment to custom hardware platforms.

Advantages of FPGA-Based Prototyping

- Short development cycles with immediate hardware feedback
- Real-time validation of HDL designs with physical I/O
- Integration of hardware, software, and IP cores on a single board
- Access to vendor debugging tools such as ILA, VIO, and hardware managers

In conclusion, development boards provide a versatile and cost-effective foundation for experimenting with digital systems, accelerating the learning process, and translating theory into practical applications.

10.13 Project Lifecycle

The development of FPGA-based systems follows a structured lifecycle that ensures systematic progress from concept to hardware realization. This lifecycle encompasses several key stages that align with engineering best practices and digital design methodologies.

1. **Requirement Analysis**
   This initial phase involves gathering and analyzing the functional and non-functional requirements of the system. This includes defining the purpose of the design (e.g., signal processing, control system, embedded computation), identifying performance targets (latency, speed, power), input/output specifications, and any hardware/software co-design constraints. The result is a clear understanding of what needs to be built and the design space in which it must operate.

2. **Specification and System Architecture Design**
   Designers translate requirements into detailed specifications. High-level block diagrams are created to illustrate system architecture. This includes identifying major functional blocks (e.g., ALU, controllers, memory interfaces), defining data and control flow, selecting communication protocols (e.g., AXI, SPI, UART), and outlining clock/reset schemes. Hardware/software partitioning decisions are made here, especially in SoC designs.

3. **HDL Development and IP Integration**
   Functional blocks are developed using Hardware Description Languages such as Verilog or VHDL. Reusable IP cores (for UART, FIFO, memory controllers, etc.) are integrated using vendor tools like Xilinx IP Integrator or Intel Platform Designer. This stage emphasizes modular design, use of parameters, and hierarchical modeling to create scalable, maintainable codebases.

4. **Simulation and Functional Verification**
   Testbenches are created to simulate each module and verify their correctness under expected and corner-case conditions. RTL simulation tools (e.g., Vivado Simulator, ModelSim) are used to visualize waveforms and debug logic. Assertions and coverage metrics may also be applied. Verification at this stage avoids costly errors in later synthesis and implementation.

5. **Synthesis and Implementation**
   In the synthesis step, HDL code is translated into a gate-level representation that maps to the FPGA's configurable logic elements (CLBs, LUTs, DSP slices). The implementation step (place and route) allocates physical resources and routes connections, taking into account timing constraints, clock domains, and placement

optimization. Reports such as timing summary, resource utilization, and routing congestion are analyzed.

6. **Bitstream Generation and FPGA Programming**
   Once implementation is complete and timing is met (i.e., timing closure), the design is compiled into a bitstream file (.bit or .sof). This file is downloaded to the FPGA through tools like Vivado Hardware Manager or Intel Programmer. The system is then powered up and validated using real I/O and peripherals.

7. **Testing and Hardware Debugging**
   Hardware testing is critical for validating the system in a real-world environment. Designers use Integrated Logic Analyzer (ILA), Virtual I/O (VIO), and UART-based print statements to inspect internal signals during runtime. Debugging helps verify correct operation and uncover issues like metastability, protocol mismatches, or timing violations.

8. **Documentation and Demonstration**
   Proper documentation includes the project specification, source code structure, testbench results, synthesis reports, and a user guide for setup and operation. For academic or industry deliverables, a demonstration is prepared using a physical setup showing real-time functionality, input/output handling, and expected performance metrics.

This structured lifecycle promotes design clarity, minimizes errors, and ensures reproducibility, particularly for large-scale or collaborative FPGA development projects. Adopting such a methodology is crucial for delivering reliable and maintainable digital systems.

10.14 Summary

FPGA-based projects open diverse opportunities in fields such as digital signal processing, embedded system control, telecommunications, and artificial intelligence. By leveraging the reconfigurability of FPGAs and the power of Verilog HDL, designers can implement tailored, high-performance hardware architectures that meet specific application demands.

This chapter showcased real-world case studies—including FIR filters, motor controllers, image processing pipelines, and AI accelerators—to highlight the versatility of FPGA platforms. System-level integration through IP cores, soft-core processors, and standard interfaces like AXI further streamlines development.

Mastering FPGA design flows—from simulation to hardware debugging—not only builds core engineering skills but also prepares designers for industry-ready prototyping and deployment of advanced digital systems.

The laboratory exercises for Chapter 10: *Real-World FPGA Projects and Applications* include Lab 14: UART Communication and a revisit of Lab 13: Traffic Light Controller, both of which are provided in the Appendix section.

10.15 Exercises

1. **Mini UART Transmitter:** Design a UART transmitter module in Verilog. Set a baud rate of 9600 and send an 8-bit data frame. Create a testbench to simulate the transmission.

2. **Traffic Light Controller:** Implement an FSM for a 3-way traffic light using Verilog. Include state timing using a counter and provide a behavioral testbench for simulation.

3. **Digital Stopwatch:** Build a stopwatch that starts, stops, and resets using button inputs. Display the count on a 7-segment display (or simulation output).

4. **PWM Generator:** Create a PWM generator with adjustable duty cycle. Simulate varying the duty cycle using input switches or testbench stimuli.

5. **Image Threshold Filter (Basic DSP):** Design a simple module that applies a binary threshold to an 8-bit grayscale pixel stream. Simulate the module using sample input data.

6. **Sensor Data Aggregator:** Simulate the logic of collecting and averaging 8-bit sensor readings. Implement this using shift registers and accumulators.

7. **RTC Counter:** Design a clock counter module that keeps track of seconds, minutes, and hours. Implement carry logic for rollovers and simulate a full 24-hour cycle.

8. **Project Case Study Presentation:** Research and summarize a real-world FPGA application (e.g., radar processing, autonomous vehicles, or cryptocurrency mining). Explain which FPGA resources are used and how Verilog contributed to the solution.

9. **Deploy on FPGA Board (Optional/Bonus):** Take one of the above designs (e.g., UART or stopwatch) and implement it on a physical FPGA development board. Document the constraints, pin assignments, and implementation steps.

Bibliography

[1] C. Huang, G. Wu, Z. Wang, and S. Song, "Research on image edge analysis and application based on Canny algorithm," in *Proc. IEEE Int. Conf. Image Process. and Computer Applications (ICIPCA)*, Changchun, China, 2023, pp. 1548–1557, doi: 10.1109/ICIPCA59209.2023.10257832.

[2] J. Lee, J. He, and K. Wang, "FPGA-based neural network accelerators for millimeter-wave radio-over-fiber systems," *Opt. Express*, vol. 28, no. 9, pp. 13384–13400, Apr. 2020, doi: 10.1364/OE.391050.

[3] J. Lorandel, J.-C. Prévotet, and M. Hélard, "Fast power and performance evaluation of FPGA-based wireless communication systems," *IEEE Access*, vol. 4, pp. 2005–2018, 2016, doi: 10.1109/ACCESS.2016.2559781.

[4] J. Zheng and Z. Wei, "FIR filter design based on FPGA," in *Proc. 10th Int. Conf. Measuring Technology and Mechatronics Automation (ICMTMA)*, Changsha, China, 2018, pp. 36–40, doi: 10.1109/ICMTMA.2018.00016.

[5] K. Kanjanasit, P. Osklang, T. Jariyanorawiss, A. Boonpoonga, and C. Phongcharoenpanich, "Artificial magnetic conductor as planar antenna for 5G evolution," *Computers, Materials & Continua*, vol. 74, no. 1, pp. 503–522, 2023, doi: 10.32604/cmc.2023.032427.

[6] M. Vaithianathan, N. S. Frank, M. Patil, M. Reddy, S. Rajasekaran, and M. U. Krishnan, "FPGA-based adaptive beamforming system for improved wireless communication performance," in *Proc. Asian Conf. Intelligent Technologies (ACOIT)*, Kolar, India, 2024, pp. 1–6, doi: 10.1109/ACOIT62457.2024.10939953.

[7] N. C., S. S. Chauhan, C. Paramasivam, and V. P. Meena, "FPGA IP core for DC motor control with adaptive neural network PID tuning, and high-resolution encoder interface," *IEEE Access*, vol. 12, pp. 169356–169369, 2024, doi: 10.1109/ACCESS.2024.3491995.

[8] S. Aygün, M. Altun, and E. O. Güneş, "Sobel filter operation in image processing via stochastic arithmetic-logic unit design," in *Proc. 25th Signal Processing and*

Communications Applications Conf. (SIU), Antalya, Turkey, 2017, pp. 1–4, doi: 10.1109/SIU.2017.7960479.

[9] S. Garg and S. J. Darak, "FPGA implementation of high speed reconfigurable filter bank for multi-standard wireless communication receivers," in *Proc. 20th Int. Symp. VLSI Design and Test (VDAT)*, Guwahati, India, 2016, pp. 1–5, doi: 10.1109/ISVDAT.2016.8064855.

[10] S. R. Clere, S. Sethumadhavan, and K. Varghese, "FPGA-based reconfigurable coprocessor for deep convolutional neural network training," in *Proc. 21st Euromicro Conf. Digital System Design (DSD)*, Prague, Czech Republic, 2018, pp. 381–388, doi: 10.1109/DSD.2018.00072.

[11] S. Soltani, Y. E. Sagduyu, R. Hasan, K. Davaslioglu, H. Deng, and T. Erpek, "Real-time and embedded deep learning on FPGA for RF signal classification," in *Proc. MILCOM 2019 - IEEE Military Communications Conf. (MILCOM)*, Norfolk, VA, USA, 2019, pp. 1–6, doi: 10.1109/MILCOM47813.2019.902109.

[12] T. Tantipiriyakul and K. Kanjanasit, "Chessboard-coding metasurface antennas with binary defects for anomalous radiation: novel and continuous development," *Journal of Current Science and Technology (JCST)*, vol. 15, no. 4, 2025. doi: 10.59796/jcst.V15N4.2025.135.

[13] T. Tantipiriyakul and K. Kanjanasit, "Design and simulation of chessboard coding wave artifacts," *Journal of Information Science and Technology (JIST)*, vol. 13, no. 2, pp. 62–68, Dec. 2023, doi: 10.14456/jist.2023.13.

[14] V.-Q.-B. Ngo, N. K. Anh, and N. K. Quang, "FPGA-based adaptive PID controller using MLP neural network for tracking motion systems," *IEEE Access*, vol. 12, pp. 91568–91574, 2024, doi: 10.1109/ACCESS.2024.3422015.

[15] X. Wan, M. Q. Qi, T. Y. Chen, and T. J. Cui, "Field-programmable beam reconfiguring based on digitally-controlled coding metasurface," *Scientific Reports*, vol. 6, p. 20663, 2016, doi: 10.1038/srep20663.

[16] Z. Liu, Y. Dou, J. Jiang, Q. Wang, and P. Chow, "An FPGA-based processor for training convolutional neural networks," in *Proc. Int. Conf. Field Programmable Technology (ICFPT)*, Melbourne, VIC, Australia, 2017, pp. 207–210, doi: 10.1109/FPT.2017.8280142.

Appendix A

Practical Session

Sesion Objectives

- Develop hands-on skills in Verilog HDL and FPGA-based circuit design.
- Use Xilinx Vivado and Nexys A7 boards for implementing digital logic systems.
- Apply combinational and sequential design principles in practical lab experiments.

To provide a structured alignment between theoretical concepts and practical exercises, Table A.1 presents the mapping of each textbook chapter to its corresponding laboratory sessions. This association ensures that students can reinforce their understanding of core topics—ranging from fundamental logic gate design to complex system-level integration—through hands-on experimentation. By following this linkage, learners can progressively build both conceptual knowledge and practical FPGA implementation skills.

Table A.1: Association of Chapters with Laboratory Exercises

Chapter	**Associated Labs**
Chapter 1: Digital Design and FPGA	Lab 1: Beginning with Xilinx Vivado; Lab 2: Starting with Xilinx Nexys A7
Chapter 2: Fundamentals of Verilog HDL	Lab 3: Basic Logic Gates
Chapter 3: Combinational Logic Design	Lab 4: Adder Design; Lab 5: MUX and DMUX Designs; Lab 6: Encoder and Decoder Design; Lab 7: Rotator and Shifter Design; Lab 8: Simple ALU Design
Chapter 4: Sequential Logic Design	Lab 9: Counters and Clock Divider Design; Lab 10: Display Counter on FPGA; Lab 11: Multiplier Design
Chapter 5: Finite State Machines and Control	Lab 12: FSM Design for Serial Adder; Lab 13: Traffic Light Controller
Chapter 6: Design for Synthesis and Timing	Reinforces Labs 9, 10, and 11
Chapter 7: FPGA Architecture and Resources	Draws examples from Lab 2 (Nexys A7 setup) and Lab 8 (ALU design)
Chapter 8: FPGA Design Flow and Toolchains	Deepens Lab 1 (Vivado workflow)
Chapter 9: System Design and IP Integration	Builds upon Lab 8 (ALU) and Lab 11 (Multiplier)
Chapter 10: Real-World FPGA Projects and Applications	Lab 14: UART Communication; Revisit Lab 13 (Traffic Light Controller)

A.1 Lab 1: Beginning with Xilinx Vivado

Lab Objectives

- Learn how to install and configure the Xilinx Vivado Design Suite.
- Create, simulate, and verify a basic Verilog project.

1. Introduction

Xilinx Vivado Design Suite is a powerful development environment for implementing digital systems on Xilinx FPGAs. It supports RTL design entry, simulation, synthesis, implementation, and bitstream generation for a wide range of Xilinx devices including Artix-7, Zynq-7000, and others. This section provides a step-by-step guide to installing Vivado, creating a new project, and verifying a basic Verilog design.

2. Installation

System Requirements:

- Operating System: Windows 10/11 (64-bit) or Ubuntu 20.04+
- RAM: Minimum 8 GB (16 GB recommended)
- Disk Space: At least 50 GB

Installation Steps:

1. Go to https://www.xilinx.com/support/download.html.
2. Navigate to `Products > Design Tools > Vivado Design Suite`
3. Register an account and download the Vivado WebPACK edition
4. Run the installer and select:
    - Vivado HL WebPACK
    - Device family support (e.g., Artix-7 for Nexys A7 board)
5. Complete the installation and license activation

3. Creating a New Vivado Project

1. Launch Vivado and select **Create Project**, as shown in Figure A.1.

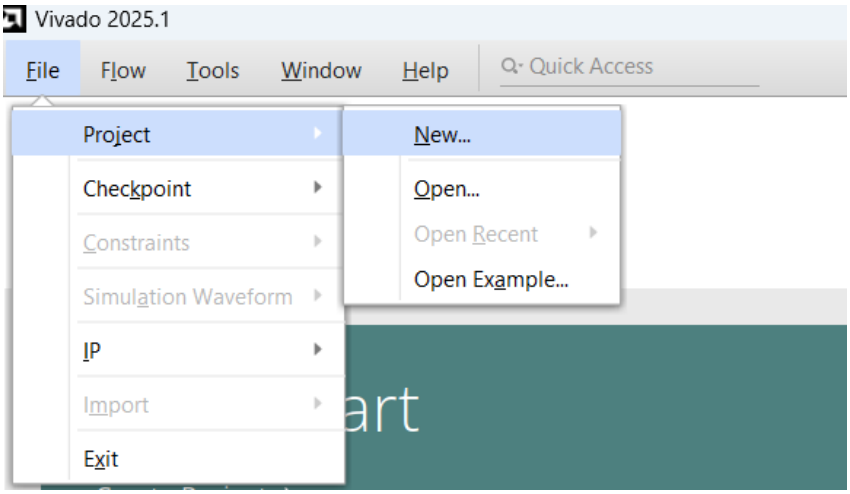


Figure A.1: Creating a new project

2. Enter project name and location, as demonstrated in Figure A.2.

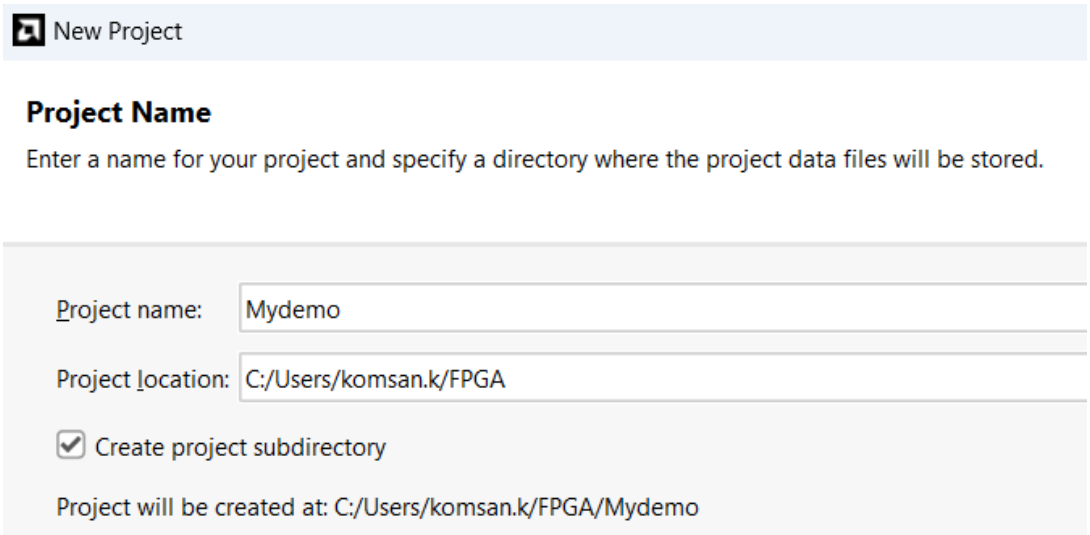


Figure A.2: Completing file name and directory location

3. Select **RTL Project** and enable `Do not specify sources at this time`

4. Select the target FPGA device `xc7a100tcsg324-1`, as shown in Figure A.3.

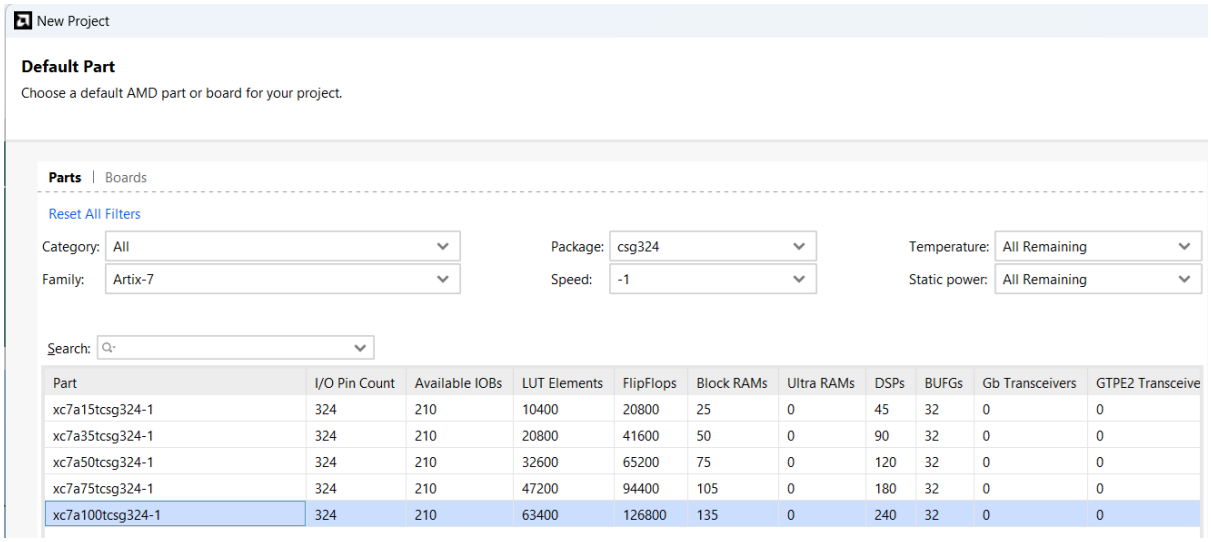


Figure A.3: Specifying the target FPGA chip

5. Finish project creation

4. Adding Design Sources

- Go to `Project Manager > Add Sources`
- Select `Add or Create Design Sources`, as shown in Figure A.4.
- Select `Add or Create Simulation Sources`, as shown in Figure A.5.

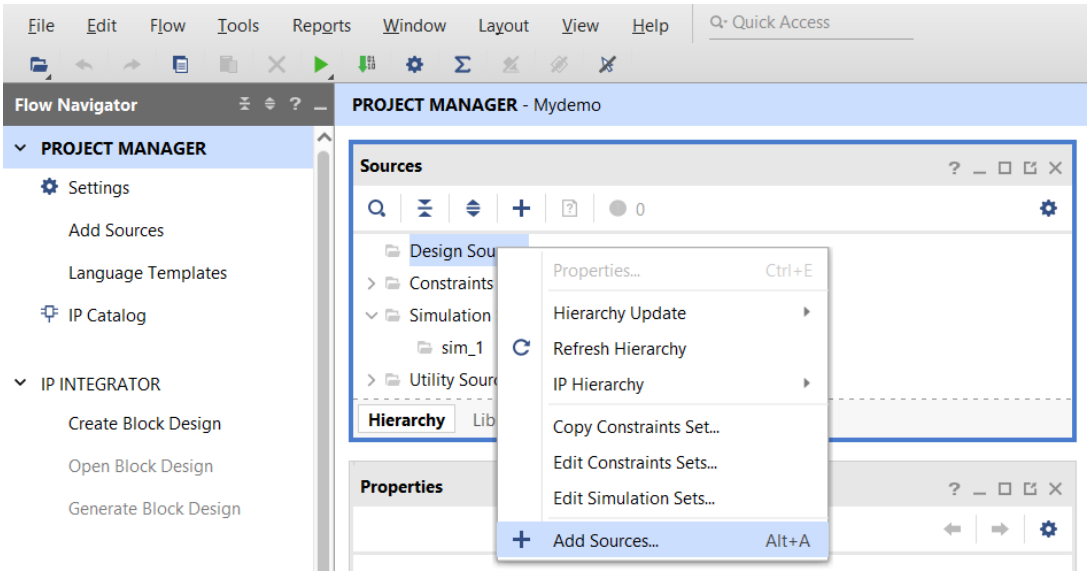


Figure A.4: Adding a new design source

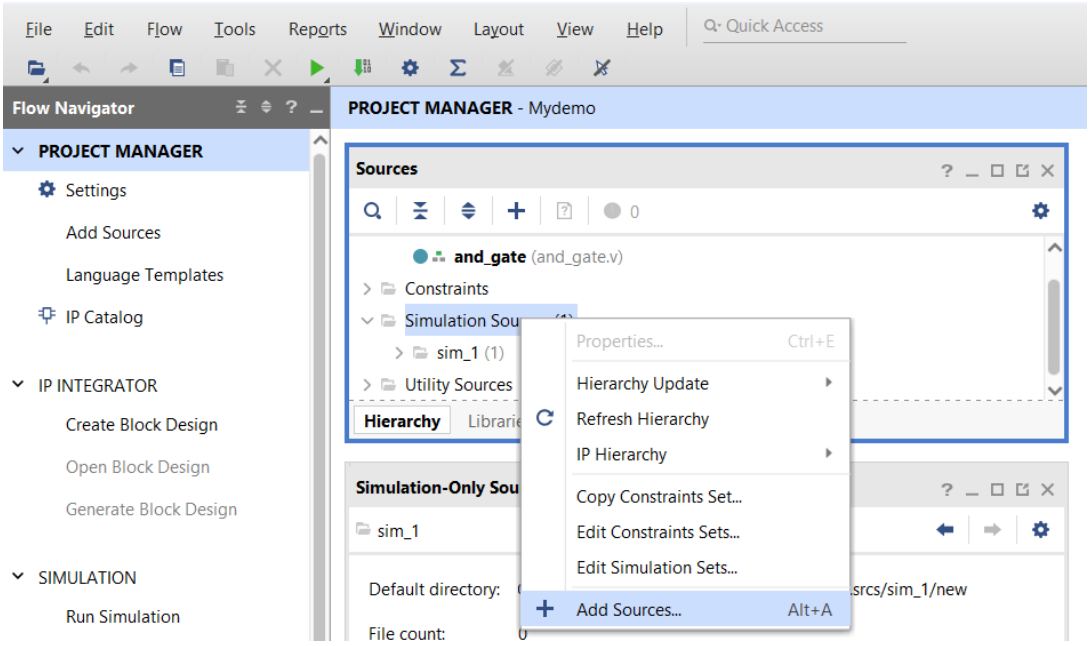


Figure A.5: Adding a new testbench source

- Add files such as `and_gate.v`, `and_gate_tb.v`

Example Verilog Module:

```verilog
// Simple 2-input AND gate module
module and_gate(input A, input B, output Y);
    assign Y = A & B;
endmodule
```

5. Running Simulation

Testbench:

```verilog
module and_gate_tb;
    reg A, B;
    wire Y;
   // Instantiate the AND gate as Unit Under Test (UUT)
    and_gate uut (.A(A), .B(B), .Y(Y));
    initial begin
  // Apply all possible input combinations to test the AND gate
        A = 0; B = 0; #10;
        A = 0; B = 1; #10;
        A = 1; B = 0; #10;
        A = 1; B = 1; #10;
    end
endmodule
```

Steps:

- Select `Run Simulation > Run Behavioral Simulation`, as demonstrated in Figure A.6.
- Use the waveform viewer to inspect outputs, as indicated in Figure A.7.

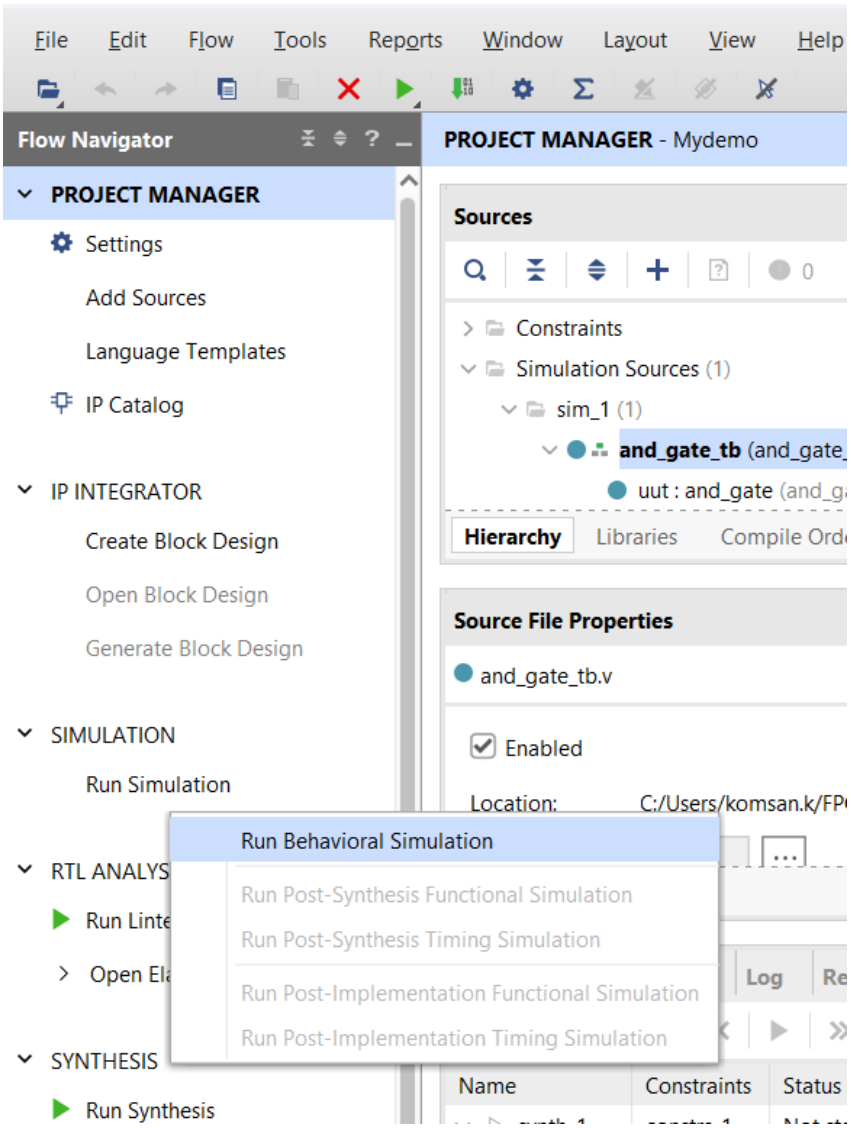


Figure A.6: Start a simulation

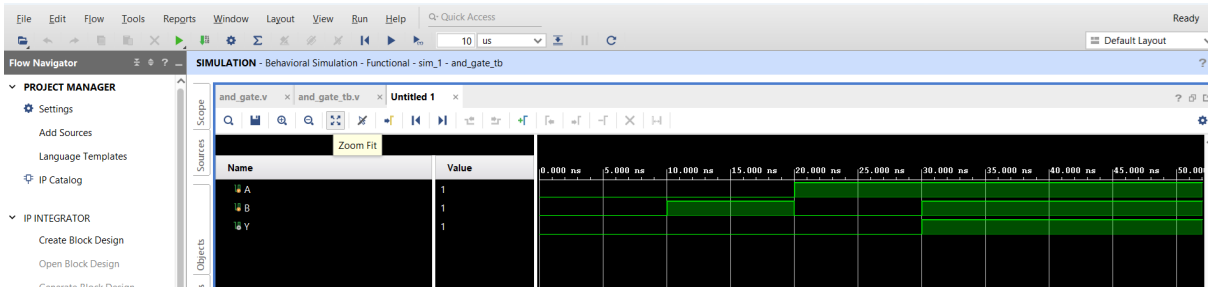


Figure A.7: Waveform viewer

6. Tips and Troubleshooting

- Use $display and $monitor for testbench output
- Map I/O pins with an appropriate XDC constraint file
- Always check the `Messages` panel for warnings and errors

7. Conclusion

This section introduced the Vivado design environment and demonstrated how to create a new project, add Verilog files, run simulation, synthesize a design, and program an FPGA device. This foundation is essential for building more complex digital systems in subsequent labs.

A.2 Lab 2: Starting with Xilinx Nexys A7

The Xilinx Nexys A7 is an FPGA development board based on the Xilinx Artix-7 family. It is ideal for learning digital design using Verilog HDL and for prototyping embedded systems. This section guides you through setting up your development environment and preparing your first project.

Lab Objectives

- Set up the Nexys A7 FPGA board and configure Vivado for Artix-7 devices.
- Create a simple Verilog project to control onboard components.
- Use XDC constraints and program the FPGA with the generated bitstream.

1. Board Overview

Figure A.8 illustrates the Nexys A7 FPGA development board, highlighting essential lab components such as 16 slide switches, 16 individual LEDs, and an 8-digit seven-segment display. Comprehensive documentation is available in the official Digilent reference manual, accessible at the Digilent website.

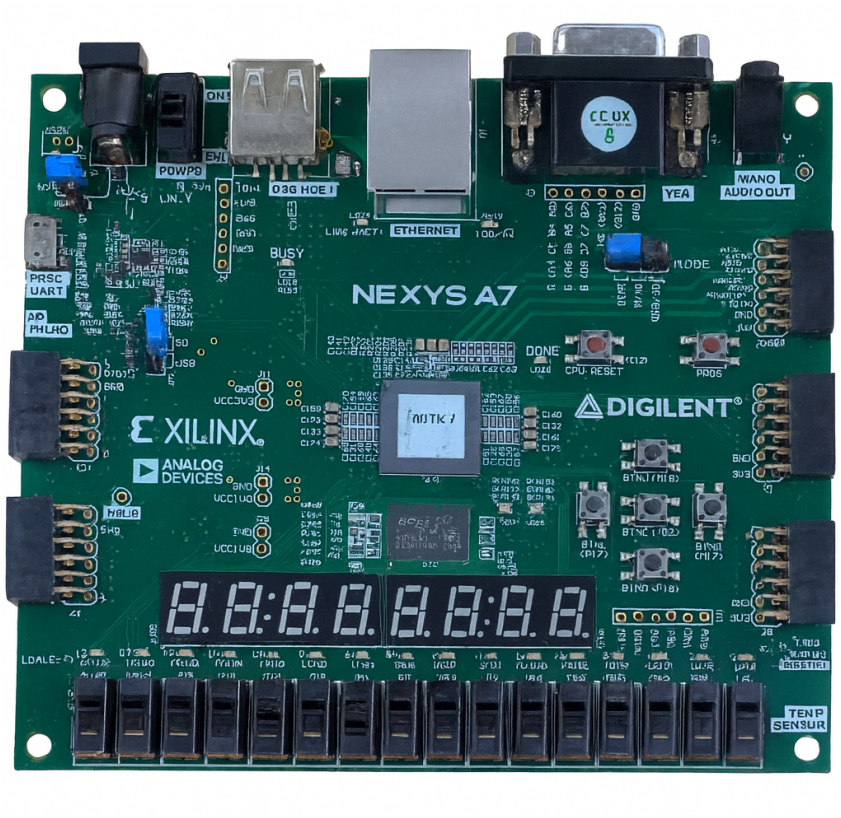


Figure A.8: Nexys A7 Features

The Nexys A7 provides:

- Xilinx Artix-7 XC7A100T-1CSG324C FPGA
- 100 MHz system clock
- 16 user switches, 16 LEDs
- 7-segment display (Eight digits)
- VGA, USB-UART, and Pmod connectors

2. Required Software

Before you begin, install the following tools:

- **Xilinx Vivado Design Suite** (WebPACK edition is free)
- **Digilent Board Files** for Nexys A7 (optional for simplified setup)
- **USB drivers** (FTDI drivers may be required for UART interface)

3. Initial Setup Steps (similar to Lab 1)

1. Connect the Nexys A7 to your computer using the micro-USB cable.
2. Open Vivado and create a new RTL project with the following options:
    - Enable `Verilog` as the target language.
    - Set the part number to `xc7a100tcsg324-1`.
3. Create source files and constraint files (.xdc) for your design.

4. Pin Mapping (Nexys A7)

The detial below lists common I/O components and their corresponding FPGA pin locations on the Nexys A7 board. These mappings should be defined in your constraints file ('.xdc') to connect Verilog ports to physical pins.

- **Clock and Reset:**
    - `clk`: `E3` (100 MHz onboard clock)
    - `rst`: `C12` (push button)
- **Slide Switches (`sw[15:0]`):**

  - V10, U11, U12, H6, T13, R16, U8, T8, R13, U18, T18, R17, R15, M13, L16, J15

- **LEDs (led[15:0]):**
  - V11, V12, V14, V15, T16, U14, T15, V16, U16, U17, V17, R18, N14, J13, K15, H17

- **Push Buttons (btn[4:0]):**
  - N17, P18, M18, M17, P17

- **7-Segment Display:**
  - an[7:0]: U13, K2, T14, P14, J14, T9, J18, J17
  - seg[6:0]: L18, T11,P15, K13, K16, R10, T10
  - dp (decimal point): H15

- **UART (USB-to-Serial):**
  - tx (FPGA to PC): D4
  - rx (PC to FPGA): C4

- **Pmod Header JA (8 signals):**
  - JA[0]: C17
  - JA[1]: D18
  - JA[2]: E18
  - JA[3]: G17
  - JA[4]: D17
  - JA[5]: E17
  - JA[6]: F18
  - JA[7]: G18

- **Pmod Header JB (4 signals):**
  - JB[0]: D14
  - JB[1]: F16
  - JB[2]: G16
  - JB[3]: H14
  - JB[4]: E16

  - JB[5]: F13
  - JB[6]: G13
  - JB[7]: H16

- **VGA Output:**
  - vga_red[3:0]: A4, C5, B4, A3
  - vga_green[3:0]: A6, B6, A5, C6
  - vga_blue[3:0]: D8, D7, C7, B7
  - vga_hsync: B11, vga_vsync: B12

5. First Project: LED Blinker

As a starting project, create a simple Verilog module to blink an LED based on a divided clock. This verifies your toolchain, constraints, and programming workflow.

```
module led_blinker ( // Toggles LED using the MSB of a counter
    input wire clk,
    output reg led
);
    reg [25:0] counter = 0;

    always @(posedge clk) begin
        counter <= counter + 1;    // Increment counter
        led <= counter[25];        // LED toggles
    end
endmodule
```

Connect ‘led’ to one of the onboard LEDs and ‘clk’ to pin ‘E3’.

6. Creating the XDC Constraints File for Nexys A7

To map Verilog HDL ports to physical FPGA pins on the Nexys A7 board, a constraints file (‘.xdc’) is required. This file specifies the pin locations and voltage standards for each I/O signal used in the design.

Below is a minimal example of a constraint file for a basic LED blinker project using the onboard 100 MHz clock and a single LED.

```
## Clock signal
set_property PACKAGE_PIN E3 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

```

```
## LED output (e.g., led[0])
set_property PACKAGE_PIN H17 [get_ports led]
set_property IOSTANDARD LVCMOS33 [get_ports led]

## Optional: Reset button
# set_property PACKAGE_PIN C12 [get_ports rst]
# set_property IOSTANDARD LVCMOS33 [get_ports rst]
```

Save the file as `led_blinker.xdc` and add it to your Vivado project using:

- **Flow Navigator** → `Add Sources` → `Add or Create Constraints`, as shown in Figure A.9.

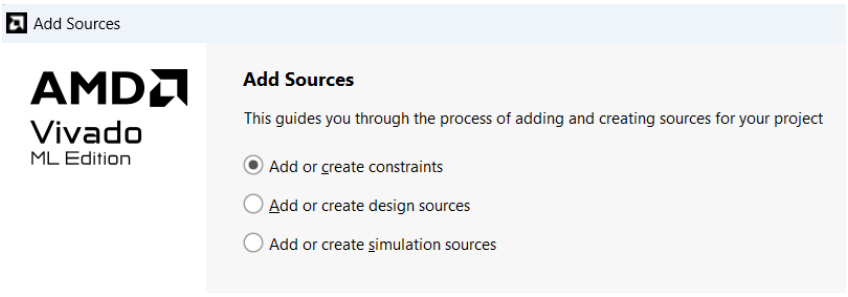


Figure A.9: Adding a new Constraints file

If you plan to use more LEDs or additional inputs, extend the constraint file accordingly by adding the appropriate pins listed in Table A.2.

Table A.2: Nexys A7 I/O pin mapping summary

Component	Signal Name	FPGA Pin(s)
Clock	clk	E3
Reset	rst	C12
LEDs	led[7:0]	U16, U17, V17, R18, N14, J13, K15, H17

7. Programming the FPGA

1. Synthesize the design in Vivado, as shown in Figure A.10.
2. Generate the bitstream, as shown in Figure A.11.
3. Open the Hardware Manager, as shown in Figure A.12.
4. Open Target to connect to the FPGA chip, as shown in Figure A.13.

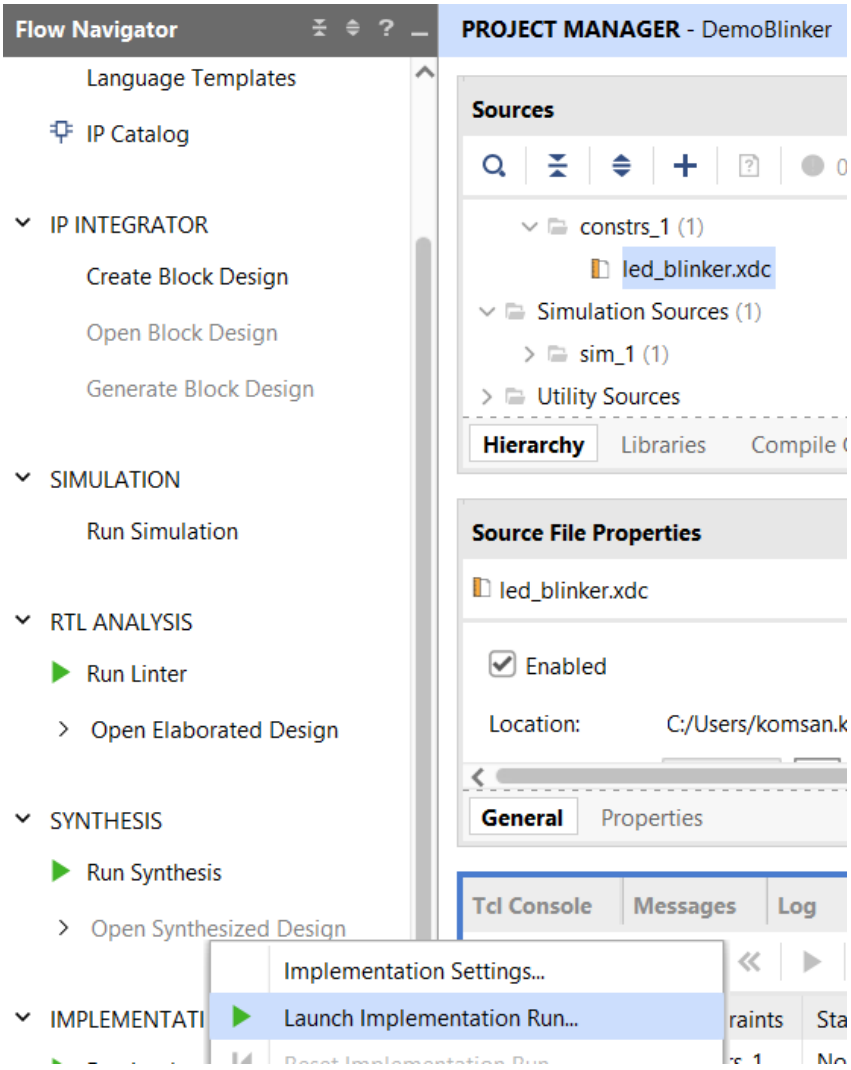


Figure A.10: Performing design synthesis

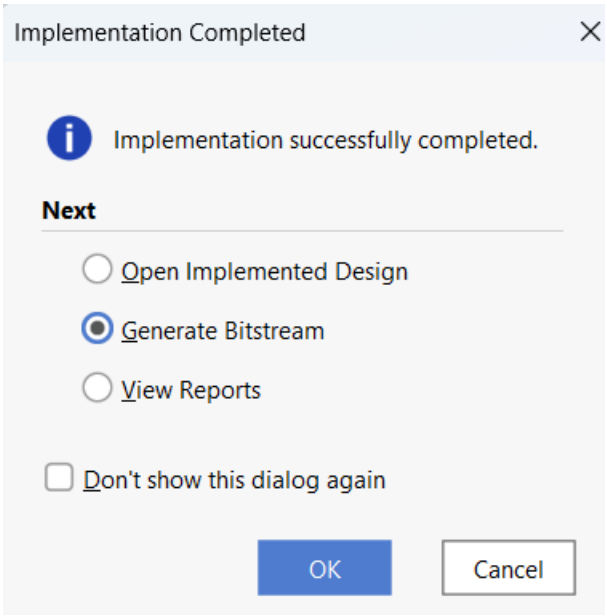


Figure A.11: Completing the Constraints file

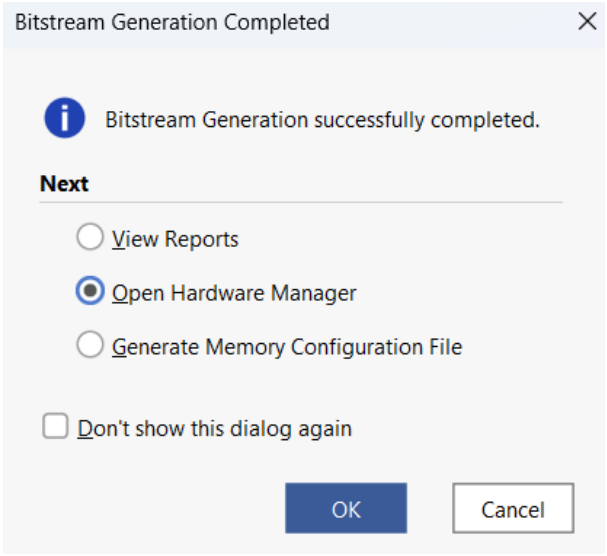


Figure A.12: Opening the hardware manager

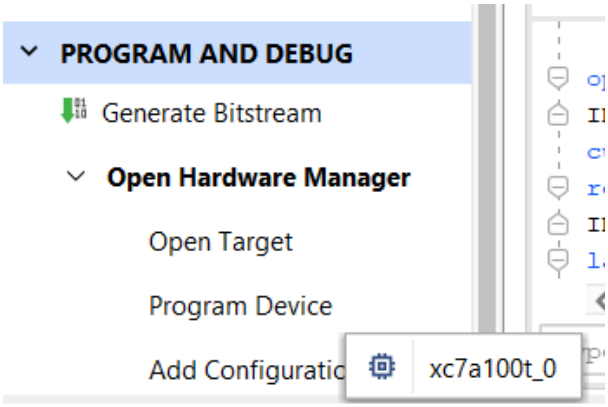


Figure A.13: Selecting the target device

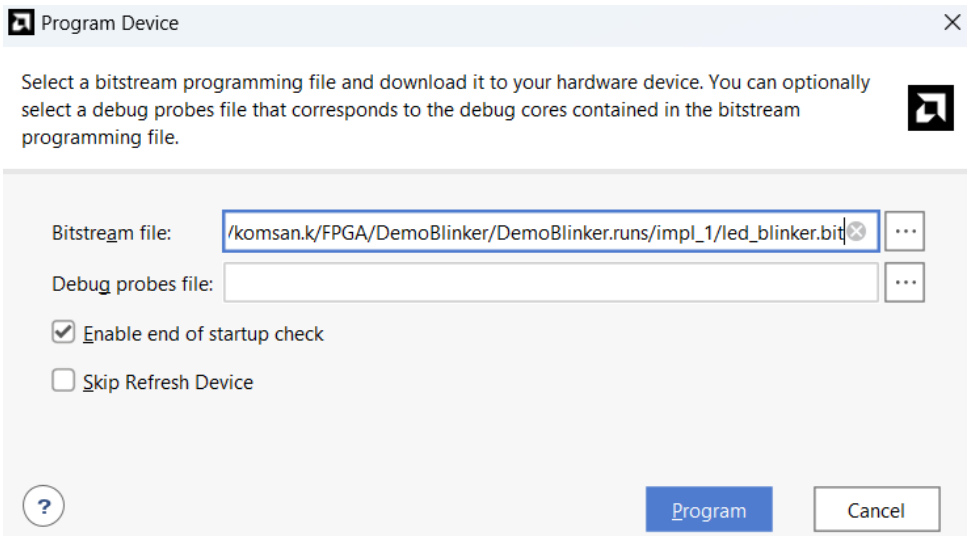


Figure A.14: Programming the device with the Bitstream file

5. Program the device with the generated '.bit' file, as shown in Figure A.14.

Once programmed, the LED should blink at approximately 0.75 Hz (assuming a 100 MHz input clock).

8. Summary

You have now completed the initial steps to configure your environment and run a basic design on the Nexys A7 FPGA. From here, you can proceed to explore modules such as counters, multiplexers, FSMs, and UART transmitters.

A.3 Lab 3: Basic Logic Gates

1. Lab Objective

- To understand and implement basic logic gates (AND, OR, NOT, NAND, NOR, XOR, XNOR) using Verilog HDL.
- To write simple Verilog modules for individual logic gates.
- To verify the functionality through testbenches and waveform simulations.

2. Background Theory

2.1 Logic Gates Overview

Digital circuits operate on binary signals (0 and 1). Basic logic gates form the foundation of these circuits. Table A.3 summarizes common logic gates along with their operations and corresponding Verilog operators.

Table A.3: Basic logic gates and their Verilog representations

Gate	Symbol	Operation	Verilog Operator
AND	$\wedge$	A * B	`&`
OR	$\vee$	A + B	`|`
NOT	$\neg$	A	`~`
NAND	$\neg(A \wedge B)$	$\widetilde{(A \,\&\, B)}$	`~(A & B)`
NOR	$\neg(A \vee B)$	$\widetilde{(A \mid B)}$	`~(A | B)`
XOR	$\oplus$	$A \oplus B$	`^`
XNOR	$\equiv$	$(A \oplus B)$	`~(A ^ B)`

2.2 Verilog Module Structure

```
module <module_name> (input ..., output ...);
    // Declarations
    // Logic
endmodule
```

3. Lab Tasks

3.1 Task 1: Implement Individual Logic Gates

(a) AND Gate

```
module and_gate(input A, input B, output Y);
    assign Y = A & B;   // Logic AND operation
endmodule
```

(b) OR Gate

```
module or_gate(input A, input B, output Y);
    assign Y = A | B; // Logic OR operation
endmodule
```

(c) NOT Gate

```
module not_gate(input A, output Y);
    assign Y = ~A; // Logic NOT operation
endmodule
```

3.2 Task 2: Testbench for AND Gate

```
// Applies all input cases and prints results
module and_gate_tb;
    reg A, B;
    wire Y;

    // Instantiate the Unit Under Test (UUT)
    and_gate uut (.A(A), .B(B), .Y(Y));

    initial begin
        // Display header
        $display("A B | Y");

        // Monitor signals and print whenever they change
        $monitor("%b %b | %b", A, B, Y);

        // Apply all input combinations with delay
        A = 0; B = 0; #10;
        A = 0; B = 1; #10;
        A = 1; B = 0; #10;
        A = 1; B = 1; #10;

        $finish; // End simulation
    end
endmodule
```

3.3 Task 3: Combined Module (Optional)

```
// Module implementing all basic 2-input logic gates
module all_gates (
    input A, input B,
    output AND_out, OR_out, NAND_out,
    output NOR_out, XOR_out, XNOR_out
);
    assign AND_out  = A & B;     // AND gate
    assign OR_out   = A | B;     // OR gate
    assign NAND_out = ~(A & B); // NAND gate
    assign NOR_out  = ~(A | B); // NOR gate
    assign XOR_out  = A ^ B;     // XOR gate
    assign XNOR_out = ~(A ^ B); // XNOR gate
endmodule
```

4. Simulation and Result Analysis

4.1 Expected Truth Table

The expected truth table for basic logic gates with two binary inputs (A and B) is shown in Table A.4. This table demonstrates the outputs of common logic operations.

Table A.4: Truth table for basic logic gates with two inputs

A	B	AND	OR	NAND	NOR	XOR	XNOR
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1

5. Questions and Exercises

1. Modify the testbench to include more delays and corner cases.
2. What is the functional difference between XOR and XNOR?
3. Describe a real-world application where NAND gates are preferred.
4. Write a 2-input multiplexer using only logic gates in Verilog.
5. Explain why the NOT gate is considered a unary operator.

6. Conclusion

This lab provided practical experience in writing Verilog HDL code for basic logic gates and using simulation tools to verify their behavior.

7. Instructor Notes

- Emphasize the difference between behavioral and structural coding styles.
- Ensure that students simulate and verify outputs with proper waveform timing.
- Encourage debugging by visual inspection and use of `$monitor`.

A.4 Lab 4: Adder Design

1. Lab Objective

- To understand and implement the design of Half Adder and Full Adder circuits using Verilog HDL.
- To design a 4-bit binary adder using Full Adders.
- To verify all designs through simulation and waveform analysis.

2. Background Theory

2.1 Half Adder

A Half Adder adds two single-bit binary numbers (A and B). It produces:

- **Sum** = A $\oplus$ B
- **Carry** = A & B

2.2 Full Adder

A Full Adder adds three 1-bit binary numbers: A, B, and Carry-in (C_{in}). It outputs:

- **Sum** = A $\oplus$ B $\oplus$ C_{in}
- **Carry-out** = (A & B) | (B & C_{in}) | (A & C_{in})

2.3 4-bit Ripple Carry Adder

A 4-bit adder is formed by connecting four Full Adders in series. Each carry-out from the previous stage becomes the carry-in of the next.

3. Lab Tasks

3.1 Task 1: Verilog Code for Half Adder

```
// Half Adder: computes Sum and Carry of two 1-bit inputs
module half_adder(input A, input B, output Sum, output Carry);
    assign Sum   = A ^ B;  // Sum is XOR of A and B
    assign Carry = A & B;  // Carry is AND of A and B
endmodule
```

3.2 Task 2: Verilog Code for Full Adder

```
// Full Adder: computes Sum and Carry-Out of three 1-bit inputs
module full_adder(input A, input B, input Cin, output Sum, output
    Cout);
   // Sum is XOR of A, B, and Carry-In
assign Sum  = A ^ B ^ Cin;
  // Carry-Out when any two or more inputs are 1
    assign Cout = (A & B) | (B & Cin) | (A & Cin);
endmodule
```

3.3 Task 3: 4-bit Ripple Carry Adder

```
// 4-bit Ripple Carry Adder built from four 1-bit Full Adders
module four_bit_adder(
    input [3:0] A,     // 4-bit input A
    input [3:0] B,     // 4-bit input B
    input Cin,         // Initial Carry-In
    output [3:0] Sum, // 4-bit Sum output
    output Cout        // Final Carry-Out
);
    wire c1, c2, c3;  // Internal carry wires

    // Connect four Full Adders in series (ripple structure)
    full_adder FA0(A[0], B[0], Cin,  Sum[0], c1);   // LSB
    full_adder FA1(A[1], B[1], c1,   Sum[1], c2);
    full_adder FA2(A[2], B[2], c2,   Sum[2], c3);
    full_adder FA3(A[3], B[3], c3,   Sum[3], Cout); // MSB
endmodule
```

4. Testbenches

4.1 Half Adder Testbench

```
// Testbench for Half Adder:
//Applies all input cases and displays truth table
module half_adder_tb;
    reg A, B;               // Inputs
    wire Sum, Carry;        // Outputs

    // Instantiate Unit Under Test (UUT)
```

```
    half_adder uut(.A(A), .B(B), .Sum(Sum), .Carry(Carry));

    initial begin
        // Display header
        $display("A B | Sum Carry");

        // Monitor signals and print whenever they change
        $monitor("%b %b |  %b     %b", A, B, Sum, Carry);

        // Apply all inputs (truth table of half adder)
        A = 0; B = 0; #10;  // Expect Sum=0, Carry=0
        A = 0; B = 1; #10;  // Expect Sum=1, Carry=0
        A = 1; B = 0; #10;  // Expect Sum=1, Carry=0
        A = 1; B = 1; #10;  // Expect Sum=0, Carry=1

        $finish; // End simulation
    end
endmodule
```

4.2 Full Adder Testbench

```
// Testbench for Full Adder:
// Applies input cases and prints truth table
module full_adder_tb;
    reg A, B, Cin;          // Inputs
    wire Sum, Cout;         // Outputs

    // Instantiate Unit Under Test (UUT)
    full_adder uut(.A(A), .B(B), .Cin(Cin), .Sum(Sum), .Cout(Cout
       ));

    initial begin
        // Display header
        $display("A B Cin | Sum Cout");

        // Monitor signals and print whenever they change
        $monitor("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

        // Apply inputs (selected cases of truth table)
        A = 0; B = 0; Cin = 0; #10;  // Expect Sum=0, Cout=0
        A = 0; B = 1; Cin = 0; #10;  // Expect Sum=1, Cout=0
```

```
        A = 1; B = 0; Cin = 0; #10;  // Expect Sum=1, Cout=0
        A = 1; B = 1; Cin = 0; #10;  // Expect Sum=0, Cout=1
        A = 0; B = 0; Cin = 1; #10;  // Expect Sum=1, Cout=0
        A = 1; B = 1; Cin = 1; #10;  // Expect Sum=1, Cout=1

        $finish; // End simulation
    end
endmodule
```

4.3 4-bit Adder Testbench

```
// Testbench for 4-bit Ripple Carry Adder
module four_bit_adder_tb;
    reg [3:0] A, B;        // 4-bit inputs
    reg Cin;               // Carry-in input
    wire [3:0] Sum;        // 4-bit sum output
    wire Cout;             // Carry-out output

    // Instantiate Unit Under Test (UUT)
    four_bit_adder uut(.A(A), .B(B), .Cin(Cin), .Sum(Sum), .Cout(
       Cout));

    initial begin
        // Display header
        $display("   A    B  Cin |  Sum Cout");

        // Monitor values: print whenever signals change
        $monitor("%b %b   %b   |  %b   %b", A, B, Cin, Sum, Cout)
           ;

        // Apply test cases
       // 1 + 2 = 3
        A = 4'b0001; B = 4'b0010; Cin = 0; #10;
       // 15 + 1 = 16 (Cout=1, Sum=0000)
        A = 4'b1111; B = 4'b0001; Cin = 0; #10;
       // 10 + 5 + 1 = 16 (Cout=1, Sum=0000)
        A = 4'b1010; B = 4'b0101; Cin = 1; #10;

        $finish; // End simulation
    end
endmodule
```

5. Simulation Results

5.1 Expected Half Adder Truth Table

Table A.5 shows the expected output of a half adder circuit. The half adder produces a Sum and a Carry output for each combination of two input bits.

Table A.5: Truth table for half adder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

5.2 Full Adder Truth Table (Partial)

Table A.6 lists a partial truth table for a full adder circuit. A full adder has three inputs (A, B, and Carry-in) and produces a Sum and Carry-out result.

Table A.6: Complete truth table for full adder

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

6. Exercises

1. Modify the full adder to display overflow condition in 4-bit addition.

2. Implement a 4-bit subtractor using full adders and 2's complement logic.

3. Extend the design to an 8-bit adder.

4. Write behavioral Verilog code for the full adder and compare with structural style.

5. Analyze propagation delay in a ripple carry adder.

7. Conclusion

This lab provided practical experience in designing and simulating Half Adders, Full Adders, and multi-bit binary adders. These are essential components for building Arithmetic Logic Units (ALUs) and understanding fundamental arithmetic operations in digital systems.

8. Instructor Notes

- Emphasize the design hierarchy (e.g., 4-bit adder using full adders).
- Discuss testbench writing and timing simulation.
- Encourage students to debug using waveform outputs.

A.5 Lab 5: MUX and DMUX Design

1. Lab Objective

- To understand and implement multiplexers and demultiplexers using the 'case' statement in Verilog HDL.
- To verify the logical correctness through simulation and waveform analysis.

2. Background Theory

2.1 Multiplexer (MUX)

A multiplexer selects one of several inputs and forwards it to a single output, controlled by select lines. For a 4-to-1 MUX, the output is defined as:

$$Y = I[S]$$

Where `I` is a 4-bit input and `S` is a 2-bit selector.

2.2 Demultiplexer (DEMUX)

A demultiplexer routes a single input to one of several outputs depending on select lines. For a 1-to-4 DEMUX:

$$Y[S] = D$$

3. Verilog Implementation using `case`

3.1 4-to-1 Multiplexer

```
// 4-to-1 Multiplexer: selects one of 4 inputs based on select
module mux_4to1 (
    input [3:0] I,    // 4 input lines
    input [1:0] S,    // 2-bit select signal
    output reg Y      // Output
);
    always @(*) begin
        case (S)
            2'b00: Y = I[0]; // Select input 0
            2'b01: Y = I[1]; // Select input 1
            2'b10: Y = I[2]; // Select input 2
            2'b11: Y = I[3]; // Select input 3
            default: Y = 1'b0; // Default output if undefined
```

```
        endcase
    end
endmodule
```

3.2 1-to-4 Demultiplexer

```
// 1-to-4 Demultiplexer: routes input D to one of 4 outputs
module demux_1to4 (
    input D,             // Single data input
    input [1:0] S,       // 2-bit select signal
    output reg [3:0] Y   // 4 output lines
);
    always @(*) begin
        Y = 4'b0000; // Default: all outputs inactive
        case (S)
            2'b00: Y = 4'b0001 & {4{D}}; // Send D to Y[0]
            2'b01: Y = 4'b0010 & {4{D}}; // Send D to Y[1]
            2'b10: Y = 4'b0100 & {4{D}}; // Send D to Y[2]
            2'b11: Y = 4'b1000 & {4{D}}; // Send D to Y[3]
        endcase
    end
endmodule
```

4. Testbenches

4.1 Multiplexer Testbench

```
// Testbench for 4-to-1 Multiplexer
module mux_tb;
    reg [3:0] I;     // 4 input lines
    reg [1:0] S;     // 2-bit select signal
    wire Y;          // Output

    // Instantiate Unit Under Test (UUT)
    mux_4to1 uut (.I(I), .S(S), .Y(Y));

    initial begin
        // Display header
        $display(" I     S | Y");

        // Apply test vector: fixed input pattern
```

```
        I = 4'b1010;  // Inputs = {I3=1, I2=0, I1=1, I0=0}

        // Change select signals to test all cases
        S = 2'b00; #10; // Expect Y = I[0] = 0
        S = 2'b01; #10; // Expect Y = I[1] = 1
        S = 2'b10; #10; // Expect Y = I[2] = 0
        S = 2'b11; #10; // Expect Y = I[3] = 1

        $finish; // End simulation
    end
endmodule
```

4.2 Demultiplexer Testbench

```
// Testbench for 1-to-4 Demultiplexer
module demux_tb;
    reg D;            // Data input
    reg [1:0] S;      // 2-bit select signal
    wire [3:0] Y;     // 4 output lines

    // Instantiate Unit Under Test (UUT)
    demux_1to4 uut (.D(D), .S(S), .Y(Y));

    initial begin
        // Display header
        $display("D  S | Y");

        D = 1; // Data input is always 1 for testing

        // Set select and check active output
        S = 2'b00; #10; // Expect Y = 0001
        S = 2'b01; #10; // Expect Y = 0010
        S = 2'b10; #10; // Expect Y = 0100
        S = 2'b11; #10; // Expect Y = 1000

        $finish; // End simulation
    end
endmodule
```

5. Simulation and Result Analysis

5.1 Truth Table: MUX

Table A.7 illustrates the behavior of a 4-to-1 multiplexer, showing how the selected input bit ('I[3:0]') is routed to the output 'Y' based on the select lines 'S'.

Table A.7: 4-to-1 multiplexer truth table

I[3:0]	**S**	**Y**
1010	00	0
1010	01	1
1010	10	0
1010	11	1

5.2 Truth Table: DEMUX

Table A.8 demonstrates the output behavior of a 1-to-4 demultiplexer. Based on the select input 'S', the input data 'D' is directed to one of the four outputs in 'Y[3:0]', with only one output line set to high at a time.

Table A.8: 1-to-4 demultiplexer truth table

D	**S**	**Y[3:0]**
1	00	0001
1	01	0010
1	10	0100
1	11	1000

6. Exercises

1. Convert the MUX to an 8-to-1 design using nested case statements.
2. Extend the DEMUX to 1-to-8 with an enable input.
3. Modify the MUX to handle a default input when select is out of range.
4. Discuss how"casez" could simplify DEMUX code.
5. Simulate with different I values and validate all paths.

7. Conclusion

This lab demonstrated the use of 'case' statements in Verilog HDL to implement multiplexers and demultiplexers. The structured approach improves readability and debugging, especially for larger systems.

8. Instructor Notes

- Emphasize correct use of blocking vs non-blocking assignments in always blocks.
- Review synthesis implications of combinational ‘case’ blocks.
- Discuss practical examples of MUX/DEMUX in bus architectures and control units.

A.6 Lab 6: Encoder and Decoder Design

1. Lab Objective

- To design and simulate binary encoders and decoders using behavioral and hierarchical modeling styles in Verilog HDL.
- To compare modular design techniques and understand abstraction in combinational logic design.

2. Background Theory

2.1 Binary Encoder

A binary encoder converts one active input line into a binary code on the output. A 4-to-2 encoder maps four input lines to two output bits.

Priority Behavior:

- If multiple inputs are active, the encoder outputs the code of the highest-priority input (e.g., I[3] has highest priority).

2.2 Binary Decoder

A decoder performs the reverse operation: it takes a binary input code and activates one output line.

2-to-4 Decoder:

- For every binary input combination, only one output line is activated.
- Useful in memory address decoding, instruction decoding, etc.

3. Verilog Implementation

3.1 Behavioral 4-to-2 Encoder

```
1 // 4-to-2 Encoder (behavioral):
2 // converts one active input into 2-bit output code
3 module encoder_4to2_behavioral (
4     input [3:0] I,  // 4 input lines (only one should be active)
5     output reg [1:0] Y // 2-bit encoded output
6 );
7     always @(*) begin
8         Y = 2'b00;            // Default output
9         if (I[3]) Y = 2'b11; // If I3 active -> Y = 11
```

```
        else if (I[2]) Y = 2'b10; // If I2 active -> Y = 10
        else if (I[1]) Y = 2'b01; // If I1 active -> Y = 01
        else if (I[0]) Y = 2'b00; // If I0 active -> Y = 00
    end
endmodule
```

3.2 Behavioral 2-to-4 Decoder

```
// 2-to-4 Decoder (behavioral):
// decodes 2-bit input into one active output
module decoder_2to4_behavioral (
    input [1:0] A,      // 2-bit input
    output reg [3:0] Y // 4 output lines
);
    always @(*) begin
        Y = 4'b0000;    // Default: all outputs inactive
      // Activate the output line selected by input A
        Y[A] = 1'b1;
    end
endmodule
```

3.3 Hierarchical 2-to-4 Decoder

Bit Cell Module:

```
// 1-bit Decoder: passes enable signal directly to output
module decoder_bit (
    input enable,  // Enable input
    output Y       // Output (active when enable=1)
);
    assign Y = enable; // Output follows enable
endmodule
```

2-to-4 Decoder Using Bit Cells:

```
// 2-to-4 Decoder (hierarchical):
// built from 1-bit decoder modules
module decoder_2to4_hierarchical (
    input [1:0] A,      // 2-bit input
    output [3:0] Y      // 4 output lines
);
    wire [3:0] enable;    // Enable signals for each decoder_bit
```

```
    assign enable = 4'b0001 << A;  // Shift '1' left by A to
       select one output

    // Instantiate four 1-bit decoders
    decoder_bit d0 (enable[0], Y[0]);  // Activate Y[0] if A=00
    decoder_bit d1 (enable[1], Y[1]);  // Activate Y[1] if A=01
    decoder_bit d2 (enable[2], Y[2]);  // Activate Y[2] if A=10
    decoder_bit d3 (enable[3], Y[3]);  // Activate Y[3] if A=11
endmodule
```

3.4 Hierarchical 4-to-2 Encoder

Priority-Based Hierarchical Encoder:

```
// 4-to-2 Encoder (hierarchical):
//encodes one active input into 2-bit output
module encoder_4to2_hierarchical (
    input [3:0] I, // 4 input lines (priority: I3 highest, I0
       lowest)
    output reg [1:0] Y // 2-bit encoded output
);
    always @(*) begin
        casez (I)
            4'b???1: Y = 2'b00; // If I0 active -> Y=00
            4'b??10: Y = 2'b01; // If I1 active -> Y=01
            4'b?100: Y = 2'b10; // If I2 active -> Y=10
            4'b1000: Y = 2'b11; // If I3 active -> Y=11
            default: Y = 2'b00; // Default when no valid input
        endcase
    end
endmodule
```

4. Testbenches (Shared)

4.1 Encoder Testbench (Generic)

```
// Testbench for 4-to-2 Encoder (behavioral or hierarchical)
module encoder_tb;
    reg [3:0] I;       // 4 input lines
    wire [1:0] Y;      // 2-bit encoded output

    // Instantiate Unit Under Test (UUT)
```

```
    encoder_4to2_behavioral uut (.I(I), .Y(Y));

    initial begin
        // Display header for truth table
        $display("Input  | Output");
        $monitor("%b   |  %b", I, Y); // Monitor input/output

        // Apply test cases (only one input active at a time)
        I = 4'b0001; #10; // Expect Y=00
        I = 4'b0010; #10; // Expect Y=01
        I = 4'b0100; #10; // Expect Y=10
        I = 4'b1000; #10; // Expect Y=11

        $finish; // End simulation
    end
endmodule
```

4.2 Decoder Testbench (Generic)

```
// Testbench for 2-to-4 Decoder (behavioral or hierarchical)
module decoder_tb;
    reg [1:0] A;      // 2-bit input
    wire [3:0] Y;     // 4-bit decoded output

    // Instantiate Unit Under Test (UUT)
    decoder_2to4_behavioral uut (.A(A), .Y(Y));

    initial begin
        // Display header for truth table
        $display("Input | Output");
        $monitor("%b    | %b", A, Y); // Monitor input/output

        // Apply all input cases (truth table of decoder)
        A = 2'b00; #10; // Expect Y = 0001
        A = 2'b01; #10; // Expect Y = 0010
        A = 2'b10; #10; // Expect Y = 0100
        A = 2'b11; #10; // Expect Y = 1000

        $finish; // End simulation
    end
endmodule
```

5. Simulation and Results

5.1 Encoder Truth Table

Table A.9 presents the output mapping of a 4-to-2 encoder. It encodes one active high input line from 'I[3:0]' into a 2-bit binary code on the output 'Y[1:0]'. Only one input should be high at any time to produce a valid result.

Table A.9: 4-to-2 encoder truth table

Input (I[3:0])	**Output (Y[1:0])**
0001	00
0010	01
0100	10
1000	11

5.2 Decoder Truth Table

Table A.10 illustrates the output pattern of a 2-to-4 decoder. Each 2-bit binary input on 'A[1:0]' activates exactly one of the four output lines in 'Y[3:0]', making it suitable for address decoding or control signal generation in digital systems.

Table A.10: 2-to-4 decoder truth table

Input (A[1:0])	**Output (Y[3:0])**
00	0001
01	0010
10	0100
11	1000

6. Exercises

1. Add an enable input to both encoder and decoder modules.

2. Modify the encoder to output a valid flag when at least one input is active.

3. Create a hierarchical 3-to-8 decoder using 2-to-4 decoders.

4. Compare structural, behavioral, and hierarchical models in terms of readability and reusability.

5. Use conditional operators ("?:") to implement a 4-to-2 encoder.

7. Conclusion

This lab explored both behavioral and hierarchical modeling techniques for encoder and decoder circuits using Verilog HDL. Through modular design and abstraction, students gained insight into structured circuit development and simulation practices.

8. Instructor Notes

- Discuss where behavioral models are more suitable than hierarchical (e.g., test environments).
- Emphasize code reuse and top-down design in hierarchical modeling.
- Encourage waveform analysis to confirm signal timing and correctness.

A.7 Lab 7: Rotator and Shifter Design

1. Lab Objective

- To implement logical shifts, arithmetic shifts, and circular rotates in Verilog HDL.
- To understand and implement a barrel shifter capable of multi-bit shifts in a single cycle.
- To simulate and verify the operation of shifters, rotators, and barrel shifters.

2. Background Theory

Shifters and rotators are commonly used in digital systems for data alignment, bit manipulation, and arithmetic operations.

Operation Types:

- **Logical Shift:** Moves bits left or right and fills vacated bits with zeros.
- **Arithmetic Shift:** Preserves the sign bit for signed numbers while shifting right.
- **Rotate:** Wraps bits around the word during shift operations.
- **Barrel Shifter:** Can shift or rotate multiple bits in a single clock cycle using combinational logic.

3. Verilog Implementations

3.1 Shifter Module

```
1  // 4-bit Shifter: supports LSL, LSR, and ASR operations
2  module shifter (
3      input [3:0] data_in,  // 4-bit input data
4      input [1:0] sel,          // 00: LSL, 01: LSR, 10: ASR
5      output reg [3:0] data_out // 4-bit output data
6  );
7      always @(*) begin
8          case (sel)
9              2'b00: data_out = data_in << 1;  // Logical Shift
                   Left
10             2'b01: data_out = data_in >> 1;  // Logical Shift
                   Right
11             // Arithmetic Shift Right (sign-extended)
12             2'b10: data_out = {data_in[3], data_in[3:1]};
```

```
            default: data_out = 4'b0000;  // Default output
        endcase
    end
endmodule
```

3.2 Rotator Module

```
// 4-bit Rotator:
// supports Rotate Left (ROL) and Rotate Right (ROR)
module rotator (
    input [3:0] data_in,       // 4-bit input data
    input [1:0] sel,           // 00: ROL, 01: ROR
    output reg [3:0] data_out // 4-bit output data
);
    always @(*) begin
        case (sel)
            // Rotate Left: MSB -> LSB
            2'b00: data_out = {data_in[2:0], data_in[3]};
           // Rotate Right: LSB -> MSB
            2'b01: data_out = {data_in[0], data_in[3:1]};
            default: data_out = 4'b0000;    // Default output
        endcase
    end
endmodule
```

3.3 Barrel Shifter Module

```
// 8-bit Barrel Shifter:
// shifts data left or right by variable amount
module barrel_shifter (
    input [7:0] data_in,       // 8-bit input data
    input [2:0] shift_amt,     // Shift amount (0-7 positions)
    input dir,                 // Direction: 0 = left, 1 = right
    output reg [7:0] data_out  // 8-bit shifted output
);
    always @(*) begin
        if (dir == 1'b0)
           // Logical left shift
            data_out = data_in << shift_amt;
        else
            // Logical right shift
```

```
            data_out = data_in >> shift_amt;
    end
endmodule
```

4. Testbenches

4.1 Barrel Shifter Testbench

```
// Testbench for 8-bit Barrel Shifter
module barrel_shifter_tb;
    reg [7:0] data_in;      // 8-bit input
    reg [2:0] shift_amt;    // Shift amount (0-7)
    reg dir;                // Direction: 0 = left, 1 = right
    wire [7:0] data_out;    // Shifted output

    // Instantiate Unit Under Test (UUT)
    barrel_shifter uut (.data_in(data_in), .shift_amt(shift_amt),
        .dir(dir), .data_out(data_out));

    initial begin
        // Display header
        $display("Input      | Shift | Dir | Output");

        // Test input pattern
        data_in = 8'b10110011; // Example input value

        // Apply test cases
        dir = 0; shift_amt = 3; #10; // Left shift by 3
           (10011000)
        dir = 1; shift_amt = 2; #10; // Right shift by 2
           (00101100)

        $finish; // End simulation
    end
endmodule
```

5. Simulation and Result Analysis

Simulation was carried out using Xilinx Vivado and ModelSim to validate the correctness of shift, rotate, and barrel shifter operations.

- The testbench applied various shift and rotate commands to 8-bit input data such as `10110011`.
- Logical left shift by 2 resulted in `11001100`.
- Logical right shift by 2 resulted in `00101100`.
- Rotate left by 2 resulted in `11001110`.
- Rotate right by 2 resulted in `11101100`.
- Barrel shifter allowed selection of any shift amount from 0 to 7 bits dynamically.

Verification Method:

- Used waveform viewers (e.g., GTKWave or Vivado Simulator) to inspect output transitions.
- Confirmed that register outputs matched expected results across all operations.
- No timing violations were observed; design met setup and hold constraints for 100 MHz clock.

6. Exercises

1. Modify the barrel shifter to support rotation (ROL, ROR).
2. Extend the barrel shifter to 16 bits with arithmetic shift support.
3. Design a combined shifter/rotator module with a 3-bit control input.
4. Synthesize the design on FPGA and verify functionality with onboard LEDs.

7. Conclusion

This lab demonstrated the implementation of basic shifters, rotators, and a barrel shifter in Verilog HDL. These components are widely used in datapaths for efficient bitwise operations and are foundational for processor-level design.

8. Instructor Notes

- Explain differences in latency and hardware complexity between barrel and serial shifters.
- Discuss application of shifters in ALUs and cryptographic hardware.
- Encourage waveform-based debugging to identify shifting behavior.

A.8 Lab 8: Simple ALU Design

1. Lab Objective

- To design a simple 4-bit Arithmetic Logic Unit (ALU) using Verilog HDL.
- To implement arithmetic (ADD, SUB) and logic operations (AND, OR, XOR, NOT).
- To verify functionality using simulation and testbenches.

2. Background Theory

An ALU (Arithmetic Logic Unit) is a combinational digital circuit that performs a set of arithmetic and logical operations. It is a critical component in CPUs, microcontrollers, and digital signal processors.

Typical ALU Functions:

- Arithmetic: Addition, Subtraction
- Logical: AND, OR, XOR, NOT
- Shift: Left Shift, Right Shift (optional)
- Comparison: Zero flag, Carry flag, Overflow detection

Opcode-based ALU Design: Operations are selected using an `opcode` (Operation Code), which controls a multiplexer to route the correct logic.

3. Verilog Implementation

3.1 ALU Module (4-bit)

```
1  // 4-bit ALU: performs arithmetic and logic operations based on
       opcode
2  module alu_4bit (
3      input [3:0] A,          // 4-bit input A
4      input [3:0] B,          // 4-bit input B
5      input [2:0] opcode,     // Operation selector
6      output reg [3:0] result,// 4-bit result
7      output reg zero         // Zero flag (1 if result == 0)
8  );
9      always @(*) begin
10         case (opcode)
```

```
            3'b000: result = A + B;     // ADD
            3'b001: result = A - B;     // SUB
            3'b010: result = A & B;     // AND
            3'b011: result = A | B;     // OR
            3'b100: result = A ^ B;     // XOR
            3'b101: result = ~A;        // NOT A (ignores B)
            3'b110: result = A << 1;    // Shift Left (logical)
            3'b111: result = A >> 1;    // Shift Right (logical)
            default: result = 4'b0000; // Default case
        endcase
        zero = (result == 4'b0000);     // Set zero flag if result
            is 0
    end
endmodule
```

4. Testbench for ALU

4.1 ALU Testbench

```
// Testbench for 4-bit ALU
module alu_tb;
    reg [3:0] A, B;         // 4-bit inputs
    reg [2:0] opcode;       // Operation selector
    wire [3:0] result;      // ALU result
    wire zero;              // Zero flag

    // Instantiate Unit Under Test (UUT)
    alu_4bit uut (
        .A(A), .B(B),
        .opcode(opcode),
        .result(result),
        .zero(zero)
    );

    initial begin
        // Display header for results
        $display("A    B    OPC | RESULT ZERO");
        $monitor("%b %b  %b   |  %b     %b", A, B, opcode, result
            , zero);

        // Test input values
```

```
22          A = 4'b0011; B = 4'b0001; // A=3, B=1
23
24          // Apply each ALU operation
25          opcode = 3'b000; #10; // ADD  (3+1=4)
26          opcode = 3'b001; #10; // SUB  (3-1=2)
27          opcode = 3'b010; #10; // AND  (0011 & 0001 = 0001)
28          opcode = 3'b011; #10; // OR   (0011 | 0001 = 0011)
29          opcode = 3'b100; #10; // XOR  (0011 ^ 0001 = 0010)
30          opcode = 3'b101; #10; // NOT  (~0011 = 1100)
31          opcode = 3'b110; #10; // Shift Left (0011 << 1 = 0110)
32          opcode = 3'b111; #10; // Shift Right (0011 >> 1 = 0001)
33
34          $finish; // End simulation
35      end
36  endmodule
```

5. Simulation and Results

The ALU was tested using Xilinx Vivado to verify functionality across all supported operations. Sample inputs were applied and corresponding outputs were validated using waveform inspection tools.

5.1 ALU Operation Table

Table A.11 presents the list of supported operations by the ALU based on the 3-bit opcode. Each opcode selects a specific arithmetic or logical function to be performed on the inputs A and B.

Table A.11: Opcode mapping for basic ALU operations

Opcode	Operation
000	A + B (Addition)
001	A - B (Subtraction)
010	A AND B
011	A OR B
100	A XOR B
101	NOT A
110	A Shift Left by 1
111	A Shift Right by 1

5.2 Sample Simulation Results

- For inputs `A = 8'b10101010`, `B = 8'b01010101`:

  - Addition (000): `11111111`
  - Subtraction (001): `01010101`
  - AND (010): `00000000`
  - OR (011): `11111111`
  - XOR (100): `11111111`
  - NOT A (101): `01010101`
  - Shift Left (110): `01010100`
  - Shift Right (111): `01010101`

- All operations were verified using Vivado waveform viewer and matched expected behavior.
- No functional errors or timing issues were observed under a 100 MHz clock constraint.

6. Exercises

1. Add overflow detection for addition and subtraction.
2. Extend the ALU to 8-bit width.
3. Implement increment and decrement operations.
4. Use a parameterized opcode width and input size.
5. Design a register file to connect to the ALU inputs and outputs.

7. Conclusion

This lab introduced the design of a basic ALU using Verilog HDL, covering both arithmetic and logic functions. The simulation confirmed functional correctness, and the design serves as a foundation for more complex datapath and processor projects.

8. Instructor Notes

- Emphasize the use of 'always @(*)' for combinational logic.
- Discuss signed vs. unsigned arithmetic.
- Highlight ALU integration into larger datapath architectures.

A.9 Lab 9: Counters and Clock Divider Design

1. Lab Objective

- To design and simulate synchronous binary counters using Verilog HDL.
- To implement a clock divider to generate slower clocks from a system clock.
- To verify functionality through simulation and waveform observation.

2. Background Theory

2.1 Synchronous Counter

A counter is a sequential circuit that goes through a predefined sequence of states. A synchronous counter updates its state on the rising (or falling) edge of a clock signal. Types of counters include:

- Up Counter
- Down Counter
- Up-Down Counter

2.2 Clock Divider

A clock divider reduces the frequency of the system clock by a programmable or fixed factor. It is implemented using counters that toggle output after a fixed number of clock cycles.

3. Verilog Implementation

3.1 4-bit Synchronous Up Counter

```
1  // 4-bit Counter:
2  // increments on each clock edge, resets to 0 when rst=1
3  module counter_4bit (
4      input clk,              // Clock input
5      input rst,              // Asynchronous reset input
6      output reg [3:0] count  // 4-bit counter output
7  );
8      always @(posedge clk or posedge rst) begin
9          if (rst)
10             count <= 4'b0000;    // Reset counter to 0
```

```
        else
          // Increment counter by 1 on each clock pulse
            count <= count + 1;
    end
endmodule
```

3.2 Clock Divider (Divide by 2^N)

```
// Clock Divider: divides input clock frequency by 2^N
module clock_divider #(parameter N = 25) (
    input clk_in,        // Input clock
    input rst,           // Asynchronous reset
    output reg clk_out   // Divided output clock
);
    reg [N-1:0] count;   // Counter register

    always @(posedge clk_in or posedge rst) begin
        if (rst) begin
            count <= 0;        // Reset counter
            clk_out <= 0;      // Reset output clock
        end else begin
            count <= count + 1;       // Increment counter
          // Output toggles at MSB (divides clock)
            clk_out <= count[N-1];
        end
    end
endmodule
```

4. Testbenches

4.1 Testbench for 4-bit Counter

```
// Testbench for 4-bit Counter
module counter_tb;
    reg clk, rst;          // Clock and reset inputs
    wire [3:0] count;      // Counter output

    // Instantiate Unit Under Test (UUT)
    counter_4bit uut (.clk(clk), .rst(rst), .count(count));

    // Clock and reset generation
```

```
    initial begin
        clk = 0; rst = 1; #5;        // Start with reset active
        rst = 0;                     // Release reset
        forever #5 clk = ~clk;       // Toggle clock every 5 time
           units
    end

    // Monitor and display counter output
    initial begin
       // Show simulation time and counter value
        $monitor("Time = %0t | Count = %b", $time, count);
        #100 $finish; // End simulation after 100 time units
    end
endmodule
```

4.2 Testbench for Clock Divider

```
// Testbench for Clock Divider
module clock_divider_tb;
    reg clk_in, rst;      // Input clock and reset
    wire clk_out;         // Divided clock output

    // Instantiate Unit Under Test (UUT) with N=4
    clock_divider #(4) uut (.clk_in(clk_in), .rst(rst), .clk_out(
       clk_out));

    // Clock and reset generation
    initial begin
        clk_in = 0; rst = 1; #5;      // Start with reset active
        rst = 0;                      // Release reset
        forever #5 clk_in = ~clk_in; // Toggle input clock every
           5 time units
    end

    // Monitor and display output
    initial begin
       // Show time and divided clock
        $monitor("Time = %0t | clk_out = %b", $time, clk_out);
        #200 $finish; // End simulation after 200 time units
    end
endmodule
```

5. Simulation and Results

The simulation timings for the 4-bit counter and clock divider are summarized in Tables A.12 . This table illustrates the waveform behavior of digital circuits over 20 clock cycles.

Table A.12: Wave timing simulation for 4-bit counter over 20 cycles

Cycle	Clock	Counter [3:0]	Divided Clock Output
0	0	0000	0
1	1	0001	0
2	0	0001	0
3	1	0010	0
4	0	0010	1
5	1	0011	1
6	0	0011	1
7	1	0100	1
8	0	0100	0
9	1	0101	0
10	0	0101	0
11	1	0110	0
12	0	0110	1
13	1	0111	1
14	0	0111	1
15	1	1000	1
16	0	1000	0
17	1	1001	0
18	0	1001	0
19	1	1010	0

6. Exercises

1. Modify the counter to create a down counter and an up-down counter.
2. Implement a counter with a terminal count that resets automatically.
3. Use the clock divider to control the blinking rate of an LED on an FPGA board.
4. Design a clock divider with selectable division factor using inputs.
5. Extend the counter to 8 bits and verify wrap-around behavior.

7. Conclusion

This lab demonstrated the design and simulation of a 4-bit counter and a clock divider using Verilog HDL. These components are essential in sequential logic design, digital

timing control, and interface timing generation.

8. Instructor Notes

- Emphasize difference between asynchronous and synchronous resets.
- Discuss FPGA timing constraints for clock generation and division.
- Highlight overflow behavior and terminal count detection.

A.10 Lab 10: Display Counter on FPGA

1. Lab Objective

- To design a 2-digit decimal counter and interface it with a 7-segment display.
- To apply multiplexing to control multiple digits with limited FPGA I/O.
- To simulate and implement the design on a real FPGA board.

2. Background Theory

A 7-segment display consists of seven LEDs labeled 'a' through 'g', arranged to display numeric digits. For multiple digits, multiplexing is used: only one digit is enabled at a time, but they are toggled fast enough for persistence of vision.

Key Concepts:

- Binary counter increments every clock cycle.
- BCD conversion is required to display decimal numbers.
- Multiplexing alternates the active digit rapidly.

3. Verilog Design Overview

3.1 BCD Counter (0–99)

```
// BCD Counter (00-99):
// counts decimal digits using two 4-bit registers (tens, ones)
module bcd_counter (
    input clk,              // Clock input
    input rst,              // Asynchronous reset
    output reg [3:0] tens, // Tens digit (0-9)
    output reg [3:0] ones  // Ones digit (0-9)
);
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            ones <= 0;  // Reset ones digit to 0
            tens <= 0;   // Reset tens digit to 0
        end else begin
            if (ones == 9) begin
             // Reset ones digit when it reaches 9
                ones <= 0;
```

```
                if (tens == 9)
              // Reset tens digit when it reaches 9 (rollover 99
                 -> 00)
                    tens <= 0;
                else
                    tens <= tens + 1; // Increment tens digit
            end else begin
                ones <= ones + 1;     // Increment ones digit
            end
        end
    end
endmodule
```

3.2 7-Segment Decoder

```
// 7-Segment Display Decoder:
// converts 4-bit BCD digit into 7-segment pattern
module seg_decoder (
    input [3:0] digit,    // 4-bit input (0-9)
    output reg [6:0] seg // 7-segment output (abcdefg)
);
    always @(*) begin
        case (digit)
            4'd0: seg = 7'b1000000; // Display "0"
            4'd1: seg = 7'b1111001; // Display "1"
            4'd2: seg = 7'b0100100; // Display "2"
            4'd3: seg = 7'b0110000; // Display "3"
            4'd4: seg = 7'b0011001; // Display "4"
            4'd5: seg = 7'b0010010; // Display "5"
            4'd6: seg = 7'b0000010; // Display "6"
            4'd7: seg = 7'b1111000; // Display "7"
            4'd8: seg = 7'b0000000; // Display "8"
            4'd9: seg = 7'b0010000; // Display "9"
            default: seg = 7'b1111111; //All segments off (blank)
        endcase
    end
endmodule
```

3.3 Multiplexed Display Control

```
// Multiplexed 2-digit 7-Segment Display Controller
module mux_display (
    input clk,              // Clock input
    input rst,              // Asynchronous reset
    input [3:0] digit1,     // Right digit (ones)
    input [3:0] digit2,     // Left digit (tens)
    output reg [6:0] seg, // Segment outputs (abcdefg)
    output reg [3:0] an    // Anode control (active low for 4
       digits)
);
    reg toggle;             // Toggles between digit1 and digit2
    wire [6:0] seg1, seg2; // Segment codes for digit1 and digit2

    // Instantiate two 7-segment decoders
    seg_decoder dec1 (.digit(digit1), .seg(seg1));
    seg_decoder dec2 (.digit(digit2), .seg(seg2));

    reg [15:0] clkdiv;  // Clock divider for multiplexing speed

    // Clock divider logic
    always @(posedge clk or posedge rst) begin
        if (rst)
            clkdiv <= 0;          // Reset divider
        else
            clkdiv <= clkdiv + 1;  // Increment divider
    end

    // Toggle between digits based on divided clock
    always @(posedge clkdiv[15]) begin
        toggle <= ~toggle;
    end

    // Multiplexing logic: alternate between digit1 and digit2
    always @(*) begin
        if (toggle) begin
            seg = seg1;      // Display right digit
            an = 4'b1110;    // Enable anode for digit1 (rightmost
               )
        end else begin
            seg = seg2;      // Display left digit
```

```
            an = 4'b1101;  // Enable anode for digit2 (next digit
               )
        end
    end
endmodule
```

3.4 Top-Level Integration

```
// Top module: BCD counter + 7-segment display multiplexer
module top_display_counter (
    input clk,         // Clock input
    input rst,         // Reset input
    output [6:0] seg, // 7-segment outputs (abcdefg)
    output [3:0] an   // Anode control for 4-digit display (
       active low)
);
    wire [3:0] ones, tens;//Wires for BCD digits (ones and tens)

    // Instantiate BCD counter (00-99)
    bcd_counter bcd (
        .clk(clk),
        .rst(rst),
        .tens(tens),
        .ones(ones)
    );

    // Instantiate display multiplexer
   // (shows two digits on 7-segment display)
    mux_display mux (
        .clk(clk),
        .rst(rst),
        .digit1(ones),  // Ones digit
        .digit2(tens),  // Tens digit
        .seg(seg),
        .an(an)
    );
endmodule
```

4. FPGA Pin Mapping (Nexys A7)

- `clk`: `W5` (100 MHz system clock)

- `rst`: Connect to a push button (e.g., `T18`)
- `seg[6:0]`: `AE26, AC26, AB26, AB25, AA26, Y25, Y26`
- `an[3:0]`: `W22, V22, U21, U22`

5. Simulation and Results

Tables A.13 and A.14 together demonstrate the correctness of the counter and its integration with the 7-segment display, where the BCD values are accurately converted and multiplexed for real-time visual output.

5.1 BCD Counter Simulation Table

The simulation below shows how the BCD counter increments 'ones' and 'tens' digits every clock cycle, wrapping correctly from 09 to 10:

Table A.13: Simulation output of BCD counter (0–15)

Clock Cycle	Tens (BCD)	Ones (BCD)
0	0	0
1	0	1
...	...	...
9	0	9
10	1	0
11	1	1
...	...	...
15	1	5

5.2 7-Segment Multiplexing Results

The output segments ('seg') are toggled between two digits at high frequency using the 'toggle' bit and divided clock. The simulation confirms correct segment encoding:

Table A.14: Simulation output of multiplexed 7-segment display

Time (ns)	Toggle	Active Digit	Segment Code (seg)
1000	0	Tens	7'b1000000 (shows '0')
2000	1	Ones	7'b1111001 (shows '1')
3000	0	Tens	7'b1000000 (shows '0')
4000	1	Ones	7'b0100100 (shows '2')

Note: When implemented on the Nexys A7 board, the display counts from '00' to '99' and rolls over. Due to multiplexing, the user sees both digits stable on the display due to persistence of vision.

6. Implementation and Testing

1. Use Vivado to create the project and add the top module and submodules.
2. Assign pins using the XDC constraints file for your board.
3. Program the FPGA and observe the counter on the 2-digit display.

7. Exercises

1. Modify the design to count down from 99 to 00.
2. Add a pause/resume button for the counter.
3. Add a clock divider for slower counting (1 Hz).
4. Expand the design to 4 digits (0000–9999).
5. Display hexadecimal values instead of decimal (0–FF).

8. Conclusion

This lab demonstrated the control of a 2-digit 7-segment display using a binary-to-BCD counter and digit multiplexing. It introduced a complete display subsystem for real-time counting, applicable in timers, counters, and embedded interfaces.

9. Instructor Notes

- Emphasize the concept of human visual persistence in multiplexing.
- Encourage modular design and simulation of each unit separately.
- Provide XDC templates for faster pin assignment.

A.11 Lab 11: Multiplier Design

1. Lab Objective

- To design and implement a binary multiplier circuit using Verilog HDL.
- To understand the principles of combinational and sequential multiplication.
- To verify functionality using simulation and waveform observation.

2. Background Theory

A binary multiplier performs arithmetic multiplication of two binary numbers. There are two main approaches:

- **Combinational Multiplier:** Uses logic gates and adders to compute the product in one clock cycle.
- **Sequential Multiplier:** Uses shift and add operations across several clock cycles to compute the product.

Unsigned Binary Multiplication:

$$\text{Product} = A \times B$$

Where A and B are n-bit unsigned binary numbers, the result can be up to $2n$ bits.

3. Verilog Implementations

3.1 4-bit Combinational Multiplier

```
// 4-bit Multiplier:
// multiplies two 4-bit numbers to produce 8-bit product
module multiplier_4bit (
    input [3:0] A,       // 4-bit input A
    input [3:0] B,       // 4-bit input B
    output [7:0] P       // 8-bit product output
);
    assign P = A * B;    // Multiply A and B
endmodule
```

3.2 4-bit Sequential Multiplier (Shift and Add)

```
// Sequential 4-bit Multiplier:
// performs shift-and-add multiplication
module seq_multiplier_4bit (
    input clk,          // Clock input
    input rst,          // Asynchronous reset
    input start,         // Start signal to begin multiplication
    input [3:0] A,       // Multiplicand (4-bit)
    input [3:0] B,       // Multiplier (4-bit)
    output reg [7:0] product,// Final product (8-bit)
    output reg done  // Done flag (high when multiplication
       finishes)
);
    reg [3:0] count;     // Step counter (0-4)
    reg [3:0] multiplicand, multiplier; // Registers for inputs
    reg [7:0] temp;      // Temporary accumulator

    always @(posedge clk or posedge rst) begin
        if (rst) begin
          // Reset all registers
            count <= 0;
            temp <= 0;
            done <= 0;
            product <= 0;
        end else if (start && !done) begin
            if (count == 0) begin
                // Initialize values at start
                multiplicand <= A;
                multiplier <= B;
                temp <= 0;
            end

        // If LSB of multiplier = 1, add shifted multiplicand
            if (multiplier[0] == 1)
                temp = temp + (multiplicand << count);

        // Shift multiplier right, increment count
            multiplier = multiplier >> 1;
            count = count + 1;

          // Finish after 4 cycles
```

```
            if (count == 4) begin
                product <= temp; // Save result
                done <= 1;       // Signal completion
            end
        end
    end
endmodule
```

4. Testbenches

4.1 Testbench for Combinational Multiplier

```
// Testbench for 4-bit Multiplier
module multiplier_tb;
    reg [3:0] A, B;      // 4-bit inputs
    wire [7:0] P;        // 8-bit product output

    // Instantiate Unit Under Test (UUT)
    multiplier_4bit uut (.A(A), .B(B), .P(P));

    initial begin
        // Monitor signals during simulation
        $monitor("A=%b, B=%b, P=%b", A, B, P);

        // Apply test cases
        A = 4'd3; B = 4'd4; #10;  // Expect 3*4=12
        A = 4'd7; B = 4'd2; #10;  // Expect 7*2=14
        A = 4'd5; B = 4'd5; #10;  // Expect 5*5=25

        $finish; // End simulation
    end
endmodule
```

4.2 Testbench for Sequential Multiplier

```
// Testbench for
// Sequential 4-bit Multiplier (shift-and-add method)
module seq_multiplier_tb;
    reg clk, rst, start;  // Clock, reset, and start control
    reg [3:0] A, B;       // 4-bit inputs
    wire [7:0] product;   // 8-bit product output
```

```
    wire done; // Done flag (high when multiplication finishes)

    // Instantiate Unit Under Test (UUT)
    seq_multiplier_4bit uut (
        .clk(clk), .rst(rst), .start(start),
        .A(A), .B(B),
        .product(product), .done(done)
    );

    // Clock generation: toggle every 5 time units
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    // Apply test cases
    initial begin
        rst = 1; start = 0; A = 0; B = 0; #10;  // Reset
        rst = 0;
        A = 4'd3; B = 4'd5; start = 1; #100;    // Test 3 * 5
      // Display result after done
        $display("Product = %d", product);
        $finish; // End simulation
    end
endmodule
```

5. Simulation and Results

5.1 Combinational Multiplier Output Table

Table A.15 shows the results from the simulation of the 4-bit combinational multiplier. The product appears immediately after inputs are applied since it is purely combinational logic.

Table A.15: Simulation output for 4-bit combinational multiplier

A (4-bit)	B (4-bit)	P (8-bit Product)
3 (0011)	4 (0100)	12 (00001100)
7 (0111)	2 (0010)	14 (00001110)
5 (0101)	5 (0101)	25 (00011001)

5.2 Sequential Multiplier Behavior

The sequential multiplier uses shift-and-add logic, requiring 4 clock cycles to complete the multiplication. Table A.16 presents a sample output from the waveform for inputs A = 3 and B = 5.

Table A.16: Clock-cycle based operation of sequential multiplier (A=3, B=5)

Cycle	Multiplier Bit	Shifted Multiplicand	Temp Sum	Carry Out
0	1	00000011	00000011	0
1	0	00000110	00000011	0
2	1	00001100	00001111	0
3	0	00011000	00001111	0
Final Product (after 4 cycles)				**00001111 (15)**

Note:

- **Combinational Multiplier:** Produces result instantly (single-cycle delay).
- **Sequential Multiplier:** Requires 4 cycles for a 4-bit operation.
- Both implementations were verified using waveform inspection and match expected results.

6. Exercises

1. Extend the multiplier to handle signed numbers using two's complement.
2. Design an 8-bit multiplier using a hierarchical approach.
3. Implement a pipelined multiplier and compare performance.
4. Simulate all $A \times B$ combinations for $A, B \in [0, 15]$.
5. Compare resource usage between combinational and sequential designs.

7. Conclusion

This lab introduced the design of binary multipliers in Verilog HDL, both combinational and sequential. Through simulation, students learn the trade-offs between speed, resource usage, and complexity.

8. Instructor Notes

- Reinforce understanding of data path width and overflow.
- Emphasize testbench design and proper timing simulation.
- Suggest using waveform viewers (e.g., GTKWave) for clarity.

A.12 Lab 12: FSM Design for Serial Adder

1. Lab Objective

- To design a serial adder using Finite State Machine (FSM) methodology in Verilog HDL.
- To simulate and verify the functionality of the FSM-based serial adder.
- To understand the control mechanism for sequential arithmetic using FSM design.

2. Background Theory

2.1 Serial Adder

A serial adder performs bit-by-bit binary addition using a single full adder and a register to store the carry. It processes one bit per clock cycle, starting from the least significant bit (LSB).

2.2 FSM Design Approach

A finite state machine consists of:

- A set of states
- Inputs and outputs
- A state transition function
- An output function

FSMs are classified into:

- Mealy Machine: Output depends on current state and inputs.
- Moore Machine: Output depends only on current state.

The serial adder FSM keeps track of the carry and controls bit-by-bit addition over multiple clock cycles.

3. Verilog Implementation

3.1 FSM-Based Serial Adder (4-bit)

```
// Serial 4-bit Adder using FSM (Finite State Machine)
module serial_adder_fsm (
    input clk,              // Clock input
    input rst,              // Asynchronous reset
    input start,            // Start signal
    input [3:0] A,          // 4-bit input A
    input [3:0] B,          // 4-bit input B
    output reg [3:0] SUM, // 4-bit sum output
    output reg done         // Done flag (high when addition
       completes)
);
    reg [2:0] bit_cnt;     // Bit counter (0-3)
    reg carry;             // Carry bit
    reg [3:0] regA, regB; // Internal registers for inputs
    reg [1:0] state;       // FSM state register

    // FSM state encoding
    localparam IDLE = 2'b00,  // Waiting for start
               LOAD = 2'b01,  // Load inputs
               ADD  = 2'b10,  // Perform bit-by-bit addition
               DONE = 2'b11;  // Addition complete

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            // Reset all signals
            state   <= IDLE;
            SUM     <= 0;
            carry   <= 0;
            done    <= 0;
            bit_cnt <= 0;
        end else begin
            case (state)
                IDLE: begin
                    if (start) state <= LOAD; // Wait for start
                       signal
                end
                LOAD: begin
                    // Initialize registers and reset counters
                    regA <= A;
                    regB <= B;
                    bit_cnt <= 0;
```

```
                    SUM <= 0;
                    carry <= 0;
                    state <= ADD;
                end
                ADD: begin
                    // Add corresponding bits + carry
                    {carry, SUM[bit_cnt]} <= regA[bit_cnt] + regB
                       [bit_cnt] + carry;
                    bit_cnt <= bit_cnt + 1; // Move to next bit
                    if (bit_cnt == 3)
                        state <= DONE;      // After last bit ->
                           DONE
                end
                DONE: begin
                    done <= 1;  // Signal completion
                    state <= IDLE; // Return to idle
                end
            endcase
        end
    end
endmodule
```

4. Testbench for Serial Adder FSM

4.1 Testbench Code

```
// Testbench for Serial 4-bit Adder FSM
module serial_adder_fsm_tb;
    reg clk, rst, start;        // Clock, reset, start control
    reg [3:0] A, B;             // 4-bit inputs
    wire [3:0] SUM;             // 4-bit sum output
    wire done;                  // Done flag

    // Instantiate Unit Under Test (UUT)
    serial_adder_fsm uut (
        .clk(clk), .rst(rst), .start(start),
        .A(A), .B(B), .SUM(SUM), .done(done)
    );

    // Clock generation: toggle every 5 time units
    initial begin
```

```
16          clk = 0;
17          forever #5 clk = ~clk;
18      end
19
20      // Test stimulus
21      initial begin
22         // Apply reset
23          rst = 1; start = 0; A = 4'b0000; B = 4'b0000; #10;
24          rst = 0;
25
26          // Test case 1: 6 + 3
27          A = 4'b0110; B = 4'b0011; start = 1; #10; // Trigger
                start
28          start = 0;
29          wait (done); #20; // Wait until addition completes
30
31          // Test case 2: 15 + 1
32          A = 4'b1111; B = 4'b0001; start = 1; #10;
33          start = 0;
34          wait (done); #20;
35
36          $finish; // End simulation
37      end
38      // Monitor signals during simulation
39      initial begin
40          $monitor("Time=%0t | A=%b B=%b SUM=%b done=%b",
41                   $time, A, B, SUM, done);
42      end
43  endmodule
```

5. Simulation and Results

5.1 Sample Simulation Output

Table A.17 shows the simulation results for two input cases. The 'SUM' output appears once all bits have been added over multiple clock cycles, and the 'done' signal goes high to indicate successful completion.

Explanation of Simulation Steps:

1. **Initialization (Time = 0–10 ns):** The system is reset and then receives the first input operands: A = 0110 (6 in decimal) and B = 0011 (3 in decimal). The 'start' signal is asserted to trigger the FSM transition from IDLE to LOAD.

Table A.17: Simulation output for FSM-based serial adder

Time (ns)	A	B	SUM	done
10	0110	0011	—-	0
30	0110	0011	1001	1
60	1111	0001	0000	1

2. **Bit-by-Bit Addition (Time = 10–30 ns):** The FSM enters the ADD state. Each clock cycle, it adds one pair of bits from A and B (starting from LSB), along with the carry from the previous cycle. After 4 clock cycles, the sum is fully computed: $6 + 3 = 9 \rightarrow 1001$. The 'done' signal is asserted high, indicating that the result is valid.

3. **Second Test Case (Time = 30–60 ns):** A new input pair is loaded: A = 1111 (15) and B = 0001 (1). The FSM repeats the same ADD state sequence. Since the result is 16, which exceeds the 4-bit range (maximum = 15), the sum wraps around to 0000, demonstrating overflow behavior. The 'done' signal again indicates completion.

6. Exercises

1. Modify the FSM to support 8-bit serial addition.
2. Add carry-out as a separate output from the most significant bit.
3. Add a ready signal to prevent multiple triggers of the FSM while busy.
4. Convert the FSM from Mealy to Moore implementation and compare.
5. Implement the serial adder using shift registers and a single full adder module.

7. Conclusion

This lab demonstrates the use of finite state machines for implementing sequential operations, using a serial adder as an example. The FSM structure enables cycle-by-cycle control over data processing and provides insights into the control logic of digital systems.

8. Instructor Notes

- Emphasize FSM diagram design before Verilog coding.
- Show differences between serial and parallel adder architectures.
- Discuss FSM state encoding and optimization techniques.

A.13 Lab 13: Traffic Light Controller

1. Lab Objective

- To design and simulate a traffic light controller using Verilog HDL.
- To apply FSM-based sequential logic design.
- To implement the design on an FPGA development board.

2. Background Theory

A traffic light controller operates based on a fixed or adaptive sequence to regulate traffic at intersections. It typically follows a deterministic state machine pattern:

- Green → Yellow → Red
- Time delays determine how long each light remains active.

This lab uses a **Moore FSM model**, where the output depends only on the current state.

3. Traffic Light State Diagram

- **S0**: Green (main road) – 5 seconds
- **S1**: Yellow (main road) – 2 seconds
- **S2**: Red (main road) – 5 seconds

Cycle repeats every 12 seconds. Clock frequency is assumed to be 1 Hz (1 second tick).

4. Verilog Implementation

4.1 FSM Code

```
// Traffic Light Controller using FSM
// States cycle: Green -> Yellow -> Red -> Green
module traffic_light_fsm (
    input clk,              // Clock input
    input rst,              // Asynchronous reset
    output reg [2:0] lights // {Red, Yellow, Green} output
);
```

```
    // Define FSM states
    typedef enum reg [1:0] {S0, S1, S2} state_t;
    state_t current_state, next_state;
    reg [3:0] timer; // Countdown timer for each light duration

    // State transition and timer update
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            current_state <= S0; // Start at Green
            timer <= 0;
        end else begin
            if (timer == 0) begin
                // Move to next state and load timer
                current_state <= next_state;
                case (next_state)
                    S0: timer <= 5; // Green for 5 cycles
                    S1: timer <= 2; // Yellow for 2 cycles
                    S2: timer <= 5; // Red for 5 cycles
                endcase
            end else begin
                timer <= timer - 1; // Decrement timer
            end
        end
    end

    // Output logic based on current state
    always @(*) begin
        case (current_state)
            // Green active
            S0: begin lights = 3'b001; next_state = S1; end
           // Yellow active
            S1: begin lights = 3'b010; next_state = S2; end
          // Red active
            S2: begin lights = 3'b100; next_state = S0; end
          // Default off
            default: begin lights = 3'b000; next_state = S0; end
        endcase
    end
endmodule
```

5. Testbench Example

```
// Testbench for Traffic Light FSM
module tb_traffic_light;
    reg clk = 0, rst = 1;      // Clock and reset signals
    wire [2:0] lights;         // {Red, Yellow, Green} outputs

    // Instantiate Unit Under Test (UUT)
    traffic_light_fsm uut (.clk(clk), .rst(rst), .lights(lights))
       ;

    // Clock generation: toggle every 5 time units
    always #5 clk = ~clk;

    // Test sequence
    initial begin
        // Display light states
        $monitor("Time=%t, State={R,Y,G}=%b", $time, lights);
        #15 rst = 0;     // Release reset after 15 time units
        #200 $finish;    // Run simulation for 200 time units
    end
endmodule
```

6. FPGA Mapping (e.g., Nexys A7)

```
## Green LED
set_property PACKAGE_PIN H17 [get_ports {lights[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lights[0]}]

## Yellow LED
set_property PACKAGE_PIN N14 [get_ports {lights[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lights[1]}]

## Red LED
set_property PACKAGE_PIN U17 [get_ports {lights[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lights[2]}]

## Clock and Reset
set_property PACKAGE_PIN E3 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

```

```
17 set_property PACKAGE_PIN P17 [get_ports rst]
18 set_property IOSTANDARD LVCMOS33 [get_ports rst]
```

7. Simulation and Results

Table A.18 shows a simplified simulation output where the FSM transitions through the Green, Yellow, and Red states based on the timer logic. The 'lights' output indicates the active light at each stage using a 3-bit encoding: {Red, Yellow, Green}.

Table A.18: Sample FSM output for traffic light controller

Time (ns)	**lights [R,Y,G]**
0	001 (Green)
50	010 (Yellow)
70	100 (Red)
120	001 (Green)

As shown in the simulation table, the FSM progresses through the traffic light states in the correct order and intervals. The transitions occur every few simulation steps, and each output corresponds to a visible LED color on the FPGA board.

8. Exercises

1. Modify FSM for two-way intersection (Main and Side road).
2. Add pedestrian walk signal with button interrupt.
3. Use 1 kHz input clock with a counter-based divider for 1 Hz tick.
4. Add buzzer output for pedestrian crossing alert.
5. Display countdown on 7-segment display for each light.

9. Conclusion

This lab demonstrated the implementation of a finite state machine for traffic light control using Verilog. Students practiced state encoding, timing control, and FPGA pin mapping for real-world interfacing.

10. Instructor Notes

- Discuss synchronous vs asynchronous reset behavior.
- Encourage waveform validation using simulation tools.
- Let students experiment with different timing intervals.

A.14 Lab 14: UART Communication

1. Lab Objective

- To understand the Universal Asynchronous Receiver Transmitter (UART) protocol.
- To design UART Transmitter and Receiver modules using Verilog HDL.
- To simulate UART behavior and optionally interface with a serial terminal via FPGA.

2. Background Theory

2.1 UART Overview

UART is a hardware communication protocol used for asynchronous serial communication. It converts parallel data into serial format (TX) and vice versa (RX) without requiring a separate clock signal.

Standard UART Frame:

- 1 start bit (0)
- 8 data bits (LSB first)
- 1 stop bit (1)

Common Parameters:

- Baud rate: 9600, 115200 bps, etc.
- No parity
- 1 start, 1 stop bit

3. Verilog Implementation

Assume a 100 MHz system clock and a baud rate of 9600.

3.1 Baud Rate Generator

```
// Baud Rate Generator: generates a tick at desired baud rate
module baud_gen (
    input clk,    // System clock input
    input rst,     // Asynchronous reset
    output reg tick  // Output tick pulse (1 clk wide)
```

```
);
    parameter BAUD_DIV = 10416; // Divider for 100 MHz -> 9600
        baud

    reg [13:0] count;   // Counter for clock division

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            count <= 0; // Reset counter
            tick <= 0;  // Reset tick
        end else begin
            if (count == BAUD_DIV - 1) begin
                count <= 0; // Reset counter after full period
                tick <= 1;  // Generate tick pulse
            end else begin
                count <= count + 1; // Increment counter
                tick <= 0;   // Keep tick low
            end
        end
    end
endmodule
```

3.2 UART Transmitter

```
// UART Transmitter
// Sends 1 start bit, 8 data bits (LSB first), and 1 stop bit
module uart_tx (
    input clk,      // System clock
    input rst,      // Asynchronous reset
    input tx_start,  // Start transmission signal
    input [7:0] tx_data,  // Data byte to transmit
    input tick,      // Baud tick from baud generator
    output reg tx,   // Serial transmit line
    output reg busy // Busy flag (1 = transmitting)
);
    reg [3:0] state; //State: 0=Idle, 1=Data, 2=Stop, 3=Done
    reg [2:0] bit_cnt; // Bit counter for 8 data bits
    reg [7:0] data;  // Data register (holds tx_data)

    always @(posedge clk or posedge rst) begin
        if (rst) begin
```

```
            tx <= 1;    // Idle line is high
            busy <= 0;     // Not transmitting
            state <= 0;    // Start in idle
        end else if (tick) begin // Advance only on baud tick
            case (state)
                0: if (tx_start) begin
                    busy <= 1;
                    data <= tx_data;  // Load data to send
                    tx <= 0;     // Send Start bit (low)
                    state <= 1;
                    bit_cnt <= 0;
                end
                1: begin // Transmit 8 data bits (LSB first)
                    tx <= data[bit_cnt];    // Output current bit
                    bit_cnt <= bit_cnt + 1;
                    if (bit_cnt == 7) state <= 2; // After last
                       bit -> stop
                end
                2: begin // Stop bit
                    tx <= 1;  // Stop bit is high
                    state <= 3;
                end
                3: begin // Return to idle
                    busy <= 0;
                    state <= 0;
                end
            endcase
        end
    end
endmodule
```

3.3 UART Receiver

```
// UART Receiver
// Receives 1 start bit, 8 data bits (LSB first), and 1 stop bit
module uart_rx (
    input clk,  // System clock
    input rst,  // Asynchronous reset
    input rx,    // Serial receive line
    input tick,  // Baud tick from baud generator
    output reg [7:0] rx_data, // Received data byte
```

```
    output reg rx_done   // Done flag (1 when byte received)
);
    reg [3:0] state;  // State: 0=Idle, 1=Data, 2=Stop, 3=Done
    reg [2:0] bit_cnt; // Bit counter for 8 data bits
    reg [7:0] data;  // Temporary storage for received bits

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= 0;   // Start in idle state
            rx_done <= 0; // Clear done flag
        end else if (tick) begin // Advance on baud tick
            case (state)
                0: if (!rx) state <= 1; // Detect start bit (rx
                   goes low)
                1: begin // Receive 8 data bits
                    data[bit_cnt] <= rx;  // Sample current bit
                    bit_cnt <= bit_cnt + 1; // Increment bit
                       counter
                    if (bit_cnt == 7) state <= 2; // After 8 bits
                        -> stop
                end
                2: begin // Stop bit
                    rx_data <= data; // Store received data
                    rx_done <= 1;    // Assert done flag
                    state <= 3;
                end
                3: begin // Return to idle
                    rx_done <= 0; // Clear done flag
                    state <= 0;
                end
            endcase
        end
    end
endmodule
```

4. Testbench for UART TX and RX

```
// Testbench for UART (TX + RX loopback)
module uart_tb;
    reg clk = 0, rst = 0;  // Clock and reset
```

```
4     reg [7:0] tx_data = 8'h55;  // Data to transmit (0x55 =
          01010101)
5     reg tx_start = 0;   // Start transmission signal
6     wire tx, rx;   // TX and RX serial lines (looped back)
7     wire [7:0] rx_data;   // Received data
8     wire tick, rx_done, busy;   // Baud tick, RX done flag, TX
          busy flag
9
10    // Loopback: connect TX -> RX
11    assign rx = tx;
12
13    // Instantiate Baud Generator (tick for 9600 baud @ 100 MHz)
14    baud_gen #(10416) baud (.clk(clk), .rst(rst), .tick(tick));
15
16    // Instantiate UART Transmitter
17    uart_tx tx_inst (
18        .clk(clk), .rst(rst),
19        .tx_start(tx_start), .tx_data(tx_data),
20        .tick(tick), .tx(tx), .busy(busy)
21    );
22
23    // Instantiate UART Receiver
24    uart_rx rx_inst (
25        .clk(clk), .rst(rst),
26        .rx(tx), .tick(tick),
27        .rx_data(rx_data), .rx_done(rx_done)
28    );
29
30    // Clock generation: toggle every 5 time units
31    always #5 clk = ~clk;
32
33    // Test sequence
34    initial begin
35        rst = 1; #20; rst = 0;    // Apply reset
36        #50 tx_start = 1; #10 tx_start = 0; // Start transmitting
              0x55
37        #10000;    // Wait long enough for TX + RX to complete
38        $finish;     // End simulation
39    end
40
41    // Monitor signals
```

```
    initial begin
        $monitor("TX=%b | RX_DATA=%h | RX_DONE=%b", tx, rx_data,
            rx_done);
    end
endmodule
```

5. Implementation Notes (Optional FPGA Deployment)

- Use 'tx' and 'rx' pins mapped to Pmod UART (or USB-UART bridge) on the FPGA board.
- Connect to a PC via USB using a serial terminal (Baud = 9600).
- Send ASCII characters to observe loopback behavior.

In a Case Usig Nexys A7

- Use the on-board USB-UART bridge available via the USB-JTAG port on the Nexys A7.
- Map the `tx` and `rx` signals to the following pins:
  - `tx`: D4
  - `rx`: C4
- Ensure the constraint file (.xdc) includes the correct I/O standards:

```
    ## UART TX (FPGA to PC)
    set_property PACKAGE_PIN D4 [get_ports {tx}]
    set_property IOSTANDARD LVCMOS33 [get_ports {tx}]

    ## UART RX (PC to FPGA)
    set_property PACKAGE_PIN C4 [get_ports {rx}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rx}]
```

- Connect the Nexys A7 to a PC via the micro-USB port.
- Open a serial terminal (e.g., PuTTY, TeraTerm) at **9600 baud**, **8 data bits**, **no parity**, and **1 stop bit**.
- Send ASCII characters from the terminal and observe the loopback (if TX and RX are shorted or echoed in logic).

6. Simulation and Results

The simulation testbench connects the UART transmitter and receiver in a loopback configuration (i.e., 'rx = tx'). A single byte (0x55 = 01010101) is transmitted and received through this setup. Table A.19 shows the expected result from the simulation monitor output.

Table A.19: UART simulation output

Time (ns)	TX Line	RX Data (Hex)
0	Idle (1)	–
50	Start Bit (0)	–
60–130	Data Bits (01010101)	–
140	Stop Bit (1)	0x55

The waveform confirms the UART protocol operation:

- The transmitter sends 1 start bit (0), followed by 8 data bits (LSB first), and a stop bit (1).
- The receiver captures the serial stream, reconstructs the byte, and sets `rx_done` high when complete.

This demonstrates a working UART data transfer loop with correct timing and data reconstruction. Waveform inspection using Vivado or GTKWave shows serialized transmission and correct reception.

7. Exercises

1. Add parity bit generation and checking.
2. Support 7-bit or 9-bit data.
3. Create a full-duplex UART loopback system.
4. Create a finite state machine that sends"HELLO" repeatedly.
5. Interface UART with 7-segment display to show received byte.

8. Conclusion

This lab introduced UART communication using Verilog HDL. Students implemented the transmitter and receiver logic with baud rate control and verified functionality using testbenches. This project is foundational for serial data transfer in FPGA-based systems.

9. Instructor Notes

- Emphasize importance of baud timing accuracy.
- Provide pre-configured XDC file for UART I/O pins.
- Suggest using GTKWave or Vivado simulator for waveform inspection.

Index

About the Author

Dr. Komsan Kanjanasit is an Assistant Professor in Electrical Engineering at the College of Computing, Prince of Songkla University, Phuket Campus, Thailand. With a solid background in digital systems, embedded hardware, and wireless communication technologies, he has made significant contributions to both teaching and research. He also directs the *System Intelligence Laboratory*, focusing on research in physical computing and intelligent systems.

Dr. Kanjanasit received his degrees in Electrical Engineering with a specialization in electronics and communication technology. His research interests encompass metamaterial antennas and wave absorbers, electromagnetic simulation, and digital coding techniques for antenna design. He has published widely in international journals and conferences and actively supervises research projects at both undergraduate and graduate levels.

Dr. Kanjanasit has made significant contributions to academic curriculum development, with a particular focus on integrating IoT, embedded and connected systems, physics simulation and modeling, cyber-physical systems, and intelligent object design into digital engineering education. In addition, he supports courses in the Artificial Intelligence and Systems Engineering (AISE) program. He has also authored and contributed to book chapters for both B.Sc. and M.Sc. programs at the School of Science and Technology, Sukhothai Thammathirat Open University (STOU), including:

- **99714:** Cyber-Physical Systems and Applications
  Chapter 1: Principles of Cyber-Physical Systems
  Chapter 2: Architecture and Models of Cyber-Physical Systems

- **99313:** Wireless Communication and Networking
  Chapter 8: 5G and 6G Wireless Communication Technologies
  Chapter 9: Mobility Protocols and Mobile IP

- **99410:** Telecommunication System Design and Management
  Chapter 7: Internet of Things (IoT) Technology
  Chapter 14: Principles of Cyber-Physical Systems