BECE407P ASIC DESIGN

Module-3: RTL Synthesis Flow

Dr. PRITAM BHATTACHARJEE

Assistant Professor (Senior Grade 2)

School of Electronics Engineering (SENSE)

Vellore Institute of Technology – Chennai

pritam.bhattacharjee@vit.ac.in, +91 8132863424

Contents

What is RTL Synthesis?

- Converts Register Transfer Level (RTL) description into gate-level netlist.
- Bridge between abstract hardware design and physical implementation.
- Used for both ASIC and FPGA design flows.

Overview of RTL Synthesis

- **HDL Parsing and Elaboration:** Reads Verilog/VHDL, builds internal representation.
- **Optimization:** Area, speed, and/or power are improved.
- **Technology Mapping:** Maps optimized logic to gates in target library.
- **Gate-level Netlist Generation:** Produces netlist for layout/back-end tools.
- **Formal Equivalence Checking (FEC):** Ensures functionality matches RTL

HDL Parsing and Elaboration in Verilog

- **Parsing** is the initial stage where the Verilog tool reads the HDL code, checks for syntax errors, and translates it into an intermediate data structure for further processing.
- **Elaboration** follows parsing and is the process where the tool resolves module hierarchies, parameter values, generates instances, and constructs the complete circuit model in memory. It finalizes all connections, computes bit widths, and binds modules to their instances—the complete hardware representation is built at this stage, before synthesis or simulation.

HDL Parsing and Elaboration in Verilog

1. **Parsing**
   - Reads each Verilog file and analyses statements, checking syntax.
   - Constructs an *abstract syntax tree* (AST) or data structure representing the HDL code.
   - Example error: If the syntax is incorrect, the parser throws an error (e.g., missing semicolon or incorrect port declaration).
2. **Elaboration**
   - Processes parameterized modules, expands generate blocks, and instantiates all objects and their connections.
   - Binds configurations to module instances.
   - Resolves hierarchy: All submodules are linked to their definitions.
   - Computes sizes for arrays, buses, and wires after parameter replacement.
   - After elaboration, the circuit model is complete and ready for simulation/synthesis.

HDL Parsing and Elaboration in Verilog

Simple Parameterized Module

```
module adder #(parameter WIDTH =
8) (
    input [WIDTH-1:0] a, b,
    output [WIDTH-1:0] sum
);
    assign sum = a + b;
endmodule
```

Top-Level Module Instantiates the Parameterized Module

```
module top;
    reg [15:0] x, y;
    wire [15:0] z;

    // Instance of adder with
WIDTH = 16
    adder #(16) my_adder (.a(x),
.b(y), .sum(z));
endmodule
```

How Parsing and Elaboration Work?

- **Parsing:** Both modules are read. The parser checks the syntax of declarations, parameter use, and connections.
- **Elaboration:**
  - The top module contains an instance `my_adder` of adder, with parameter `WIDTH=16`.
  - The elaborator binds `my_adder` to the adder definition and replaces WIDTH with 16, updating port sizes to `[15:0]`.
  - Hierarchy is resolved: any connections in top are mapped to `my_adder`.
  - The circuit is now ready for simulation or synthesis with correct bit-widths, connections, and instances.

HDL Parsing and Elaboration in Verilog

Example with `generate` Block

```
module mult_adder #(parameter N = 4) (
    input [N-1:0] a, b,
    output [N-1:0] sum
);
    genvar i;
    generate
        for (i=0; i<N; i=i+1) begin : adder_gen
            assign sum[i] = a[i] + b[i];
        end
    endgenerate
endmodule
```

During elaboration, the tool expands the loop, generating N assignments for the given value of N.

Key Points

- Elaboration occurs after parsing, but before synthesis or simulation.
- It binds module instances, resolves parameters, expands generate blocks, and builds the full design hierarchy.
- Without correct elaboration, the design cannot be synthesized or simulated properly.

Schematic Illustration

After elaboration, the internal representation (shown in EDA tools) is a hierarchical tree where all modules, signals, and parameters are resolved and connections finalized.

In summary:

- **Parsing:** Reads and validates the Verilog code's syntax.
- **Elaboration:** Builds the design hierarchy, applies parameters, and finalizes all connections, creating the full hardware model for simulation/synthesis.

These steps are essential for the correct behavioural and structural representation of a Verilog design in tools.

Optimization

Optimization during RTL synthesis in Verilog aims to improve area, speed, and power of the resulting hardware. Optimization techniques can be applied both at the coding level (RTL) and by synthesis tools.

Optimization

Optimization during RTL synthesis in Verilog aims to improve area, speed, and power of the resulting hardware. Optimization techniques can be applied both at the coding level (RTL) and by synthesis tools.

1. Area Optimization

- **Resource Sharing:** Use shared functional blocks to reduce hardware.
- **Remove Redundancies:** Avoid duplicate logic and unused signals.
- **Use Efficient Encoding:** For state machines, use one-hot, binary, or gray encoding as appropriate.

Example: Shared Adder instead of Duplicates

```
// Inefficient: Two separate adders
assign sum1 = a1 + b1;
assign sum2 = a2 + b2;

// Efficient: One shared adder, if possible (multiplexing operands)
wire [7:0] add_a, add_b, add_out;
assign add_a = sel ? a1 : a2;
assign add_b = sel ? b1 : b2;
assign add_out = add_a + add_b;
```

This technique reduces area by sharing computation blocks when operations are mutually exclusive.

Optimization

Optimization during RTL synthesis in Verilog aims to improve area, speed, and power of the resulting hardware. Optimization techniques can be applied both at the coding level (RTL) and by synthesis tools.

2. Speed (Timing) Optimization

- **Pipeline Registers:** Insert registers to break long combinational logic paths (increase clock frequency).
- **Balance Logic Depth:** Match delay across paths by splitting long combinational chains.
- **Retiming:** Synthesis tools can move registers forward/backward for optimal timing.

Example: Simple Pipelining

```
// Single-cycle combinational
assign result = (a * b) + c;

// Two-stage pipeline for speed
reg [15:0] mult_reg;
always @(posedge clk) begin
  mult_reg <= a * b;
end
assign result = mult_reg + c;
```

By pipelining, you make the circuit faster and able to run at a higher clock frequency.

Optimization

Optimization during RTL synthesis in Verilog aims to improve area, speed, and power of the resulting hardware. Optimization techniques can be applied both at the coding level (RTL) and by synthesis tools.

3. Power Optimization

- **Clock Gating:** Disable clocks to unused modules.
- **Optimize Signal Toggles:** Avoid constantly toggling signals that aren't needed.
- **Use Low-Power Structures:** Such as multi-threshold CMOS, data gating, and minimize toggling on buses.

Example: Clock Gating

```
reg clk_enable; // When 0, disables clock
always @(posedge clk) begin
  if (clk_enable)
    q <= d;
end
```

Here, clock enable logic turns off activity, reducing dynamic power consumption when possible.

Optimization

Optimization during RTL synthesis in Verilog aims to improve area, speed, and power of the resulting hardware. Optimization techniques can be applied both at the coding level (RTL) and by synthesis tools.

4. Synthesis Directives & Attributes

Synthesis tools allow attributes/pragmas for direct optimization hints:

- **keep/preserve:** Prevents removal/minimization.
- **maxfan:** Limits the fanout for timing purposes.
- **ramstyle/multstyle:** Instructs mapping to certain hardware resources.

Example: Using Directives

```
(* preserve *) reg my_reg;
reg data /* synthesis maxfan = 8 preserve */;
```

These hints are respected by certain synthesis tools to optimize area, speed, or power as required.

Optimization

Optimization during RTL synthesis in Verilog aims to improve area, speed, and power of the resulting hardware. Optimization techniques can be applied both at the coding level (RTL) and by synthesis tools.

5. Constant Propagation and Logic Folding

Synthesis tools can propagate constants and fold logic to minimize gate count and improve speed.

Example: Constant Propagation

```
assign out = enable ? data : 0;
```

If enable is always '1', the synthesis removes the multiplexing logic—they optimize based on RTL code and constraints.

Optimization

Optimization during RTL synthesis in Verilog aims to improve area, speed, and power of the resulting hardware. Optimization techniques can be applied both at the coding level (RTL) and by synthesis tools.

5. Constant Propagation and Logic Folding

Synthesis tools can propagate constants and fold logic to minimize gate count and improve speed.

Example: Constant Propagation

```
assign out = enable ? data : 0;
```

If enable is always '1', the synthesis removes the multiplexing logic—they optimize based on RTL code and constraints.

Recommendations

- Write concise, intent-driven RTL code.
- Use parameters for configurability and efficient reuse.
- Always analyse synthesis reports for timing, area, and power, iterating design accordingly.
- Apply clock gating and pipeline registers for large/fast circuits.
- Use synthesis directives for tool-guided optimization.

Optimization is a collaborative process between RTL coding style and synthesis tool capabilities. Good coding enables the tools to further optimize for area, speed, and power according to technology constraints and project goals.

Technology Mapping in Verilog RTL Synthesis

Technology mapping is the step in digital logic synthesis where an optimized abstract logic representation (post-optimization and generic mapping) is translated to a netlist composed of gates available in the target technology library, such as NAND, NOR, and Flip-Flops of specific drive strengths and delays. This critical phase ensures that the synthesized circuit can be physically implemented using the selected standard cells, maximizing performance and efficiency for area, speed, and power.

Technology Mapping in Verilog RTL Synthesis

Technology mapping is the step in digital logic synthesis where an optimized abstract logic representation (post-optimization and generic mapping) is translated to a netlist composed of gates available in the target technology library, such as NAND, NOR, and Flip-Flops of specific drive strengths and delays. This critical phase ensures that the synthesized circuit can be physically implemented using the selected standard cells, maximizing performance and efficiency for area, speed, and power.

Technology Mapping – Process Overview

1. **Generic Logic Netlist:** After logic optimization, your design consists of generic logic gates and flip-flops.
2. **Library Binding:** The generic gates are mapped to their closest counterparts in the technology library. For instance, an AND gate in your design is replaced by an AND2X1 cell from the library.
3. **Optimization:** The mapped netlist is further optimized for area, speed, and power using characteristics of the selected cells (gate drive strength, delay, etc.).

Technology Mapping in Verilog RTL Synthesis

Example: Technology Mapping Flow

Consider a simple RTL module:

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

Step 1: RTL Optimization Output

- The synthesis tool determines the needed logic: one AND function.

Step 2: Mapping to Technology Library

- If the target library contains a cell: AND2×1 (2-input AND, 1× strength), the synthesis tool replaces the abstract AND with this cell.

Mapped gate-level representation:

```
module simple_logic(input a, input b, output y);
    AND2×1 u_and (.A(a), .B(b), .Y(y));
endmodule
```

Here, AND2×1 is a technology-specific cell defined in the vendor's library, characterized by its drive, delay, and physical properties.

Technology Mapping in Verilog RTL Synthesis

Practical Example with Multiple Cells

Suppose your RTL describes the following:

```
module example (input a, input b, input c, output y);
    assign y = (a & b) | c;
endmodule
```

- Generic Netlist: 1 AND gate, 1 OR gate.
- Technology Mapping:
  - Map AND to AND2×1
  - Map OR to OR2×1
- Resulting Netlist (using gate-level instantiation):

```
AND2×1 u_and (.A(a), .B(b), .Y(w));
OR2×1  u_or  (.A(w), .B(c), .Y(y));
```

AND2×1 and OR2×1 are technology-specific cells provided by the foundry. Their characteristics (area, power, timing) are known to the synthesis tool, which can select different drive strengths, implement multi-level gates, or merge structures for optimization.

Technology Mapping in Verilog RTL Synthesis

Technology Mapping – Key Considerations

- The selection is **timing-driven**: If the output has tight timing constraints, larger (faster but bigger) drive cells may be used instead of the smallest area cells.
- The process enables **physical implementation**: Only after mapping to standard cells does the design become manufacturable.
- Advanced mapping tools (e.g., Synopsys, Cadence) allow for conditional and recursive cell mapping, matching wider logic patterns (such as arithmetic or multiplexers) to dedicated, optimized cells if available in the library.

Summary

RTL Expression	Generic Gate	Mapped Technology Cell
a & b	AND	AND2×1
`a	b`	OR	OR
a ^ b	XOR	XOR2×1
Flip-flop	DFF	DFFPOS×1

- Technology mapping is essential for transforming a functional logic description into a real, physical netlist ready for layout, fabrication, and manufacturing.
- It allows the synthesis tool to account for real-world parameters such as delays, drive strength, thresholds, and power/area trade-offs specific to your foundry's library.
- The process is highly customizable—designers can provide libraries for different technologies and re-map generic logic as needed.
- Technology mapping takes your logic design from the abstract, technology-independent domain to the real, physically realizable gates—bridging the gap between design intent and silicon implementation.

Gate-level Netlist Generation

- A gate-level netlist describes the design in terms of only primitive gates and their interconnections, without behavioral (RTL) constructs.
- Enables the use of back-end EDA tools for layout, simulation with real gate delays, and manufacturing.
- Typically written in structural Verilog, with each element directly corresponding to silicon gates available in the target library.

Gate-level Netlist Generation

- A gate-level netlist describes the design in terms of only primitive gates and their interconnections, without behavioral (RTL) constructs.
- Enables the use of back-end EDA tools for layout, simulation with real gate delays, and manufacturing.
- Typically written in structural Verilog, with each element directly corresponding to silicon gates available in the target library.

The Generation Process

1. **RTL Synthesis:** The RTL (behavioural Verilog) is parsed, elaborated, optimized, and mapped to the technology library.
2. **Netlist Creation:** The synthesis tool writes out the connected gates as a Verilog, which is now technology-specific.
3. **Usage:** This file is imported into place-and-route (layout) tools, timing analysis, or formal verification tools for downstream processing.

Gate-level Netlist Generation

Let's say your original RTL is:

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

After gate-level synthesis, the netlist might look like:

```
module simple_logic (a, b, y);
  input a;
  input b;
  output y;

  AND2X1 U1 (.A(a), .B(b), .Y(y)); // Instantiates AND gate from library
endmodule
```

Here, AND2×1 is a gate-instance from the chosen standard cell library. The netlist may also contain wiring declarations, more gate instances (e.g., flip-flops, buffers), and all module interconnections.

Gate-level Netlist Generation

Gate-Level Example for Sequential Logic

```
module dff_example (clk, d, q);
  input clk, d;
  output q;

  DFFPOSX1 U2 (.Q(q), .CLK(clk), .D(d));
endmodule
```

DFFPOS×1 is a standard-cell positive edge-triggered D flip-flop, provided by the technology library.

Gate-level Netlist Generation

Suppose your RTL had something like:

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

A gate-level structural netlist:

```
wire w;
AND2X1 U1 (.A(a), .B(b), .Y(w));
OR2X1  U2 (.A(w), .B(c), .Y(y));
```

This connects the output of an AND gate to an OR gate—both mapped to technology cells.

Gate-level Netlist Generation

Netlist and Back-End Integration

- Gate-level netlists include only primitive gates and wires, making them suitable for layout and timing-aware simulation.
- Back-end tools (place-and-route, physical synthesis) use these netlists to derive how the design will be manufactured and its physical constraints.
- Each signal in the netlist corresponds to wiring on silicon, and every gate to a library cell.

Formal Equivalence Checking (FEC)

FEC is a method used in ASIC design to mathematically prove that two representations of a circuit (usually the RTL and synthesized gate-level netlist) are functionally identical for all possible input combinations. The process eliminates the need for exhaustive simulation and is essential after synthesis to ensure that no functional errors have been introduced by optimizations or manual changes (ECOs).

Formal Equivalence Checking (FEC)

FEC is a method used in ASIC design to mathematically prove that two representations of a circuit (usually the RTL and synthesized gate-level netlist) are functionally identical for all possible input combinations. The process eliminates the need for exhaustive simulation and is essential after synthesis to ensure that no functional errors have been introduced by optimizations or manual changes (ECOs).

1. How FEC Works?

- **Inputs:** Two models (e.g., RTL and gate-level netlist)
- **Process:** Compares logic structures and output responses across all possible inputs
- **Tools:** Cadence Conformal LEC, Synopsys Formality, Siemens FormalPro
- **Outcome:** Proves equivalence or finds a counterexample (a mismatch)

Formal Equivalence Checking (FEC)

FEC is a method used in ASIC design to mathematically prove that two representations of a circuit (usually the RTL and synthesized gate-level netlist) are functionally identical for all possible input combinations. The process eliminates the need for exhaustive simulation and is essential after synthesis to ensure that no functional errors have been introduced by optimizations or manual changes (ECOs).

2. Typical Flow

- **Read and elaborate both models:** Parse and resolve module hierarchy for both the RTL and netlist.
- **Key point mapping:** Map registers, outputs, latches, and black-boxes between the two designs.
- **Formal comparison:** The tool checks that both models produce exactly the same output for every possible input, clock cycle by clock cycle (machine equivalence).
- **Report generation:** If equivalence fails, the tool pinpoints the mismatch and often produces a counterexample input vector.

Formal Equivalence Checking (FEC)

FEC is a method used in ASIC design to mathematically prove that two representations of a circuit (usually the RTL and synthesized gate-level netlist) are functionally identical for all possible input combinations. The process eliminates the need for exhaustive simulation and is essential after synthesis to ensure that no functional errors have been introduced by optimizations or manual changes (ECOs).

3. Example Code Snippets

Suppose you have an RTL and its synthesized netlist:

RTL (Golden Reference)

```
module simple_add(input [7:0] a, b,
output [7:0] sum);
  assign sum = a + b;
endmodule
```

Synthesized Gate-level Netlist

```
module simple_add (input [7:0] a, b, output [7:0] sum);
  ADD8X1 U1 (.A(a), .B(b), .Y(sum));
endmodule
```

Here, ADD8×1 is a technology library cell used by synthesis.

Formal Equivalence Checking (FEC)

FEC is a method used in ASIC design to mathematically prove that two representations of a circuit (usually the RTL and synthesized gate-level netlist) are functionally identical for all possible input combinations. The process eliminates the need for exhaustive simulation and is essential after synthesis to ensure that no functional errors have been introduced by optimizations or manual changes (ECOs).

FEC Tool TCL Flow (Cadence Conformal LEC example)

```
read design RTL_golden.v -Verilog -Golden
read design netlist_synth.v -Verilog -Revised
add compare points -all
compare
report verification
# If non-equivalent, diagnose and report test vector causing mismatch
diagnose -noneq -verbose -summary -revised
report test vector -noneq -ncsim > test_vector.tcl
```

The above script loads both files, sets up comparison points (outputs), runs the comparison, and diagnoses failures (if any).

Formal Equivalence Checking (FEC)

FEC is a method used in ASIC design to mathematically prove that two representations of a circuit (usually the RTL and synthesized gate-level netlist) are functionally identical for all possible input combinations. The process eliminates the need for exhaustive simulation and is essential after synthesis to ensure that no functional errors have been introduced by optimizations or manual changes (ECOs).

4. What Happens on Mismatch? (Debugging)

If FEC finds that for some input (say, `a=8'd255`, `b=8'd1`) the outputs from RTL and netlist differ:

- The tool will report the input vector causing mismatch.
- Generates a diagnostic report and a testbench snippet to reproduce and debug the failure.

For example:

```
initial begin
  a = 8'd255;
  b = 8'd1;
  #10;
  $display("RTL Output: %d, Netlist Output: %d", rtl_sum, netlist_sum);
end
```

This helps designers identify and fix any functional errors introduced during synthesis or manual netlist changes.

Formal Equivalence Checking (FEC)

FEC is a method used in ASIC design to mathematically prove that two representations of a circuit (usually the RTL and synthesized gate-level netlist) are functionally identical for all possible input combinations. The process eliminates the need for exhaustive simulation and is essential after synthesis to ensure that no functional errors have been introduced by optimizations or manual changes (ECOs).

5. Why Use FEC?

- Simulation cannot exhaustively test all scenarios, especially in large designs.
- Manual logic changes or ECOs risk introducing errors.
- Formality ensures that the optimized netlist is guaranteed to implement the same function as the reference RTL, eliminating doubt about synthesis tool or manual errors.

Formal Equivalence Checking (FEC)

FEC is a method used in ASIC design to mathematically prove that two representations of a circuit (usually the RTL and synthesized gate-level netlist) are functionally identical for all possible input combinations. The process eliminates the need for exhaustive simulation and is essential after synthesis to ensure that no functional errors have been introduced by optimizations or manual changes (ECOs).

5. Why Use FEC?

- Simulation cannot exhaustively test all scenarios, especially in large designs.
- Manual logic changes or ECOs risk introducing errors.
- Formality ensures that the optimized netlist is guaranteed to implement the same function as the reference RTL, eliminating doubt about synthesis tool or manual errors.

Formal Equivalence Checking is a must-have step in digital logic design: it proves, mathematically and exhaustively, that all optimizations, gate-level mappings, and manual changes have kept your design functionally intact—giving you confidence before costly manufacturing or tape-out steps. Tools automate the process with clear setup and debugging flows, leveraging your Verilog/RTL and netlist files as discussed in the previous slides.

Synthesis Design Environment & Constraints

- **Environment:** Uses tools such as Synopsys Design Compiler, Cadence Genus.
- **Constraints:**
  - **Timing:** Setup/hold, clock frequency.
  - **Area:** Limits on resource usage.
  - **I/O:** Pin mappings.
  - **Location:** Placement of specific logic.
- Effective constraint management is essential for meeting design goals and avoiding conflicts.

Architecture of Logic Synthesizer

- **Main Components:**
  - HDL Parser: Reads RTL code.
  - Elaboration Engine: Builds hierarchical design representation.
  - Logic Minimizer: Reduces gate count/levels.
  - Technology Mapper: Maps logic to library cells.
  - Optimizer: Refines for timing, area, power.
- Outputs gate-level netlist suitable for further place-and-route.

Architecture of Logic Synthesizer

Stage	Main Function	Example Code Snippet	Remarks

HDL Parser	Reads/syntax-checks HDL	`module simple_module(input a, input b, output y);`
		`  assign y = a & b;`
		`endmodule`
			Reads and parses the HDL code (e.g., Verilog), checks for syntax and semantic errors, and builds an internal data structure.
			Input: The original RTL code is provided to the parser.

Elaboration Engine	Binds parameters, resolves hierarchy	`module adder #(parameter WIDTH = 8) (`
		`  input [WIDTH-1:0] a, b,`
		`  output [WIDTH-1:0] sum`
		`);`
		`  assign sum = a + b;`
		`endmodule`
		
		`module top;`
		`  wire [15:0] x, y, z;`
		`  adder #(16) my_adder (.a(x), .b(y), .sum(z)); // Elaboration binds WIDTH=16`
		`endmodule`
			Resolves module hierarchies, expands parameters, unrolls generate blocks, binds all instances, and creates a complete, technology-independent representation.
			
			Result: All instances and parameters are resolved and ready for logic synthesis.

Logic Optimizer	Reduces area, speed, power	`module optimized_module(input sel, input [7:0] a1, a2, b1, b2, output [7:0] sum);`
		`  wire [7:0] add_a, add_b;`
		`  assign add_a = sel ? a1 : a2;`
		`  assign add_b = sel ? b1 : b2;`
		`  assign sum = add_a + add_b;`
		`endmodule`
			Applies technology-independent and -dependent optimizations to reduce area, improve timing, and lower power consumption. Optimization can include constant propagation, logic simplification, resource sharing, or pipelining.
			
			Effect: Simplifies logic or shares resources wherever possible.

Technology Mapper	Translates logic to library cells	`// Technology mapped netlist example`
		`module simple_and(input a, input b, output y);`
		`  AND2X1 U1 (.A(a), .B(b), .Y(y)); // AND2X1 is a library-specific cell`
		`endmodule`
			Maps the generic, optimized logic to the technology library's standard cells (e.g., NAND, NOR, specialized flip-flops). Considers the target process' constraints (timing, power, area).
			
			Result: All logic gates assigned to real, manufacturable cell types.

Netlist Generator	Outputs manufacturable gate-level netlist	`module dff_example(input clk, input d, output q);`
		`  DFFPOSX1 U2 (.CLK(clk), .D(d), .Q(q)); // Standard cell D flip-flop`
		`endmodule`
			Outputs the final gate-level netlist in structural Verilog or a similar format, listing all instantiated gates and their interconnections. This netlist is used for layout, simulation, and manufacturing.
			Output: Complete gate-level netlist with only library cells.