RTL to GDSII Workshop

Using Open-Source Tools

Complete Workshop Coverage:

Introduction to RTL-to-GDSII Flow	Toolchain & Workflow Overview
Environment Setup & Unix Basics	TCL Scripting for EDA
High-Level Synthesis (Bambu)	RTL Design & Best Practices
Simulation & Verification	Logic Synthesis (Yosys)
Power Analysis (OpenSTA)	OpenROAD Installation
Physical Design (OpenROAD)	DRC/LVS/GDSII Verification

Workshop Lead: Your Name

Version: 1.0

Date: December 29, 2025

Contents

1 Introduction

1.1 Overview

The RTL-to-GDSII (Register Transfer Level to Graphic Data Stream Information Interchange) flow is the complete process of transforming a high-level hardware description into a physical layout ready for semiconductor fabrication. This workshop provides hands-on experience with open-source tools that enable this transformation.

1.2 What is RTL-to-GDSII?

The RTL-to-GDSII flow encompasses several critical stages:

1. **RTL Design** – Hardware description using languages like Verilog or VHDL
2. **Verification** – Functional simulation and validation
3. **Synthesis** – Conversion of RTL to gate-level netlist
4. **Floor Planning** – Arrangement of major functional blocks
5. **Placement** – Positioning of standard cells
6. **Clock Tree Synthesis** – Distribution of clock signals
7. **Routing** – Creation of metal interconnections
8. **Verification** – Physical verification (DRC, LVS, timing)
9. **GDSII Generation** – Final layout for fabrication

1.3 Why Open-Source Tools?

Open-source EDA (Electronic Design Automation) tools have revolutionized hardware design by:

- **Democratizing Access** – Making chip design accessible to students, researchers, and small companies
- **Transparency** – Allowing users to understand and modify tool behavior
- **Cost Reduction** – Eliminating expensive licensing fees
- **Community Support** – Fostering collaboration and rapid innovation
- **Educational Value** – Providing learning opportunities without financial barriers

1.4 Workshop Objectives

By the end of this workshop, participants will be able to:

- Set up and configure a complete open-source RTL-to-GDSII toolchain
- Design and verify digital circuits using Verilog
- Perform logic synthesis using Yosys
- Conduct physical design using OpenROAD
- Verify designs using open-source verification tools
- Generate GDSII files ready for fabrication
- Automate design flows using TCL scripting

1.5 Target Audience

This workshop is designed for:

- Digital design engineers transitioning to open-source tools
- Graduate students studying VLSI design
- Researchers exploring hardware design automation
- Educators preparing curriculum for chip design courses
- Hobbyists interested in semiconductor design

1.6 Prerequisites

Participants should have:

- Basic understanding of digital logic design
- Familiarity with Verilog or VHDL
- Linux command-line experience (basic level)
- Access to a Linux system (native or WSL)
- At least 20GB of free disk space for tools and libraries

1.7 Workshop Structure

The workshop follows a progressive learning path:

1. Environment setup and foundational tools
2. Scripting and automation techniques
3. Front-end design (HLS, RTL, verification)
4. Synthesis and optimization
5. Physical design and implementation
6. Verification and sign-off

Each module includes theoretical background, practical examples, hands-on exercises, and real-world design considerations.

2 Toolchain and Workflow

2.1 Open-Source EDA Toolchain

This workshop utilizes a comprehensive set of open-source tools that collectively provide a complete RTL-to-GDSII design flow. The table below summarizes each tool and its role in the design process.

Tool	Purpose	Key Features
High-Level Synthesis & RTL Design		
Bambu HLS	Convert C/C++ code to RTL (Verilog)	Automated scheduling, resource allocation, interface generation
Simulation & Verification		
Icarus Verilog	Verilog simulation and compilation	IEEE 1364 compliant, VCD output, fast simulation
GTKWave	Waveform visualization	VCD/FST support, signal search, measurement tools
Covered	Code coverage analysis	Line, toggle, FSM, and branch coverage metrics
Logic Synthesis		
Yosys	RTL synthesis and technology mapping	Optimization passes, Liberty (.lib) support, netlist generation
Timing & Power Analysis		
OpenSTA	Static timing analysis and power estimation	Setup/hold analysis, delay calculation, power reporting
Physical Design		
OpenROAD	Complete physical design flow	Floor planning, placement, CTS, routing, LEF/DEF support
Physical Verification		
Magic VLSI	Layout editing and DRC verification	Design rule checking, GDSII export, parasitic extraction
Netgen	Layout vs. Schematic (LVS) verification	Hierarchical comparison, mismatch reporting

Table 1: Open-source EDA tools used in the RTL-to-GDSII workshop

2.2 Process Design Kits (PDKs)

Standard cell libraries and technology files used in this workshop:

2.3 Typical Design Flow

The following steps represent the standard RTL-to-GDSII flow using these tools:

PDK	Technology	Description
FreePDK45	45nm	Academic predictive PDK from NC State University
SkyWater 130nm	130nm	Fully open-source PDK from Google & SkyWater
Nangate45	45nm	Open standard cell library for academic use

Table 2: Process Design Kits (PDKs) and standard cell libraries

Step	Task	Tools Used
1	Design Entry	Write RTL in Verilog or generate from C using Bambu HLS
2	Functional Verification	Simulate with Icarus Verilog, visualize with GTKWave, measure coverage with Covered
3	Logic Synthesis	Synthesize with Yosys, map to target technology library
4	Timing & Power Analysis	Analyze with OpenSTA, verify constraints
5	Physical Design	Floor plan, place, CTS, and route with OpenROAD
6	Physical Verification	DRC with Magic, LVS with Netgen
7	GDSII Generation	Export final layout using Magic VLSI

Table 3: RTL-to-GDSII design flow stages

2.4 Tool Integration and Automation

All tools support automation through:

- **TCL Scripts:** Primary scripting language for most EDA tools
- **Makefiles:** Build process automation
- **Shell Scripts:** Tool invocation and file management

2.5 System Requirements

Component	Specification
Operating System	Ubuntu 20.04 LTS or newer (or WSL2 on Windows)
RAM	8GB minimum, 16GB recommended
Storage	30GB free disk space
Processor	Multi-core CPU recommended for faster builds

Table 4: Minimum system requirements

3 Environment and Tools Setup

3.1 Overview

This section introduces the initial setup and fundamental Unix commands necessary before starting the RTL-to-GDSII design flow. Understanding basic shell operations is essential for handling design files, toolchains, and automation scripts throughout the open-source VLSI design environment.

3.2 Setting up a Unix Environment

For VLSI design, a Unix-based system is preferred. Windows users can enable a compatible environment using Windows Subsystem for Linux (WSL):

```
wsl --install
```

After installation, reboot the system, create a username and password, and open the Ubuntu terminal to access a Linux shell environment.

3.3 Basic Unix Commands

Below is a list of essential Unix commands and their purposes in the design workflow:

- **ls** – List all files and directories in the current working directory.
- **cd** – Change directories within the file system.
- **pwd** – Print the current working directory path.
- **mkdir** – Create new directories for organizing design projects.
- **mv** – Move or rename files and directories.
- **cp** – Copy files and directories.
- **touch** – Create an empty file (useful for starting scripts or testbenches).
- **rm** – Remove files or directories.
- **cat** – Display file contents directly in the terminal.
- **which** – Show the path of an executable command.
- **man** – Access the manual page for a command to view usage details.
- **sudo** – Execute commands with superuser privileges (e.g., system updates).
- **du** – Check disk usage by files or directories.
- **df** – Display available disk space in the system.
- **ps** – Show active processes currently running.
- **top** – Display a dynamic real-time system resource monitor.
- **bg** – Move a stopped process to the background.
- **fg** – Bring a background process to the foreground.
- **history** – Display a list of recently executed commands.
- **whoami** – Display the username of the current user.

3.4 Practical Application in VLSI Flow

These commands are foundational for:

- Navigating directories containing RTL, synthesis, and layout files.
- Managing input/output netlists and intermediate reports.
- Running synthesis and simulation scripts efficiently.
- Monitoring system performance during computationally heavy tasks such as place-and-route.

3.5 Exercise

1. Set up WSL or a Linux environment on your workstation.
2. Practice creating directories for each VLSI design stage (e.g., rtl/, synth/, pnr/, gds/).
3. Use the above Unix commands to create, move, and view files within your flow.

3.6 Summary

A basic understanding of Unix commands is vital for working with open-source tools in the RTL-to-GDSII flow. These skills will streamline automation, debugging, and design management in later stages of this workshop.

4 Scripting with TCL

4.1 Overview

This module introduces TCL (Tool Command Language), a powerful scripting language extensively used in Electronic Design Automation (EDA) tools and design flows. TCL enables automation, tool customization, and integration between multiple tools in the RTL-to-GDSII process.

4.2 Purpose in VLSI Design

TCL scripting allows designers to:

- Automate repetitive commands in synthesis, place-and-route, and verification.
- Interface multiple EDA tools in a unified flow.
- Create parameterized design scripts for better reproducibility.
- Modify and control tool environments dynamically.

4.3 Basic TCL Commands and Syntax

TCL syntax is similar to many programming languages and includes constructs such as loops, conditionals, lists, and procedures.

Example 1: Basic List Operations

```tcl
set List {0 1 2 3 4 5 6}
set index -1
foreach elem $List {
  incr index
  puts "Index: $index"
  if {$elem % 2 == 0} {
    lset List $index [expr {-$elem}]
  }
  puts "Updated list: $List"
}
```

Explanation:

- **set** – Defines a variable and assigns a value.
- **foreach** – Iterates over each element in a list.
- **incr** – Increments a numeric variable.
- **puts** – Prints to the terminal.
- **if** – Conditional check similar to other languages.
- **lset** – Updates an element of a list at a given index.
- **expr** – Evaluates an arithmetic expression.

Example 2: File Handling Commands

```
set fp [open "input.txt" w+]
puts $fp "test"
close $fp

set fp [open "input.txt" r]
set file_data [read $fp]
puts $file_data
close $fp
```

Explanation:

- **open** – Opens a file channel in a specific mode (read, write, append, etc.).
- **close** – Closes the file channel.
- **puts** – Writes data to a file or displays output.
- **read** – Reads file contents into a variable.

Example 3: Procedures and Return Statements

```
proc printSumProduct {x y} {
  set sum [expr {$x + $y}]
  set prod [expr {$x * $y}]
  puts "Sum is : $sum"
  puts "Product is : $prod"
  return
  puts "This line will not be printed"
}

puts [printSumProduct 10 50]
```

Explanation:

- **proc** – Defines a reusable procedure or function.
- **return** – Terminates execution and returns control to the caller.

Example 4: Executing System Commands

```
puts [exec ls]
puts [exec pwd]
```

Explanation:

- **exec** – Executes external system commands within a TCL script.
- In this example, the script lists directory contents and prints the current working directory.

4.4 Running TCL Scripts

TCL scripts are saved with a
texttt.tcl extension and executed using:

```
tclsh script_name.tcl
```

4.5 Practical Usage in VLSI Flow

TCL is widely embedded in open-source and commercial tools such as Yosys, OpenROAD, Magic, and Synopsys tools. It is used to:

- Configure design environments.
- Execute synthesis and timing analysis scripts.
- Generate reports automatically.
- Manage flow automation pipelines.

4.6 Summary

TCL scripting is a cornerstone of modern EDA automation. A clear understanding of its commands and structure is essential for customizing design flows and improving productivity in RTL-to-GDSII projects.

5 High-Level Synthesis

5.1 Overview

This module introduces High-Level Synthesis (HLS) and demonstrates using an open-source HLS tool (Bambu) to convert C/C++ behavioral descriptions into RTL (Verilog). HLS speeds up design exploration and lets you prototype hardware accelerators or compute kernels before hand-coding RTL.

5.2 Why use HLS

- Faster design space exploration — change algorithmic code and re-generate RTL quickly.
- Easier expression of complex control and dataflow in C/C++/SystemC.
- Good for generating hardware accelerators or data-path heavy modules.

5.3 Installing Bambu (recommended on Ubuntu / WSL)

A minimal sequence to prepare an Ubuntu environment and run the Bambu AppImage:

```
# Update and install basic dependencies
sudo apt-get update
sudo apt-get install -y build-essential gcc-multilib git \
    iverilog verilator wget

# Download the Bambu AppImage
# (check bambu release page for latest version)
wget https://release.bambuhls.eu/appimage/bambu-0.9.7.AppImage
chmod +x bambu-0.9.7.AppImage

# If AppImage needs FUSE on your platform:
sudo add-apt-repository universe
sudo apt-get install -y libfuse2
```

Verify installation by running:

```
./bambu-0.9.7.AppImage --help
```

5.4 Basic usage example: GCD Algorithm

Throughout this workshop, we use the **Greatest Common Divisor (GCD)** as our golden thread example. This algorithm is ideal because it demonstrates both control flow (loops and conditionals) and datapath operations (subtraction and comparison), making it a realistic hardware design case.

We start with a C implementation of Euclid's algorithm saved in `gcd.c`:

```
// gcd.c
void gcd(int a, int b, int *out) {
    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    *out = a;
}
```

Key Insight: HLS will automatically generate the control logic for the `while` loop, creating a finite state machine (FSM) that:

- Waits for inputs `a` and `b`
- Repeatedly performs subtraction based on comparison
- Exits when `a == b` and writes the result

Run Bambu to synthesize this function (from the directory containing `gcd.c`):

```
./bambu-0.9.7.AppImage gcd.c --top-fname=gcd
```

Outputs typically include:

- Generated Verilog (e.g., `gcd.v`)
- Synthesis reports (latency, resource usage, FSM / datapath breakdown)
- Optional intermediate files (JSON, graphs)

5.5 What to inspect in the generated RTL

- **Control FSM:** Implements the `while` loop and branching logic. Look for state transitions that correspond to the loop condition (`a != b`) and the conditional subtraction.
- **Datapath operators:** Two subtraction operations (`a - b` and `b - a`), comparison units (`a > b`, `a == b`), and registers to hold `a` and `b`.
- **Interface signals:** Top-level wrapper for the synthesized function, including `start`, `done`, clock, and reset signals.
- **Performance metrics:** Reported latency (cycles to compute GCD), area (operators/registers), and memory/resource usage.

Note: The GCD algorithm has *data-dependent latency* — the number of iterations (and thus clock cycles) depends on the input values. HLS reports the worst-case or average latency based on its analysis.

5.6 Verification workflow

1. Simulate the original C code with test vectors (e.g., GCD(48, 18) should return 6).
2. Simulate the generated Verilog using Icarus Verilog or Verilator with the same inputs.
3. Compare outputs to ensure functional equivalence between C and RTL.
4. Integrate the generated RTL into your Yosys/OpenROAD/OpenLane flow for synthesis and physical design.

Workshop Continuity: In the next sections, we will manually write an RTL version of the GCD unit to understand the control/datapath separation, then compare it with the HLS-generated version.

5.7 Practical tips

- Start with small, self-contained functions to learn how constructs map to RTL.
- Prefer static-sized arrays and avoid library calls that Bambu may not support.
- Use Bambu command flags (see its documentation) to tune optimization, latency vs. area, and interface handling.
- Inspect both the generated Verilog and the synthesis reports — the latter often contains valuable optimization hints.
- Automate the HLS→RTL→simulation verification with a small script (shell/tcl) so you can iterate quickly.

5.8 Exercise

1. Install Bambu in your WSL/Ubuntu environment using the steps above.
2. Synthesize the GCD function from `gcd.c` and inspect the generated Verilog.
3. Examine the synthesis report: How many states does the FSM have? How many subtractors are instantiated?
4. Run a Verilator or Icarus Verilog simulation of the generated RTL with test cases:
    - GCD(48, 18) → expected result: 6
    - GCD(100, 25) → expected result: 25
    - GCD(17, 19) → expected result: 1
5. Compare the cycle count for each test case and relate it to the number of loop iterations in the C code.

5.9 References and further reading

- Bambu HLS documentation and release page (check for the latest AppImage and flags).
- Verilator and Icarus Verilog user guides for simulation.
- Example HLS-to-RTL tutorials and papers — useful to understand common HLS idioms and best practices.

6 RTL Design and Simulation

6.1 Overview

Register Transfer Level (RTL) design is the abstraction level where digital circuits are described in terms of data flow between registers and the logical operations performed on signals. This module covers RTL design principles and best practices using Verilog.

6.2 RTL Design Fundamentals

RTL design focuses on describing hardware behavior at a level where:

- Data movement between registers is explicit
- Combinational logic operates on data during clock cycles
- Timing is controlled by clock signals
- Hardware resources are inferred from code structure

6.3 Verilog RTL Design Basics

Module Declaration

Every Verilog design begins with a module declaration:

```verilog
module example_module(
    input wire clk,
    input wire rst_n,
    input wire [7:0] data_in,
    output reg [7:0] data_out,
    output wire valid
);
  // Module contents
endmodule
```

Combinational Logic

Use `assign` statements or `always @(*)` blocks for combinational logic:

```verilog
// Using assign
assign sum = a + b;

// Using always block
always @(*) begin
    if (select)
        mux_out = input_a;
    else
        mux_out = input_b;
end
```

Sequential Logic

Use `always @(posedge clk)` for synchronous sequential logic:

```verilog
always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        counter <= 8'b0;
    else if (enable)
        counter <= counter + 1;
end
```

6.4 Common RTL Design Patterns

GCD Unit: Manual FSM + Datapath Implementation

In the previous section, we used HLS to automatically generate RTL for the GCD algorithm. Now we manually design the same functionality to understand the **separation of control logic (FSM) and datapath**.

This design demonstrates:

- A finite state machine to control the computation flow
- Datapath registers (`a` and `b`) that hold intermediate values
- Conditional subtraction based on comparison results
- Handshake protocol with `start` and `done` signals

```
module gcd (
    input clk, rst_n, start,
    input [15:0] a_in, b_in,
    output reg done,
    output reg [15:0] result
);
    reg [1:0] state, next_state;
    reg [15:0] a, b, next_a, next_b;
    localparam IDLE = 2'b00, CALC = 2'b01, FINISH = 2'b10;

    // State Update (Sequential Logic)
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) state <= IDLE;
        else {state, a, b} <= {next_state, next_a, next_b};
    end

    // Next State & Datapath Logic (Combinational)
    always @(*) begin
        next_state = state;
        next_a = a;
        next_b = b;
        done = 0;

        case (state)
            IDLE: begin
                if (start) begin
                    next_a = a_in;
                    next_b = b_in;
                    next_state = CALC;
                end
            end
            CALC: begin
                if (a == b)
                    next_state = FINISH;
                else if (a > b)
                    next_a = a - b;
                else
```

```
                    next_b = b - a;
                end
                FINISH: begin
                    done = 1;
                    result = a;
                    next_state = IDLE;
                end
            endcase
        end
endmodule
```

Design Breakdown

- **IDLE State:** Waits for `start` signal, then loads inputs `a_in` and `b_in` into registers.
- **CALC State:** Implements Euclid's algorithm — subtracts the smaller value from the larger until they are equal.
- **FINISH State:** Asserts `done` and outputs the GCD result, then returns to IDLE.
- **Datapath:** Registers `a` and `b` store working values; two subtractors (`a - b` and `b - a`) perform computation.

Key Observation: This manual design makes the control/datapath split explicit. Later, during synthesis (Yosys), we will see how resource sharing can optimize the two subtraction operations into a single hardware unit.

Parameterized Counter (Auxiliary Example)

For reference, here's a simple parameterized counter that can be used as a building block in larger designs:

```
module counter #(
    parameter WIDTH = 8
)(
    input wire clk,
    input wire rst_n,
    input wire enable,
    output reg [WIDTH-1:0] count,
    output wire overflow
);
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            count <= {WIDTH{1'b0}};
        else if (enable)
            count <= count + 1;
    end

    assign overflow = (count == {WIDTH{1'b1}});
endmodule
```

6.5 RTL Design Best Practices

1. **Synchronous Reset:** Use synchronous resets unless asynchronous is required
2. **Avoid Latches:** Ensure all paths in combinational blocks are defined

3. **Clock Domain Crossing:** Use proper synchronizers for CDC
4. **Parameterization:** Use parameters for reusable, scalable designs
5. **Naming Conventions:** Use consistent, descriptive signal names
6. **Comments:** Document complex logic and design intent
7. **One Clock Per Module:** Avoid multiple clocks in a single module when possible

6.6 Common RTL Coding Mistakes

- **Incomplete sensitivity lists:** Always use `@(*)` for combinational logic
- **Mixing blocking and non-blocking:** Use `<=` for sequential, `=` for combinational
- **Unintended latches:** Ensure all variables are assigned in all branches
- **Race conditions:** Avoid multiple assignments to the same signal

6.7 RTL Testbench Structure

A proper testbench includes clock generation, reset sequencing, stimulus application, and waveform dumping. Here's a template structure:

```
`timescale 1ns/1ps

module tb_example;
    // Clock and reset
    reg clk;
    reg rst_n;

    // Test signals
    reg [7:0] data_in;
    wire [7:0] data_out;

    // Instantiate DUT (Device Under Test)
    example_module dut (
        .clk(clk),
        .rst_n(rst_n),
        .data_in(data_in),
        .data_out(data_out)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;  // 100MHz clock
    end

    // Test sequence
    initial begin
        // Initialize signals
        rst_n = 0;
        data_in = 8'h00;

        // Release reset
        #20 rst_n = 1;
        // Apply test vectors
        #10 data_in = 8'hAA;
        #10 data_in = 8'h55;

        // End simulation
        #100 $finish;
    end

    // Waveform dump
    initial begin
        $dumpfile("waveform.vcd");
        $dumpvars(0, tb_example);
    end
endmodule
```

Note: In the next section, we'll create a specific testbench for the GCD module to verify its functionality with concrete test cases.

6.8 Exercise

1. Implement the GCD module presented above and save it as `gcd.v`.
2. Analyze the design: How many states are used? What signals control state transitions?
3. Identify the datapath elements: registers, comparators, and subtractors.
4. Create a simple testbench (we'll do this formally in the next section).
5. **Challenge:** Modify the design to use only one subtractor by adding a multiplexer to select operands. How does this affect the FSM?

6.9 Summary

RTL design is the foundation of digital hardware implementation. Mastering RTL coding practices, understanding synthesis implications, and writing comprehensive testbenches are essential skills for successful ASIC and FPGA design.

7 Simulation and Verification

7.1 Overview

This module introduces simulation-based verification using **Icarus Verilog**, an open-source Verilog simulation tool. The tutorial demonstrates how to install and use Icarus Verilog, GTKWave for waveform analysis, and the **Covered** tool for code coverage evaluation.

7.2 Installation Steps

Before starting, ensure that a Linux environment (Ubuntu or WSL) is available.

Installing Icarus Verilog

```
# Clone the source repository
git clone https://github.com/steveicarus/iverilog.git
cd iverilog

# Install dependencies
sudo apt-get update
sudo apt-get install gperf autoconf gcc g++ flex bison make

# Build and install Icarus Verilog
sh autoconf.sh
./configure
make
sudo make install
```

After installation, verify with:

```
iverilog -v
```

Installing GTKWave (Waveform Viewer)

```
sudo apt install gtkwave
```

Launch with:

```
gtkwave
```

Installing Covered (Code Coverage Tool)

```
# Clone the Covered repository
git clone https://github.com/chiphackers/covered.git
cd covered

# Install dependencies
sudo apt update
sudo apt-get install zlib1g-dev tcl8.6 tcl8.6-dev \
    tk8.6 tk8.6-dev doxygen

# Configure, build, and install
./configure
make
sudo make install
```

7.3 Simulation Flow Using Icarus Verilog

To perform simulation-based verification, follow these steps:

1. Create your design module (e.g., `gcd.v`).
2. Write a corresponding testbench (e.g., `tb_gcd.v`).
3. Compile the design and testbench:
   ```
   iverilog -o gcd_sim gcd.v tb_gcd.v
   ```
4. Run the simulation:
   ```
   vvp gcd_sim
   ```
5. The simulation produces a VCD file (e.g., `gcd.vcd`) containing waveform data.
6. Visualize it using GTKWave:
   ```
   gtkwave gcd.vcd
   ```

7.4 Example: GCD Unit Testbench

Continuing with our golden thread example, we now verify the GCD module designed in the previous section.

The design file `gcd.v` (already presented in Section 6):

```
module gcd (
    input clk, rst_n, start,
    input [15:0] a_in, b_in,
    output reg done,
    output reg [15:0] result
);
    reg [1:0] state, next_state;
    reg [15:0] a, b, next_a, next_b;
    localparam IDLE = 2'b00, CALC = 2'b01, FINISH = 2'b10;

    // State Update
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) state <= IDLE;
        else {state, a, b} <= {next_state, next_a, next_b};
    end

    // Next State & Logic
    always @(*) begin
        next_state = state; next_a = a; next_b = b; done = 0;
        case (state)
            IDLE: begin
                if (start) begin
                    next_a = a_in; next_b = b_in; next_state = CALC;
                end
            end
            CALC: begin
                if (a == b) next_state = FINISH;
                else if (a > b) next_a = a - b;
                else next_b = b - a;
            end
```

```
            FINISH: begin
                done = 1; result = a; next_state = IDLE;
            end
        endcase
    end
endmodule
```

Testbench file `tb_gcd.v`:

```
`timescale 1ns/1ps

module tb_gcd;
    reg clk, rst_n, start;
    reg [15:0] a_in, b_in;
    wire done;
    wire [15:0] result;

    // Instantiate the GCD module
    gcd uut (
        .clk(clk),
        .rst_n(rst_n),
        .start(start),
        .a_in(a_in),
        .b_in(b_in),
        .done(done),
        .result(result)
    );

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

    initial begin
        // Initialize signals
        clk = 0;
        rst_n = 0;
        start = 0;

        // Release reset after 20ns
        #20 rst_n = 1;

        // Test Case: GCD of 48 and 18 is 6
        #10 a_in = 48; b_in = 18; start = 1;
        #10 start = 0;
        wait(done);
        $display("GCD(48, 18) = %d (expected 6)", result);

        // End simulation
        #50 $finish;
    end

    // Waveform dump for GTKWave
    initial begin
        $dumpfile("gcd.vcd");
        $dumpvars(0, tb_gcd);
    end
endmodule
```

Understanding the Testbench

- **Clock:** Generated with `always #5 clk = ~clk`, creating a 10ns period (100MHz).
- **Reset:** Held low for 20ns, then released to initialize the design.
- **Test Stimulus:** Applies inputs `a_in = 48` and `b_in = 18` with `start = 1` for one cycle.
- **Wait for Completion:** Uses `wait(done)` to pause until the computation finishes.
- **Result Display:** Prints the computed GCD value for verification.
- **VCD Dump:** Records all signal transitions for waveform viewing.

7.5 Viewing Waveforms

Launch GTKWave to inspect signal transitions:

```
gtkwave gcd.vcd
```

The waveform viewer displays:

- **Clock (clk):** Regular clock toggling
- **Reset (rst_n):** Initial reset pulse
- **Start:** One-cycle pulse to initiate computation
- **State:** FSM state transitions (IDLE $\rightarrow$ CALC $\rightarrow$ FINISH)
- **a and b:** Datapath registers showing subtraction iterations
- **Done:** Assertion when computation completes
- **Result:** Final GCD output (should be 6 for inputs 48 and 18)

Observe how the values of `a` and `b` change:

- Cycle 1: a=48, b=18
- Cycle 2: a=30, b=18 (48-18)
- Cycle 3: a=12, b=18 (30-18)
- Cycle 4: a=12, b=6 (18-12)
- Cycle 5: a=6, b=6 (12-6)
- Cycle 6: Done asserted, result=6

7.6 Code Coverage Using Covered

Covered evaluates how well the testbench stimulates the design.

Generating Coverage Data

```
covered score -t tb_gcd -v tb_gcd.v -v gcd.v \
    -vcd gcd.vcd -o gcd.cdd
```

Viewing Coverage Report

```
covered report -d v gcd.cdd
```

The report includes:

- **Line coverage:** Which lines of code were executed
- **Toggle coverage:** Which signals changed value
- **FSM coverage:** Which state transitions occurred
- **Branch coverage:** Which conditional branches were taken

Analysis: With only one test case (GCD(48,18)), you may not achieve 100% coverage. To improve:

- Add test cases where `a < b` initially
- Test edge cases: GCD(1,1), GCD(0,N), GCD(N,N)
- Test co-prime numbers: GCD(17,19) = 1

7.7 Exercise

1. Install Icarus Verilog, GTKWave, and Covered (if not already done).
2. Simulate the GCD design with the provided testbench and observe the waveform.
3. Modify the testbench to add more test cases:
   - GCD(100, 25) → expected: 25
   - GCD(17, 19) → expected: 1
   - GCD(64, 64) → expected: 64
4. Generate and interpret coverage reports. What percentage of code is covered?
5. **Challenge:** Add assertions to check that the result is always less than or equal to both inputs.

7.8 Summary

Simulation-based verification ensures design correctness before physical implementation. Using our GCD example, we demonstrated how to:

- Write a comprehensive testbench with stimulus and checking
- Visualize signal behavior using GTKWave
- Measure code coverage to ensure thorough verification

In the next section, we will synthesize this same GCD module using Yosys, transforming it from RTL to a gate-level netlist.

8 Logic Synthesis (Yosys)

8.1 Overview

This module introduces **Yosys Open Synthesis Suite**, an open-source framework for performing RTL synthesis and technology mapping. The tutorial demonstrates installing Yosys, preparing the environment, and synthesizing a Verilog design to a gate-level netlist mapped to a standard cell library.

8.2 Installing Yosys

Yosys runs on Linux-based systems such as Ubuntu or WSL. To install the tool, first ensure all dependencies are available:

```
sudo apt-get install build-essential clang bison flex \
  libreadline-dev gawk tcl-dev libffi-dev git graphviz xdot \
  pkg-config python3 libboost-system-dev libboost-python-dev \
  libboost-filesystem-dev zlib1g-dev
```

Clone the official Yosys repository and compile it:

```
git clone https://github.com/YosysHQ/yosys.git
cd yosys
make
sudo make install
```

After installation, launch Yosys using:

```
yosys
```

8.3 Library Setup

For synthesis mapping, a technology library (`.lib` file) is required. The tutorial uses the **Silvaco Open-Cell 45nm FreePDK** library. This library is distributed by Silvaco and requires registration via their website. After completing the form, a download link for the FreePDK45 library is emailed to the user (valid for three days).

Once downloaded, extract the library and locate the NLDM folder containing the Liberty file (e.g., `FreePDK45_osu_stdcells.lib`).

8.4 Example Design

Continuing with our GCD unit as the golden thread example, we will synthesize the manually written RTL from Section 6.

The example Verilog design (`gcd.v`):

```
module gcd (
    input clk, rst_n, start,
    input [15:0] a_in, b_in,
    output reg done,
    output reg [15:0] result
);
    reg [1:0] state, next_state;
    reg [15:0] a, b, next_a, next_b;
    localparam IDLE = 2'b00, CALC = 2'b01, FINISH = 2'b10;

    // State Update
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) state <= IDLE;
        else {state, a, b} <= {next_state, next_a, next_b};
    end

    // Next State & Logic
    always @(*) begin
        next_state = state; next_a = a; next_b = b; done = 0;
        case (state)
            IDLE: begin
                if (start) begin
                    next_a = a_in; next_b = b_in; next_state = CALC;
                end
            end
            CALC: begin
                if (a == b) next_state = FINISH;
                else if (a > b) next_a = a - b;
                else next_b = b - a;
            end
            FINISH: begin
                done = 1; result = a; next_state = IDLE;
            end
        endcase
    end
endmodule
```

Key Hardware Structures:

- FSM with 3 states (IDLE, CALC, FINISH)
- Two 16-bit subtractors: `a - b` and `b - a`
- Comparators: `a == b` and `a > b`
- Datapath registers: `a`, `b`, and `result`

8.5 Yosys TCL Script for Synthesis

A sample TCL script (`yosys_gcd.tcl`) automates the synthesis process:

```
# Read RTL
read_verilog gcd.v

# Map generic cells to internal library
techmap

# Map flip-flops using the target technology library
dfflibmap -liberty FreePDK45_osu_stdcells.lib

# Logic optimization and technology mapping
abc -liberty FreePDK45_osu_stdcells.lib

# Clean unused wires and cells
clean

# Write out the synthesized netlist
write_verilog gcd_synth.v

# Display statistics
stat
```

8.6 Running Synthesis in Yosys

Start Yosys and run the TCL script:

```
yosys> script yosys_gcd.tcl
```

This executes all synthesis steps sequentially and produces a gate-level Verilog file (`gcd_synth.v`) mapped to the standard cell library.

8.7 Generated Netlist Overview

The synthesized netlist connects the input and output pins through instantiated logic gates and sequential cells from the technology library. The result typically includes:

- **State registers:** Flip-flops for FSM state and datapath registers (`a`, `b`)
- **Subtractors:** Two 16-bit subtraction units (or one shared unit if optimized)
- **Comparators:** Equality and greater-than comparison logic
- **Control logic:** Multiplexers and gates implementing the FSM transitions
- **Wire connections:** Routing between standard cells and I/O ports

Inspect the `stat` output to see cell counts, chip area, and resource utilization.

8.8 Key Yosys Commands Summary

- **read_verilog** — Reads the input Verilog source file.
- **techmap** — Maps high-level RTL to generic logic primitives.
- **dfflibmap** — Maps flip-flops to library-specific sequential cells.
- **abc -liberty** — Performs logic optimization and mapping using the provided Liberty file.
- **clean** — Removes unused cells and wires.
- **write_verilog** — Exports the synthesized gate-level Verilog netlist.

8.9 Exercise

1. Install Yosys using the steps above.
2. Download and extract the Silvaco FreePDK45 library.
3. Synthesize the GCD design using the TCL script.
4. Inspect the generated netlist (`gcd_synth.v`) and identify:
   - How many flip-flops are instantiated?
   - How many subtractor circuits are present?
   - What is the total chip area reported by `stat`?
5. Visualize the synthesized netlist using `show` command or `netlistsvg`.

8.10 Summary

This tutorial introduced RTL synthesis using Yosys and demonstrated how to map a Verilog design to a real technology library. The generated netlist can be further used for placement, routing, and timing verification in subsequent stages of the RTL-to-GDSII design flow.

8.11 Logic Optimization using Yosys

This section extends the previous Yosys synthesis tutorial by exploring **logic optimization techniques**, particularly **resource sharing**. The goal is to understand how Yosys identifies and optimizes redundant hardware resources, thereby minimizing circuit area.

Concept of Resource Sharing

Resource sharing aims to reuse hardware components when multiple operations can be performed using the same resource at different times.

Application to GCD: Our GCD design contains two 16-bit subtractors:

```
else if (a > b)
    next_a = a - b;
else
    next_b = b - a;
```

A naive implementation creates two separate subtraction units. However, since these operations are *mutually exclusive* (only one executes per clock cycle based on the comparison), Yosys can optimize this into:

- **One shared 16-bit subtractor**
- **Input multiplexers** to select operands: (a,b) or (b,a)
- **Output demultiplexer** to route result to either `next_a` or `next_b`

This optimization significantly reduces area by replacing two subtractors (∼2N gates for N-bit) with one subtractor plus small muxes (∼N + constant gates).

Running Yosys Without Optimization

First, synthesize the GCD design without resource sharing:

```
read_verilog gcd.v
proc; opt; techmap
dfflibmap -liberty FreePDK45_osu_stdcells.lib
abc -liberty FreePDK45_osu_stdcells.lib
clean
stat
write_verilog gcd_unopt.v
```

This generates an unoptimized netlist with two subtractor circuits. Note the chip area reported by `stat`.

Running Yosys with Resource Sharing

Next, enable aggressive resource sharing:

```
read_verilog gcd.v
proc; opt; share -aggressive
techmap
dfflibmap -liberty FreePDK45_osu_stdcells.lib
abc -liberty FreePDK45_osu_stdcells.lib
clean
stat
write_verilog gcd_opt.v
```

The command `share -aggressive` enables resource sharing for arithmetic units. The optimized netlist now contains only one subtractor with additional multiplexing logic.

Observations

Compare the two synthesis runs:

- **Cell count:** The optimized design has fewer arithmetic cells
- **Area:** Reduced by approximately 20-30% depending on the library
- **File size:** `gcd_opt.v` is smaller than `gcd_unopt.v`
- **Critical path:** May slightly increase due to added mux delay, but area savings usually dominate

Visualization

Run the following Yosys command to visualize the synthesized netlist:

```
show
```

Compare both netlists to confirm that the optimized design uses a single subtractor with multiplexers on its inputs and outputs.

Design Tradeoffs

Resource sharing demonstrates a classic hardware design tradeoff:

- **Area:** Reduced (fewer arithmetic units)
- **Latency:** Unchanged (operations still take one cycle each)
- **Timing:** Slightly worse (additional mux delay in critical path)
- **Power:** Generally reduced due to smaller area

For the GCD design, resource sharing is almost always beneficial since the algorithm is not timing-critical and area reduction is valuable.

Summary

This optimization step demonstrates Yosys's ability to perform resource sharing and area reduction automatically. By leveraging these optimization passes, designers can significantly improve hardware efficiency before physical design.

The key insight for our GCD example: *mutually exclusive operations in RTL can share hardware resources, and synthesis tools can detect and exploit this automatically with the right optimization flags.*

9 Power Analysis (OpenSTA)

9.1 Overview

This module introduces **power analysis using OpenSTA**, an open-source Static Timing Analysis (STA) tool that also provides capabilities for estimating dynamic and leakage power. Continuing with our golden thread example, we analyze the power consumption of the **GCD unit** synthesized in Section 8.

9.2 Requirements

Before starting, ensure that:

- **OpenSTA** is installed on your system. (Installation details are given in Tutorial 7.)
- You have the following files from previous sections:
  - `gcd_synth.v` – Gate-level netlist from Yosys (Section 8)
  - `gcd.sdc` – Timing constraints file
  - `FreePDK45_osu_stdcells.lib` – Technology library used in synthesis

9.3 Concepts of Power Dissipation

In a CMOS circuit, total power dissipation has two major components:

1. **Dynamic Power Dissipation** – due to switching activity.
2. **Static (Leakage) Power Dissipation** – due to leakage currents in transistors.

Dynamic power can be divided into:

- **Internal Power:** Power dissipated within a cell due to charging/discharging of internal capacitances and short-circuit currents during switching.
- **Switching (External) Power:** Power consumed by charging/discharging external load capacitances driven by the cell.

9.4 Internal and Switching Power Models

Internal power is characterized in the technology library as a Non-Linear Power Model (NLPM). It is typically represented as a two-dimensional lookup table with:

- Input transition time (slew) as one axis.
- Output capacitance (load) as the other axis.

Each entry in this table corresponds to the *energy consumed per transition*, expressed in femtojoules (fJ).

9.5 Files Used in this Tutorial

- **Verilog Netlist:** `gcd_synth.v` – The synthesized GCD design containing FSM logic, datapath registers, subtractors, and comparators.
- **SDC File:** `gcd.sdc` – Defines timing constraints such as clock period, input/output delay, input slew, and output load.
- **Library:** `FreePDK45_osu_stdcells.lib` – Technology library defining power, delay, and leakage parameters for the standard cells.
- **TCL Script:** `gcd_power.tcl` – Script to execute the power analysis commands.

9.6 TCL Script Example

Create a file `gcd_power.tcl` with the following content:

```
# Read technology library
read_lib FreePDK45_osu_stdcells.lib

# Read the synthesized Verilog netlist
read_verilog gcd_synth.v

# Link instances to library cells
link_design gcd

# Read timing constraints
read_sdc gcd.sdc

# Set activity (signal switching probability)
# Higher activity for datapath due to subtractors
set_activity 0.2

# Report power
report_power
```

9.7 Running the Power Analysis

Run OpenSTA using the following commands:

```
sta
source gcd_power.tcl
```

This executes the TCL script, and the tool prints the power report including:

- Internal power (per instance and total)
- Switching power (per net and total)
- Leakage power (per instance and total)
- Total power consumption

9.8 Understanding GCD Power Characteristics

The GCD design consumes more power than a simple inverter due to:

Dynamic Power Contributors:

- **Subtractors:** Two 16-bit subtraction units (or one shared) with high switching activity during CALC state
- **Comparators:** Equality and greater-than comparison logic switching every cycle
- **Datapath Registers:** 16-bit registers for `a` and `b` updating frequently
- **FSM State Registers:** Transitioning between IDLE, CALC, and FINISH states

Leakage Power Contributors:

- More standard cells = higher cumulative leakage
- Arithmetic cells (adders/subtractors) typically have higher leakage than simple logic gates
- All flip-flops contribute to leakage even when idle

9.9 Power Breakdown Example

For a typical GCD implementation with activity factor 0.2 and 100MHz clock:

Internal Power:

- Each subtractor cell: ∼50-100 µW
- Flip-flops (state + datapath): ∼20-40 µW total
- Combinational logic: ∼30-50 µW
- **Total Internal:** ∼100-200 µW

Switching Power:

- Depends heavily on net capacitance and toggle rate
- Datapath nets (16-bit buses) contribute significantly
- **Total Switching:** ∼50-100 µW

Leakage Power:

- FreePDK45 has moderate leakage characteristics
- ∼50-100 cells in the design
- **Total Leakage:** ∼10-20 µW

Total Power: ∼160-320 µW (varies with activity and technology corner)

9.10 Impact of Optimization

If you synthesized with resource sharing (Section 8), compare power:

- **Unoptimized (two subtractors):** Higher dynamic power due to more switching, higher leakage due to more cells
- **Optimized (one subtractor):** Lower dynamic and leakage power, but added mux power
- **Net result:** Optimized version typically saves 15-25% total power

9.11 Interpretation of Power Report

The OpenSTA report will show:

- **Per-instance breakdown:** Identify which modules consume the most power (usually subtractors)
- **Per-net switching:** High-fanout nets (like clock and reset) contribute significantly
- **Hierarchical view:** If your design has hierarchy, see power per sub-module

Look for:

- Unexpectedly high power in specific instances (may indicate over-sized cells)
- High switching power on specific nets (may need buffering or optimization)
- Leakage dominated designs (may benefit from power gating or lower-leakage cells)

9.12 Exercise

1. Run power analysis on both optimized and unoptimized GCD netlists. Compare the results.
2. Modify the activity factor in the TCL script (try 0.1, 0.3, 0.5) and observe impact on dynamic power.
3. Change the clock frequency in the SDC file and re-run analysis. Plot power vs. frequency.
4. Identify the top 5 power-consuming instances in the design. What cell types are they?
5. **Challenge:** If you have the HLS-generated netlist, compare its power to your manual RTL version.

9.13 Summary

This tutorial demonstrated power analysis of the GCD unit using OpenSTA. You learned:

- How to analyze synthesized netlists for power consumption
- The breakdown of internal, switching, and leakage power
- Why arithmetic-heavy designs like GCD have higher power than simple logic
- The impact of optimization techniques on power consumption

Understanding power consumption early in the design flow helps make informed decisions about optimization, clock frequency, and target technology before investing in physical implementation.

10 Installing and Preparing OpenROAD

10.1 Overview

This section explains the installation and setup of **OpenROAD**, an open-source integrated physical design tool. OpenROAD automates chip layout tasks — including floorplanning, placement, clock tree synthesis, and routing — transforming a synthesized Verilog netlist into a final routed layout.

10.2 About OpenROAD

OpenROAD (Open Rapid Object-oriented Automated Design) provides a complete RTL-to-GDSII physical design flow, handling:

- Floorplanning and Power Planning
- Placement and Clock Tree Synthesis (CTS)
- Global and Detailed Routing
- Parasitic Extraction and Reporting

10.3 Installation Steps

Execute these commands in a Linux or WSL terminal:

```
# Clone repository
git clone --recursive \
    https://github.com/The-OpenROAD-Project/OpenROAD.git
cd OpenROAD

# Install dependencies
sudo ./etc/DependencyInstaller.sh

# Build OpenROAD
mkdir build
cd build
cmake ..
make -j$(nproc)
sudo make install
```

Installation may take about 1–2 hours. After completion, run:

```
openroad
```

10.4 Included Libraries and Examples

The OpenROAD repository includes preconfigured libraries:

- **Nangate45**
- **ASAP7**
- **SkyWater130**

Each library folder provides example Verilog files, constraint files, and TCL scripts for step-by-step flow demonstrations.

10.5 Next Steps

Once installed, users can explore sample scripts in:

```
OpenROAD/test/
```

These include examples such as `gcd_nangate45.tcl`, which will be used for chip planning and placement in the following module.

11 Physical Design using OpenROAD

11.1 Overview

This section covers the complete physical design process using OpenROAD, from floorplanning to routing. We continue with our **GCD unit golden thread example**, taking the same design we:

- Generated using HLS (Section 5)
- Manually designed at RTL (Section 6)
- Verified through simulation (Section 7)
- Synthesized to gates with Yosys (Section 8)

We now perform **place and route** to create a physical layout that can be fabricated. The example uses the `gcd_nangate45.tcl` script with the Nangate45 library.

Key Continuity: This is the same GCD algorithm computing greatest common divisor — we're now translating it from abstract gates to concrete silicon geometry.

11.2 Inputs Required

- Synthesized Verilog netlist (`gcd.v` or `gcd_synth.v` from Yosys)
- Constraint file (`gcd_nangate45.sdc`) specifying timing requirements
- Technology files:
  - `Nangate45_typ.lib` (timing library)
  - `Nangate45_stdcell.lef` (standard cell layout)
  - `Nangate45_tech.lef` (routing rules and metal stack)
- Flow TCL script: `gcd_nangate45.tcl`

Important Configuration Step:

- If using the default `gcd_nangate45.tcl` from OpenROAD examples, you must **edit the script** to point to YOUR synthesized netlist from Section 8.
- Locate the line: `read_verilog <path>/gcd.v`
- Change it to: `read_verilog ./gcd_synth.v` (or the full path to your Yosys output)
- Similarly, ensure the SDC file path points to your constraints file.

Note: The GCD netlist contains the FSM states, datapath registers (a, b, result), subtractor logic, and comparators we designed earlier. OpenROAD will place these gates and route the connections between them.

11.3 1. Floorplanning and Power Planning

- Define die/core area using bottom-left (LX, LY) and top-right (UX, UY) coordinates.
- Generate site rows for cell placement.
- Place IO pins randomly or by constraint.
- Add tap and end-cap cells to prevent latch-up and protect boundaries.

- Generate the Power Distribution Network (PDN) using:

  ```
  source flow_pdn.tcl
  ```

  This builds VDD and VSS grids across metal layers.

11.4 2. Placement

- Perform global placement with congestion estimation.
- Set signal and clock routing layers.
- Repair violations (slew, capacitance, fanout) and resize gates.
- Execute detailed placement to legalize cell positions.

The resulting DEF shows correctly aligned cells within placement rows. You can identify:

- **State register flip-flops** for the FSM
- **Datapath registers** holding `a`, `b`, and `result`
- **Arithmetic logic** for the subtractors and comparators
- **Control multiplexers** selecting between operations

11.5 3. Clock Tree Synthesis (CTS)

- Clone and repair clock inverters.
- Build the clock tree with buffer insertion and clustering:

  ```
  clock_tree_synthesis -root clk -buf_list {BUF_X2 BUF_X4}
  ```

- Re-run detailed placement for clock buffers.
- Repair setup and hold violations after CTS.

11.6 4. Routing

Global Routing:

- Generates route guides using global congestion analysis.
- Identifies congestion and antenna issues.

Detailed Routing:

- Uses multiple threads for performance.
- Creates metal traces respecting DRC rules.
- Inserts filler cells and checks antenna violations.
- Outputs final `.def`, `.gds`, and `.spef` (parasitics) files.

11.7 5. Parasitic Extraction and Reporting

After routing, extract RC data:

```
extract_parasitics -rc_file nangate45.rc
report_timing
report_power
```

Generates timing and power summaries, worst slack reports, and congestion maps.

11.8 Visualization and Checks

- Use the GUI command:

  ```
  openroad -gui -log gcd.log -script gcd_nangate45.tcl
  ```

- Explore:
  - Placement density (heatmap)
  - Clock tree topology
  - Routing congestion and power grid

11.9 Summary

This module completes the physical design phase in the RTL-to-GDSII flow. You learned how OpenROAD automates floorplanning, placement, CTS, and routing, producing a verified and routed layout ready for DRC/LVS and GDS export.

Workshop Golden Thread Complete: We have now taken the GCD algorithm from:

1. High-level C code (HLS synthesis)
2. Manual RTL design (FSM + Datapath)
3. Functional verification (Simulation)
4. Gate-level netlist (Logic synthesis with optimization)
5. Physical layout (Place and route)

This unified example demonstrates the complete RTL-to-GDSII flow using a single, coherent design throughout all stages.

12 DRC / LVS / GDSII Verification

12.1 Overview

Physical verification ensures that the fabricated chip will function correctly and meet manufacturing requirements. This module covers Design Rule Checking (DRC), Layout vs. Schematic (LVS) verification, and GDSII generation using open-source tools.

12.2 Design Rule Checking (DRC)

DRC verifies that the layout adheres to the foundry's manufacturing rules:

- Minimum width and spacing of metal layers
- Via dimensions and enclosure rules
- Well and substrate tap requirements
- Antenna rules for plasma damage prevention
- Density requirements for chemical-mechanical polishing (CMP)

12.3 Installing Magic VLSI

Magic is an open-source layout editor with integrated DRC and extraction capabilities:

```
# Install dependencies
sudo apt-get update
sudo apt-get install -y m4 tcl-dev tk-dev libcairo2-dev \
    mesa-common-dev libglu1-mesa-dev

# Clone and build Magic
git clone https://github.com/RTimothyEdwards/magic.git
cd magic
./configure
make
sudo make install
```

Verify installation:

```
magic --version
```

12.4 Running DRC with Magic

Load a GDSII or DEF file and run DRC:

```
magic -d XR -noconsole -dnull design.gds
```

In the Magic TCL console:

```
# Load technology file
tech load sky130A

# Load design
gds read design.gds

# Run DRC
drc check
drc why
```

```
# Generate DRC report
drc catchup
drc listall why > drc_report.txt
```

12.5 Layout vs. Schematic (LVS)

LVS verification confirms that the physical layout matches the intended circuit netlist.

Installing Netgen

Netgen performs circuit-level netlist comparison:

```
# Clone and build Netgen
git clone https://github.com/RTimothyEdwards/netgen.git
cd netgen
./configure
make
sudo make install
```

Running LVS

Compare layout-extracted netlist with schematic netlist:

```
netgen -batch lvs "layout.spice design" \
    "schematic.spice design" \
    sky130A_setup.tcl lvs_report.txt
```

Example LVS TCL script:

```
# Load technology setup
source sky130A_setup.tcl

# Read layout netlist
readnet spice layout.spice

# Read schematic netlist
readnet spice schematic.spice

# Run LVS comparison
lvs "layout.spice design" "schematic.spice design" \
    sky130A_setup.tcl lvs_output.txt

# Check results
puts "LVS Complete"
quit
```

12.6 GDSII Generation

GDSII is the industry-standard format for IC layout data.

Exporting GDSII from Magic

```
# In Magic TCL console
gds write design.gds
```

Converting DEF to GDSII

If starting from DEF format (from OpenROAD):

```
magic -T sky130A.tech -dnull -noconsole << EOF
def read design.def
gds write design.gds
quit
EOF
```

12.7 GDSII Verification Steps

1. **Layer Verification:** Ensure all required layers are present
2. **Hierarchy Check:** Verify cell hierarchy is correct
3. **Boundary Check:** Confirm chip boundaries are defined
4. **Text Labels:** Verify port and net labels
5. **File Integrity:** Check GDSII file is not corrupted

12.8 Viewing GDSII Files

Use KLayout for GDSII visualization:

```
# Install KLayout
sudo apt-get install klayout

# View GDSII
klayout design.gds
```

12.9 Parasitic Extraction

Extract resistance and capacitance from layout:

```
# In Magic console
extract all
ext2spice lvs
ext2spice cthresh 0.01
ext2spice rthresh 0.01
ext2spice
```

This generates a SPICE netlist with parasitics for post-layout simulation.

12.10 Common DRC Violations

- **Minimum spacing:** Metal tracks too close together
- **Minimum width:** Wires thinner than allowed
- **Enclosure:** Via not properly enclosed by metal
- **Density:** Metal density outside acceptable range
- **Antenna:** Long routing causing plasma damage risk

12.11 Common LVS Errors

- **Missing nets:** Layout missing connections from schematic
- **Device mismatch:** Different number of transistors
- **Swapped pins:** Incorrectly connected terminals
- **Shorted nets:** Unintended connections in layout
- **Size mismatch:** Device dimensions differ from schematic

12.12 Sign-off Checklist

Before tape-out, verify:

1. All DRC violations resolved
2. LVS comparison passes cleanly
3. Timing closure achieved (setup and hold)
4. Power analysis shows acceptable consumption
5. Electromigration rules satisfied
6. IR drop within specifications
7. GDSII file format validated
8. Layer map confirmed with foundry

12.13 Exercise

1. Load a sample design in Magic and run DRC
2. Fix common DRC violations
3. Extract netlist from layout
4. Perform LVS comparison
5. Generate final GDSII file
6. View GDSII in KLayout and verify layers

12.14 Summary

Physical verification is the final gatekeeper before fabrication. DRC ensures manufacturability, LVS confirms functional correctness, and proper GDSII generation provides the foundry with accurate layout data. Mastering these verification steps is critical for successful chip tape-out.

A Useful Commands and References

A.1 Unix/Linux Command Reference

File and Directory Operations

```
# List files with details
ls -lah

# Change directory
cd /path/to/directory
cd ..                    # Go up one level
cd ~                     # Go to home directory

# Create directory
mkdir -p path/to/directory

# Copy files
cp source.v destination.v
cp -r source_dir/ dest_dir/

# Move/rename files
mv old_name.v new_name.v

# Remove files/directories
rm file.v
rm -rf directory/

# Find files
find . -name "*.v"
find . -type f -mtime -7   # Files modified in last 7 days

# Search in files
grep "module" *.v
grep -r "always" design/
```

System Information

```
# Disk usage
df -h
du -sh directory/

# Memory usage
free -h

# Process monitoring
top
htop
ps aux | grep yosys

# System information
uname -a
lsb_release -a
```

A.2 Git Commands for Version Control

```
# Initialize repository
git init

# Clone repository
git clone https://github.com/user/repo.git

# Check status
git status

# Add files
git add design.v
git add .

# Commit changes
git commit -m "Description of changes"

# Push to remote
git push origin main

# Pull updates
git pull

# Create branch
git checkout -b feature-branch

# View commit history
git log --oneline
```

A.3 Icarus Verilog Commands

```
# Compile Verilog files
iverilog -o simulation design.v testbench.v

# Run simulation
vvp simulation

# With specific timescale
iverilog -g2005-sv -o sim design.sv testbench.sv

# Generate VCD waveform
vvp simulation -vcd

# View waveform
gtkwave waveform.vcd
```

A.4 Yosys Synthesis Commands

```
# Read Verilog
read_verilog design.v

# Synthesize
synth -top module_name

# Technology mapping
dfflibmap -liberty library.lib
```

```
abc -liberty library.lib

# Write netlist
write_verilog -noattr netlist.v

# Generate reports
stat
check

# Show schematic
show
```

A.5 OpenSTA Commands

```
# Read liberty file
read_liberty library.lib

# Read netlist
read_verilog netlist.v

# Link design
link_design top_module

# Read SDC constraints
read_sdc constraints.sdc

# Report timing
report_checks -path_delay min_max
report_tns
report_wns

# Report power
report_power
```

A.6 OpenROAD Commands

```
# Read LEF/DEF
read_lef technology.lef
read_def design.def

# Floor planning
initialize_floorplan \
    -die_area "0 0 1000 1000" \
    -core_area "10 10 990 990"

# Placement
global_placement
detailed_placement

# Clock tree synthesis
clock_tree_synthesis -root_buf BUF_X1 \
    -buf_list BUF_X1

# Routing
global_route
detailed_route
```

```
# Write output
write_def output.def
```

A.7 Magic VLSI Commands

```
# Load technology
tech load sky130A

# Read GDS
gds read design.gds

# Read DEF
def read design.def

# Run DRC
drc check
drc why

# Extract netlist
extract all
ext2spice lvs
ext2spice

# Write GDS
gds write output.gds
```

A.8 TCL Scripting Quick Reference

```
# Variables
set var_name value
puts $var_name

# Lists
set my_list {item1 item2 item3}
lindex $my_list 0
llength $my_list
lappend my_list item4

# Control structures
if {$a > $b} {
    puts "a is greater"
} else {
    puts "b is greater"
}

foreach item $my_list {
    puts $item
}

for {set i 0} {$i < 10} {incr i} {
    puts $i
}

# Procedures
proc my_proc {arg1 arg2} {
    return [expr {$arg1 + $arg2}]
}

# File operations
set fp [open "file.txt" r]
set data [read $fp]
close $fp
```

A.9 Makefile Template

```
# Makefile for RTL-to-GDSII flow

DESIGN = top_module
VERILOG_FILES = design.v
TB_FILES = testbench.v
LIB = library.lib

# Simulation
sim:
  iverilog -o $(DESIGN).vvp $(VERILOG_FILES) $(TB_FILES)
  vvp $(DESIGN).vvp
  gtkwave waveform.vcd

# Synthesis
synth:
  yosys -p "read_verilog $(VERILOG_FILES); \
           synth -top $(DESIGN); \
           dfflibmap -liberty $(LIB); \
           abc -liberty $(LIB); \
           write_verilog netlist.v"

# Clean
clean:
  rm -f *.vvp *.vcd netlist.v *.log

.PHONY: sim synth clean
```

A.10 Online Resources

Tool Documentation

- **Yosys:** https://yosyshq.net/yosys/
- **OpenROAD:** https://openroad.readthedocs.io/
- **Icarus Verilog:** http://iverilog.icarus.com/
- **Magic:** http://opencircuitdesign.com/magic/
- **SkyWater PDK:** https://skywater-pdk.readthedocs.io/

Learning Resources

- **Verilog Tutorial:** https://www.asic-world.com/verilog/
- **Digital Design:** https://www.nandland.com/

- **VLSI Resources:** `https://vlsi.pro/`
- **OpenLane:** `https://openlane.readthedocs.io/`

Community Forums

- **Reddit r/FPGA:** `https://reddit.com/r/FPGA`
- **Slack (SkyWater):** `https://skywater-pdk.slack.com/`
- **Gitter (OpenROAD):** Open-source EDA discussions

A.11 Common File Extensions

- `.v` – Verilog source file
- `.sv` – SystemVerilog file
- `.vcd` – Value Change Dump (waveform)
- `.lib` – Liberty timing library
- `.lef` – Library Exchange Format (physical cells)
- `.def` – Design Exchange Format (placement data)
- `.sdc` – Synopsys Design Constraints (timing)
- `.spice` – SPICE netlist
- `.gds` – GDSII layout file
- `.mag` – Magic layout file
- `.tcl` – TCL script

A.12 Troubleshooting Tips

Simulation Issues

- Check for syntax errors in Verilog
- Ensure testbench generates clock properly
- Verify $dumpfile and $dumpvars are called
- Use $display for debugging

Synthesis Issues

- Ensure Liberty file path is correct
- Check for unmapped cells in netlist
- Verify module names match between files
- Review synthesis warnings carefully

Physical Design Issues

- Verify LEF/DEF compatibility
- Check floorplan dimensions
- Ensure adequate power/ground routing
- Review routing congestion reports

A.13 Performance Optimization Tips

- Use incremental compilation when possible
- Enable multi-threading in tools that support it
- Optimize TCL scripts to reduce redundant operations
- Use appropriate optimization flags in synthesis
- Monitor system resources (RAM, disk I/O)