10 RTL Synthesis

...before attempting to translate our data into the rigorous language of symbols, it is above all things necessary to ascertain the ***intended*** *import of the words we are using. But this necessity cannot be regarded as an evil by those who value correctness of thought, and regard the right employment of language as both its instrument and its safeguard.*

—George Boole, *An Investigation of the Laws of Thought*, Chapter 4, 1854

We model hardware at the *register transfer level* (RTL) using hardware description languages (HDLs) such as Verilog and VHDL, as discussed in the preceding chapters. Subsequently, we *synthesize* the RTL model and obtain a netlist. During the initial phases of synthesis, we translate the RTL model to a netlist consisting of primitive logic gates, arithmetic blocks, and memory units, including registers. We refer to this step as *RTL synthesis*.

RTL synthesis involves analyzing the RTL model and instantiating appropriate circuit elements based on the semantics of the HDL. Therefore, it needs to comprehend various HDL constructs while translating them to circuit elements. In this chapter, we will illustrate the translation of a few essential Verilog constructs to hardware. These examples help us understand the correspondence between the Verilog constructs and the circuit elements. These concepts are often helpful while developing an RTL model, evaluating the impact of RTL code changes on the quality of result (QoR) measures, making manual RTL modifications, and interpreting the results of logic synthesis.

There are some optimization tasks that we can perform more efficiently at the level of RTL model. For example, we can efficiently carry out optimization related to resource allocation, arithmetic operators, multiplexer usage, and finite state machines (FSMs) at RTL because the HDL constructs allow easy identification of targets and make modifications at a higher abstraction level. We will discuss these optimizations also in this chapter.

10.1 LOGIC SYNTHESIS TASKS

We divide logic synthesis into a series of smaller tasks, as illustrated in Figure 10.1.

The initial portion of logic synthesis consists of *parsing* the RTL code, *elaboration*, RTL-specific optimization, and translation to primitive logic gates, arithmetic blocks, registers, memory units, and FSMs. We group these tasks as RTL synthesis and will discuss them in detail in this chapter.

After RTL synthesis, we carry out aggressive *logic optimization*, technology mapping, and technology-dependent logic optimizations. These tasks produce the final netlist that we can use for physical design. We will discuss these tasks in the subsequent chapters.

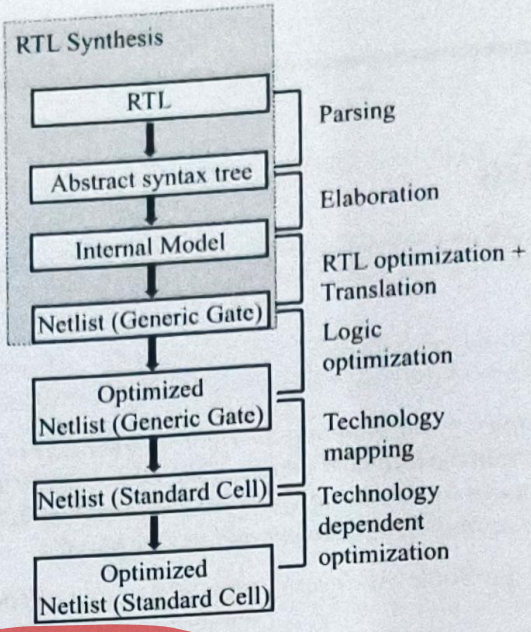

Figure 10.1 Major tasks in logic synthesis

10.2 PARSING

An RTL synthesis tool reads the given RTL files and populates a data structure for further processing. First, it breaks the sequence of characters in a file into a sequence of *tokens* or words that have special meaning for the given language. For example, the keywords in Verilog such as `always`, `module`, `endmodule`, `begin`, `end`, etc. can be the tokens [1]. This process is known as *lexical analysis*. Then, it analyzes whether the grammar of the given HDL is honored in the given RTL code. If there are any syntax errors, the tool reports them. If the RTL code is error-free, the tool populates a hierarchical data structure, typically in the form of a *syntax tree*. This process is known as *parsing*.

Example 10.1 Consider the following piece of Verilog code.

```
module top();
endmodule
module mid();
        ...
        always @ (*)
        begin
              a <= x+y;
              b <= 0;
              c = x;
        end
endmodule
```

The *abstract syntax tree* (AST) for the above code is shown in Figure 10.2. If a construct or design entity P contains another construct or design entity Q, we make P the parent of Q. Note that the AST of Figure 10.2 shows only the elements present in the above code.

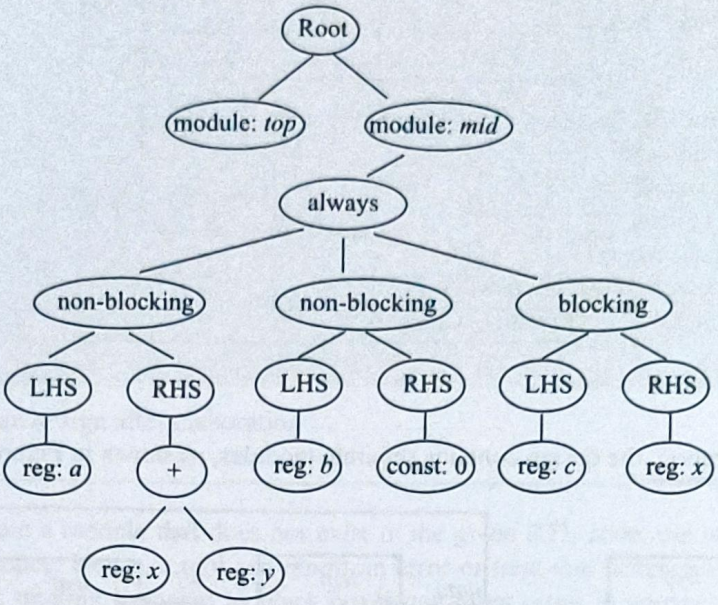

Figure 10.2 An abstract syntax tree

10.3 ELABORATION

After parsing, *elaboration* is carried out for a design. During elaboration, a tool checks whether the connections among RTL-specified components are legitimate. If they are legitimate, the tool will make the connections in the internally created design model, else it will report an error or warning depending on the severity of the violation. For example, if a module instantiates a sub-module with a port name A, the sub-module should contain a port named A. If the port named A does not exist, the tool will report an error. Similarly, if the port widths in the instantiation and its master module are inconsistent, the tool will report an error or warning.

Example 10.2 Consider the following piece of Verilog code.

```
module leaf(d, clk, q);
      input d, clk;
      output q;
      ...
endmodule
```

```
module middle(D, CLK, Q);
        input D, CLK;
        output Q;
        leaf F1(D, CLK, Q);
endmodule

module Top(data, clk, result);
        input data, clk;
        output result;
        wire w1, w2;
        leaf I1(data, clk, w1);
        leaf I2(.d(w1), .clk(clk), .q(w2));
        middle I3(.D(w2), .Q(result), .CLK(clk));
endmodule
```

Before elaboration, the design contains separate modules, as shown in Figure 10.3.

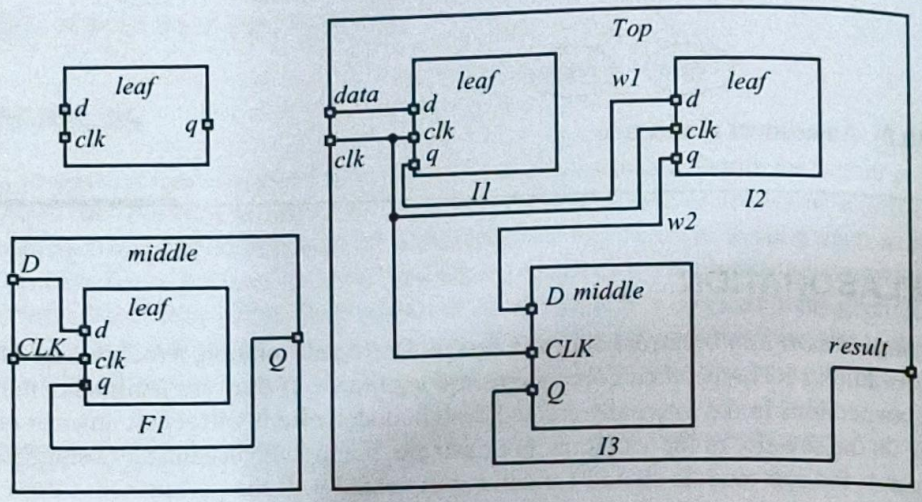

Figure 10.3 Given design before elaboration

The modules are not linked, and the hierarchy of modules is not yet created. Note that the information on the direction of the instance pin is initially only in the master module (e.g., the direction of pin *d* of instance *I1* is in the master module *leaf*). Hence, before elaboration, the tool can assume any direction for the instance pin. In this example, we have shown their direction as input.

After elaboration, the modules get linked, and the hierarchy of the modules gets created, as shown in Figure 10.4. Additionally, the direction of instance pins gets inferred, and appropriate connections are made. If there were any inconsistencies in the instance pin and master module's port (in name or width), the tool would have reported them.

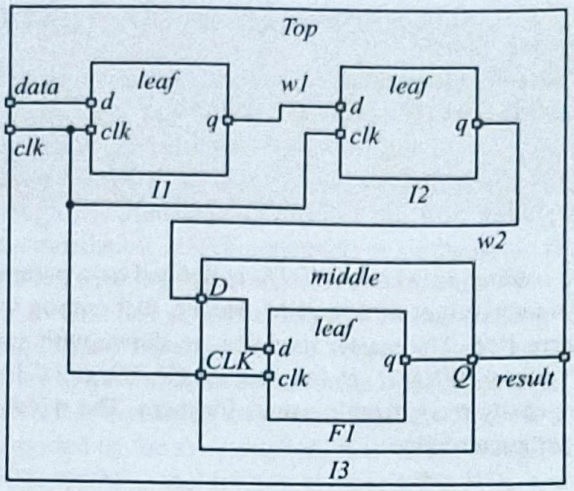

Figure 10.4 Given design after elaboration

If we instantiate a module that does not exist in the given RTL code, the tool will be unable to resolve its reference. Hence, a tool can report an error or treat that instance as a *black box* and proceed. However, treating instances as black boxes can often result in unexpected behavior, and we should try to avoid undefined references in a design hierarchy. Such problems typically occur when we have not provided the complete set of RTL files to the synthesis tool or provided the wrong module name during instantiation.

Elaboration needs to process *parameterized modules* differently because they can have different interfaces for varying parameters. Typically, elaboration creates *separate* modules with different interfaces for each distinct set of parameters. A tool can name such an internally created module with its own naming convention. We illustrate it in the following example.

Example 10.3 Consider the following piece of Verilog code.

```
module counter(clk, rst, count);
      parameter WIDTH=4;
      input clk, rst;
      output  [WIDTH-1:0] count;
      reg  [WIDTH-1:0] count;
      ...
endmodule
module top(clk, rst, count1, count2, count3);
      input clk, rst;
      output [15:0]count1;
      output [7:0]count2;
      output [3:0]count3;
      ...
```

```
        counter C1 (clk, rst, count3) ;
        counter # (8) C2 (clk, rst, count2) ;
        counter # ( .WIDTH (16) ) C3 (clk, rst, count1) ;
        ...
endmodule
```

It contains a module *counter* in which *WIDTH* is defined as a parameter. It is instantiated in the module *top* with parameter values 4, 8, and 16. Hence, elaboration will produce three master modules as shown in Figure 10.5. The master modules are shown with names *counter_WIDTH_4*, *counter_WIDTH_8*, and *counter_WIDTH_16* for instance *C1*, *C2*, and *C3*, respectively. A synthesis tool can choose any other easily recognizable names for them. The width of the output port *count* varies depending on the parameter value.

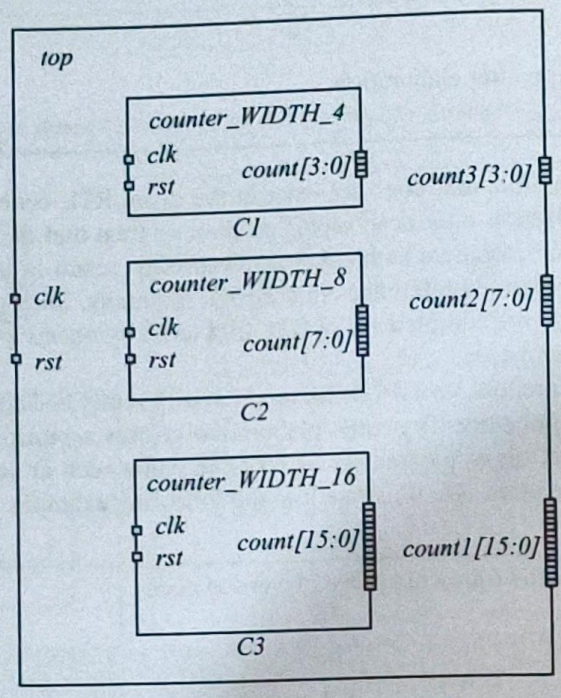

Figure 10.5 Given design after elaboration (connections not shown)

Sometimes we need to implement different instances of the same module differently, irrespective of whether they are parameterized. For example, one instance of a module can be on the critical path of a circuit, while two others instances of the same module are not timing critical. In such cases, we want different instances of the same module to be implemented differently. To achieve this, we can create separate copies of the module for different instances, despite having the same interface and the RTL code. This process is known as *uniquification*. Note that elaboration does not carry out uniquification by default. We need to instruct synthesis tools to perform uniquification, and industry-standard synthesis tools typically provide a mechanism for it.

10.4 VERILOG CONSTRUCTS TO HARDWARE

Once elaboration is done, the synthesis tool has created the design hierarchy and connected the RTL-specified instances. Thus, a general structure of the netlist is already built. However, the internal details of the modules are not yet determined. A module can contain various HDL constructs. An RTL synthesis tool needs to comprehend these constructs, transform them into circuit elements, and instantiate them. Though the transforming process can vary among RTL synthesis tools, the HDL semantics govern the translation of RTL constructs to hardware. In this section, we present the translation of some commonly used Verilog constructs to hardware.

Besides providing a mechanism for modeling hardware, Verilog has constructs that ease simulation and other design tasks. Therefore, some Verilog constructs cannot be synthesized into circuit elements, and they are helpful in other design tasks. Moreover, an RTL synthesis tool may not support all synthesizable Verilog constructs or modeling styles. Therefore, we need to be aware of the Verilog constructs supported by the synthesis tool that we are using for design implementation.

The behavior of an RTL synthesis tool for non-synthesizable HDL constructs is tool dependent. It can ignore constructs that it cannot synthesize and continue with the remaining RTL code. For example, # delay specification is not supported in RTL synthesis. When a synthesis tool encounters a Verilog statement such as:

```
out1 <= #12 a;
```

it will treat the statement simply as:

```
out1 <= a;
```

Note that the delay specification such as #12 is easy to model and consider during simulation. Therefore, we commonly use delay specification in testbenches for functional verification. However, designing hardware with an exact delay specification is not possible for various reasons, such as process-induced and environmental variations. Moreover, we do not need delay elements with an exact delay specification in a circuit. Typically, we need circuit elements that have delay within a given *range* or with some allowed variations.[1]

Some other constructs of Verilog that are typically not supported by RTL synthesis tools are `initial` block, `fork`, `join`, `force`, `release`, data types `real` and `time`, `$display`, `$monitor`, and other systems tasks.

In the following paragraphs, we describe the direct translation of some commonly used Verilog constructs to hardware. However, note that synthesis tools are free to implement them using other functionally equivalent hardware. Additionally, subsequent optimization can change the circuit structure obtained using direct translation.

10.4.1 Assign Statement

An `assign` statement assigns a Verilog expression to a wire or a vector of wires. Depending on the right-hand side (RHS) of the `assign` statement, various combinational circuit elements can be inferred. The following example illustrates it.

[1] We will discuss tackling delay of circuit elements in Chapter 14 ("Static timing analysis").

Example 10.4 Consider the following Verilog code.

```
module top(a, b, c, p, q, s, x, y, out1, out2, out3);
    input a, b, c, p, q, s;
    input [3:0]x, y;
    output out1;
    output [3:0]out2;
    output out3;
    assign out1 = (a & b) | c; // logic network
    assign out2 = (x & y) ; // bitwise logic network
    assign out3 = (s) ? q : p; // multiplexer
endmodule
```

One possible synthesis result is shown in Figure 10.6.

The first `assign` statement results in a logic network defined by the RHS expression.

The second `assign` statement has a 4-bit vector on the left-hand side (LHS). It results in four identical logic structures driving each bit of the vector.

The third `assign` statement results in a two-to-one multiplexer due to the conditional operator.

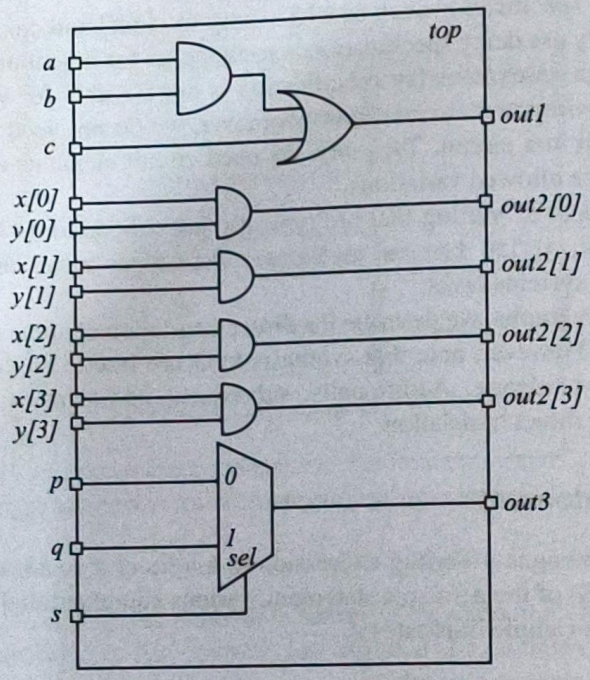

Figure 10.6 Synthesized circuit for the given Verilog code (with `assign` statements)

10.4.2 If–else Statement

An `if–else` statement translates to a multiplexer or selecting logic. If it results in a multiplexer, the signal appearing in the `if` clause goes to the select lines of the multiplexer.

Example 10.5 Consider the following Verilog code.

```
module top(a, b, s, out1);
    input a, b, s;
    output out1;
    reg out1;

    always @(*) begin
        if (s==1'b0)
            out1 = a;
        else
            out1 = b;
    end
endmodule
```

One possible synthesis result is shown in Figure 10.7.

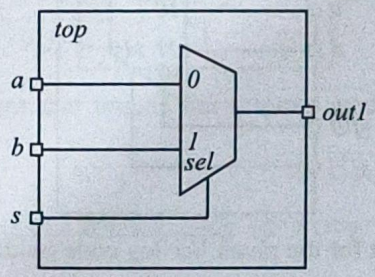

Figure 10.7 Synthesized circuit for the given Verilog code (with an `if–else` statement)

10.4.3 Case Statement

A case statement (specified using `case`, `casex`, or `casez`) also translate to multiplexers or select logic. The size of the multiplexer depends on the size of the control signal in the case statement.

Example 10.6 Consider the following Verilog code.

```
module top(a, b, c, d, s, out1);
    input a, b, c, d;
```

```
    input [1:0]s;
    output out1;
    reg out1;

    always @(*) begin
        case (s)
            2'b00: out1 = a;
            2'b01: out1 = b;
            2'b10: out1 = c;
            2'b11: out1 = d;
            default: out1=a
        endcase

    end
endmodule
```

One possible synthesis result is shown in Figure 10.8. It contains a four-to-one multiplexer.

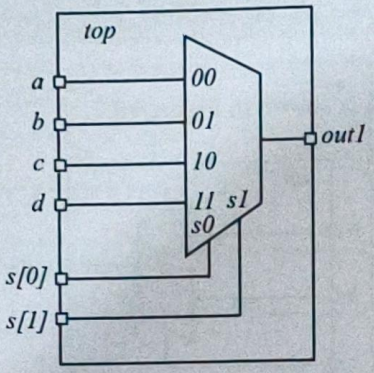

Figure 10.8 Synthesized circuit for the given Verilog code (with a case statement)

Verilog permits a case statement in which multiple items match. If there are multiple matches for a case statement, the *first* matched item gets the priority. Hence, priority multiplexers are inferred for overlapping matches in the case statement.

Example 10.7 Consider the following Verilog code.

```
module top(a, b, c, s, out1);
    input a, b, c;
    input [1:0]s;
    output out1;
```

```
    reg out1;
    always @(*) begin
       casez (s)
          2'b1?: out1 = a;
          2'b?1: out1 = b;
          default: out1 = c;
       endcase
    end
endmodule
```

One of the possible synthesis results is shown in Figure 10.9. Note that the multiplexer connections implement the priority specified in the case statement. The condition (*s[1]*=1) gets the highest priority, followed by (*s[0]*=1), and the default condition.

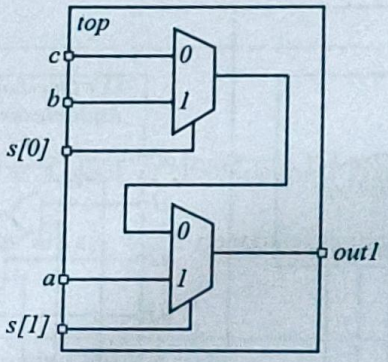

Figure 10.9 Synthesized circuit that retains the priority modeled by the case statement using multiplexers

A similar kind of priority can be defined using `if-else-if` construct also, as shown in the following code.

```
module top(a, b, c, s, out1);
    input a, b, c;
    input [1:0]s;
    output out1;
    reg out1;

    always @(*) begin
       if (s[1] == 1'b1)
          out1 = a;
       else if (s[0] == 1'b1)
          out1 = b;
       else
```

```
            out1 = c;
    end
endmodule
```

It will get synthesized similar to the circuit shown in Figure 10.9. However, note that an RTL synthesis tool can internally employ other kinds of multiplexers and produce a different but functionally equivalent circuit. For example, it can use a *one-hot multiplexer* for representing the circuit, as shown in Figure 10.10. A one-hot multiplexer has N data inputs and N select inputs. If ith select input is 1, it produces the ith data input at the output pin. Additionally, only one select input pin can have a value of 1 in a one-hot multiplexer (otherwise it can produce incorrect output). We can easily translate the control logic of `if`–`else`–`if` and case statements to one-hot multiplexer-based circuit. Additionally, it allows easier optimization of the control logic. Therefore, one-hot multiplexer-based representation of control logic has merits in RTL synthesis.

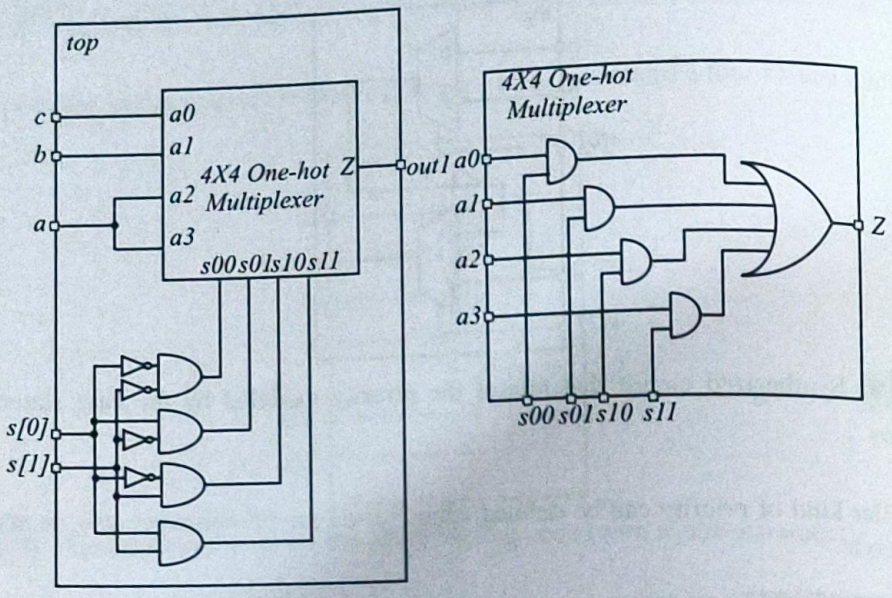

Figure 10.10 Synthesized circuit that uses one-hot multiplexer

For example, we can convert a 4×4 one-hot multiplexer of Figure 10.10 to a 3×3 multiplexer because two inputs are identical (port *a* feeds *a2* and *a3* pins of the multiplexer) [2]. The resultant circuit is shown in Figure 10.11. We have merged the data pin *a3* with *a2*. Additionally, the corresponding select pin *s10* is driven by the logical OR of the original control signals, i.e., *s[0]'.s[1]+s[0].s[1]=s[1]*. Further, we have eliminated pin *s11*. Thus, we obtain an optimized circuit shown in Figure 10.11. Subsequently, we can convert the optimized one-hot multiplexer-based circuit to binary multiplexer-based circuit or to other generic gates. For complicated control logic, the one-hot multiplexer-based optimizations can be more powerful and efficient.

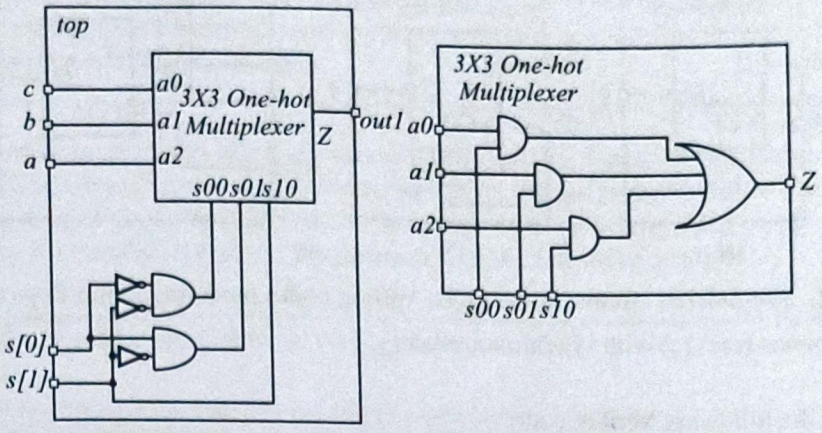

Figure 10.11 Synthesized circuit after optimizing one-hot multiplexer

10.4.4 Always Block

An always block can be inferred as a block of combinational logic or a block of sequential logic containing flip-flops and latches.

When the sensitivity list of an always block contains `posedge` or `negedge`, flip-flops are inferred.

Example 10.8 Consider the following Verilog code.

```
module DFlipFlop(d, clk, q);
    input d, clk;
    output q;
    reg q;

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

The always block is sensitive to the rising edge of *clk*. When the rising edge of *clk* occurs, the value at *d* gets written to *q*. At other time instants, *q* holds its previous value. Hence, a positive-edge triggered D flip-flop gets inferred, as shown in Figure 10.12(a).[2]

[2] A negative-edge triggered flip-flop gets inferred when the `always` block is sensitive to the `negedge` of *clk*.

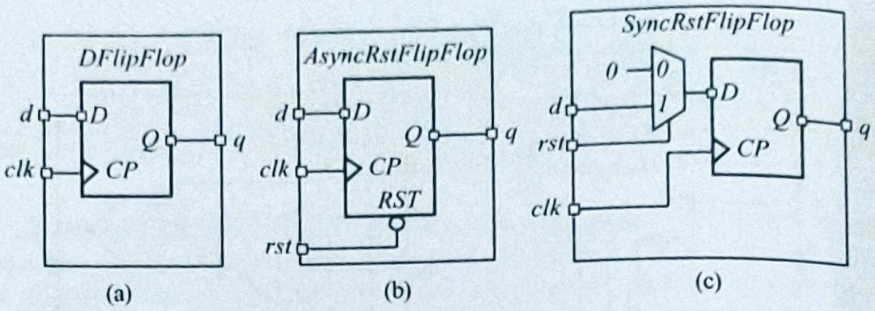

Figure 10.12 Synthesized circuit for the given Verilog codes modeling D flip-flops (a) simple (b) with asynchronous reset (c) with synchronous reset

Consider the following Verilog code.

```
module AsyncRstFlipFlop(d, clk, q, rst);
    input d, clk, rst;
    output q;
    reg q;

    always @( posedge clk or negedge rst)
        if (rst == 1'b0) begin
            q <= 1'b0;
        end else begin
            q <= d;
        end
endmodule
```

The always block is sensitive to rising edge of *clk* and falling edge of *rst*. When *rst*=0, *q* gets reset to 0. Hence, an asynchronously reset flip-flop gets inferred, as shown in Figure 10.12(b).

Consider the following Verilog code.

```
module SyncRstFlipFlop(d, clk, q, rst);
    input d, clk, rst;
    output q;
    reg q;

    always @( posedge clk)
        if (rst == 1'b0) begin
            q <= 1'b0;
        end else begin
            q <= d;
        end
endmodule
```

The always block is sensitive to only the rising edge of *clk*. When the rising edge of *clk* occurs, *q* becomes 0 if *rst*=0, else the value at *d* gets assigned to *q*. Hence, a flip-flop with synchronous reset gets inferred, as shown in Figure 10.12(c).

The statements in an always block are executed sequentially. Therefore, the connections of flip-flops inferred in an edge-sensitive always block depend on the assignments within that block. Since the mechanism of updating LHS is different for blocking and nonblocking assignments, the synthesis results can vary for these assignments. We illustrate it in the following example.

Example 10.9 Consider the following Verilog code.

```
module top1(in1, clk, out1);
    input in1, clk;
    output out1;
    reg reg1, reg2, reg3, out1;

    always @(posedge clk)
        begin
            reg1 = in1;
            reg2 = reg1;
            reg3 = reg2;
            out1 = reg3;
        end
endmodule
```

The assignments inside the always block are blocking. Therefore, whenever a rising clock edge occurs, the input *in1* will be assigned to register *reg1*, then to *reg2*, then to *reg3*, and finally to *out1*. Thus, *in1* gets transferred to *out1* at every clock edge. Hence, the above code can be synthesized to the circuit shown in Figure 10.13(a). The values of *reg1*, *reg2*, and *reg3* are redundant (not used in the circuit) and the flip-flops corresponding to them will be eliminated.

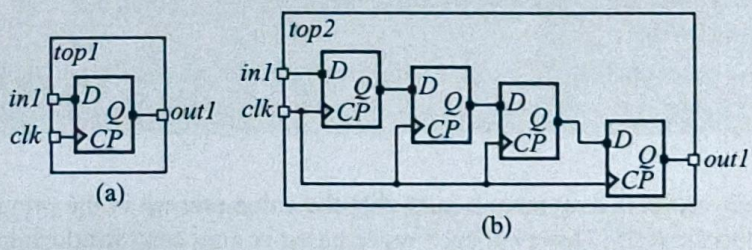

Figure 10.13 Synthesized circuit for the given Verilog codes

However, we often want a pipeline behavior (similar to a shift register) in the above Verilog code. We can achieve it using nonblocking assignments, as shown in the following code.

```
module top2(in1, clk, out1);
    input in1, clk;
    output out1;
    reg reg1, reg2, reg3, out1;

    always @(posedge clk)
        begin
            reg1 <= in1;
            reg2 <= reg1;
            reg3 <= reg2;
            out1 <= reg3;
        end
endmodule
```

Recall that a nonblocking assignment gets executed in two steps. In the first step, only the RHS is evaluated and the LHS is only scheduled to be updated with the RHS. In the second step, its LHS is updated. Hence, when the rising clock edge appears, the value *reg2* gets updated with the value of *reg1* held in the previous cycle. Similarly, *reg3* is updated with the value of *reg2* of the previous cycle, and *out1* is updated with the value of *reg3* in the previous cycle. Thus, we obtain the behavior of a shift register in the above code, and it will be synthesized as shown in Figure 10.13(b).

We can obtain a shift register by reordering blocking assignments also, as shown below.

```
module top1(in1, clk, out1);
    input in1, clk;
    output out1;
    reg reg1, reg2, reg3, out1;

    always @(posedge clk)
        begin
            out1 = reg3;
            reg3 = reg2;
            reg2 = reg1;
            reg1 = in1;
        end
endmodule
```

The above code reorders assignments such that the value present in the previous cycle on the LHS gets updated to the RHS. However, such order-based coding can introduce inadvertant errors. Therefore, we should prefer to use nonblocking assignments in always block that we expect to generate a sequential logic [3].

When the sensitivity list does not contain `posedge` or `negedge` constructs (i.e., it has only level-sensitive signals), the always block is inferred as a block consisting of purely combinational circuit elements or latches.

When the value of a variable is updated (refreshed) in every possible path (conditional branches in the code) within an always block, a purely combinational logic block gets inferred. However, if it retains its old value in some paths of the always block, a latch gets inferred. The following example illustrates it.

Example 10.10 Consider the following Verilog code.

```
module top(a, b, c, s, en, out1, out2);
    input a, b, c, s, en;
    output out1, out2;
    reg out1, out2;

    always @(*) begin
        if (s==1'b0)
            out1 = a;
        else
            out1 = b;
    end

    always @(*) begin
        if (en==1'b1)
            out2 = c;
    end
endmodule
```

In the first always block, the variable *out1* gets assigned. It gets updated to either *a* or *b*, whenever the always block is executed. Hence, the variable *out1* gets updated in all possible paths in the always block. Therefore, it gets synthesized to a purely combinational logic block. Since there are two possible cases for the `if` statement, a multiplexer is typically inferred for this always block, as shown in Figure 10.14(a).

In the second always block, the variable *out2* gets assigned. It gets updated to *c* only when *en*=1. It implies that it retains the old value when *en*=*0*. This always block is equivalent to the following Verilog code.

```
    always @(*) begin
            if (en==1'b1)
                out2 = c;
            else
                out2 = out2;
    end
```

The above portion of code can be synthesized as shown in Figure 10.14(b). It has a multiplexer in which the output recirculates if *en*=0. However, note that a multiplexer with a recirculating output is functionally equivalent to a latch.[3] Hence, the second always block gets synthesized to a latch, as shown in Figure 10.14(a).

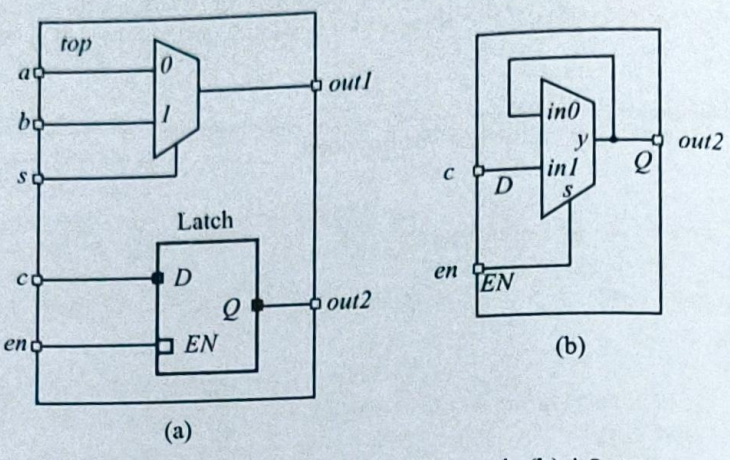

Figure 10.14 Synthesized circuit for the (a) given Verilog code (b) `if` statement in which the value gets retained when the condition is false

The above example illustrates that latches get inferred when a variable is not updated in all possible paths in an always block. Sometimes we need to model such a situation, and latches are indeed required in our design. However, latches often get inferred due to incorrect modeling of a combinational block. For example, if we inadvertently miss updating value in one of the branches of a `case` statement, a latch will be inferred. To avoid such problems, we should use `default` case to cover all possible paths in a `case` statement. Alternatively, we can set the output of a combinational logic to a default value at the beginning of the always block. It will cover the cases in which the output is not updated in some conditional branches within the always block, and we will obtain a combinational logic block.

Example 10.11 Consider the following Verilog code.

```
module top(in1, out1);
    input [0:1]in1;
    output out1;
    reg out1;

    always @* begin
        case(in1[0:1])
            2'b00: out1 = 1'b0;
```

[3] See Section 1.1.6 ("Sequential circuits") for the latch implementation.

```
            2'b01: out1 = 1'b1;
            2'b10: out1 = 1'b1;
        endcase
    end
endmodule
```

A latch will be inferred because there is no update on *out1* when *in1*=2'b11. Hence, *out1* would retain the previous value for this case. If we want that a purely combinational circuit is inferred, we need to define the behavior for the case *in1*=2'b11 also. An easy workaround for the `case` statement would be to add a `default` branch, as shown below.

```
module top(in1, out1);
    input [0:1]in1;
    output out1;
    reg out1;

    always @* begin
        case(in1[0:1])
            2'b00: out1 = 1'b0;
            2'b01: out1 = 1'b1;
            2'b10: out1 = 1'b1;
            default: out1 = 1'b0;
        endcase
    end
endmodule
```

Another mistake we can make is to omit some input signals of a combinational block from the sensitivity list of an always block. It will not change the synthesis output because the inference of inputs of combinational logic is not based on the sensitivity list but on the RHS of the assignments within that block [4]. However, it can lead to *mismatches in the simulation results* of the RTL model, and the post-synthesis netlist [4]. The problem can occur because the RTL simulation will not enter the always block for signal transitions missed in the sensitivity list. However, the gate-level simulation of the synthesized netlist will cause the combinational logic to respond to those transitions. Therefore, the simulation of the RTL model and gate-level netlist will be inconsistent. It can open up verification gaps and lead to design failure.

Example 10.12 Consider the following Verilog code.

```
module top(a, b, c, out1, out2);
    input a, b, c;
```

```
    output out1, out2;
    reg out1, out2;

    always @(a or b or c)
        out1 = a & b & c;

    always @(a or b)
        out2 = a & b & c;
endmodule
```

The first always block has all the signals on which *out1* depends in the sensitivity list. It will be synthesized into a three-input AND gate. The RTL model and gate-level netlist will give the same simulation results.

The second always block has *c* missing from the sensitivity list. However, the expression for *out2* contains *c* also. Nevertheless, it will be synthesized into a three-input AND gate based on the RHS expression. However, the RTL simulation of this always block will produce different results because a change to the variable *c* alone will not be observed at *out2*.

To avoid the above problem, we can use `@*` in the sensitivity list of the combinational logic.

Sometimes an always block can be non-synthesizable. For example, an always block in which the sensitivity list contains both edge- and level-triggered signals is non-synthesizable because, typically, we do not have hardware elements that match such behavior.

10.4.5 For Loops

A `for` loop can infer repeated logic structure. Typically, an RTL synthesis tool unrolls (expands) the loop and instantiates circuit elements for each iteration. Therefore, the synthesis tool must know the number of iterations in a `for` loop *during synthesis*. Hence, a `for` loop is synthesizable only when the number of iterations is *fixed*.

Example 10.13 Consider the following Verilog code.

```
module top(a, b, cin, sum, cout);
    input [3:0]a, b;
    input cin;
    output [3:0]sum;
    output cout;
    reg [3:0]sum;
    reg cout;
    reg carry;
    reg [2:0]idx;

    always @(*) begin
```

```
        carry = cin;
        for (idx=0; idx < 4; idx=idx+1)
        begin
            {carry, sum[idx]}=a[idx]+b[idx]+carry;
        end
        cout = carry;
    end
endmodule
```

The `for` loop iterates four times. Therefore, RTL synthesis tool will unroll the loop as follows:

```
{carry, sum[0]}=a[0]+b[0]+cin;
{carry, sum[1]}=a[1]+b[1]+carry;
{carry, sum[2]}=a[2]+b[2]+carry;
{cout, sum[3]}=a[3]+b[3]+carry;
```

Hence, it will use four 1-bit full adders in a cascaded configuration, as shown in Figure 10.15.

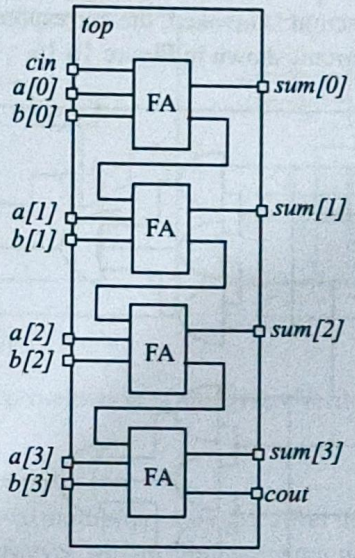

Figure 10.15 Synthesized circuit for the given Verilog code with a `for` loop

10.4.6 Functions

A Verilog `function` synthesizes to a combinational logic block with one output (scalar or vector).

Example 10.14 Consider the following Verilog code.

```
module top(a, b, c, d, e, out1, out2);
    input a, b, c, d, e;
```

```
    output out1, out2;
    reg out2;

    function MAJOR3;
        input A, B, C;
        begin
            MAJOR3 = (A&B) | (B&C) | (C&A);
        end
    endfunction
    assign out1 = MAJOR3(a, b, c);
    always @(*) begin
        out2 = MAJOR3(c, d, e);
    end
endmodule
```

The function *MAJOR3* will get synthesized to the combinational logic block based on the given Boolean expression. When the function is invoked, the corresponding combinational logic block is instantiated. Thus, we obtain the circuit shown in Figure 10.16.

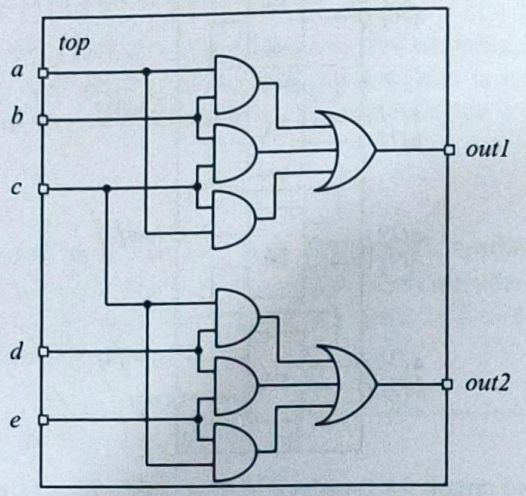

Figure 10.16 Synthesized circuit for the given Verilog code with `function`

10.4.7 Operators

An RTL synthesis tool can directly translate *bit-wise operators* and logical operators to their corresponding logic gates and optimize them subsequently. However, *arithmetic* and *relational* operators are handled differently. An RTL synthesis tool first translates them to some internal representation or data structure, such as a graph with the operator–operand relationship preserved. The data structure is chosen that facilitates *optimizations* at the *operator level*. We explain two such optimization techniques, *resource sharing* and *speculation*, in the following paragraphs.

1. **Resource sharing:** We can utilize the same computational element for multiple computations. This optimization strategy is known as *resource sharing*. By sharing resources we can save area. However, we need to ensure that its impact on the timing of a circuit is acceptable. We illustrate it in the following example.

Example 10.15 Consider the following piece of Verilog code.

```
if (sel == 1'b0)
    z = a*b;
else
    z = x*y;
```

Assume that a, b, x, and y are of 8 bits, and z is of 16 bits. We can represent the above code schematically using Figure 10.17(a).

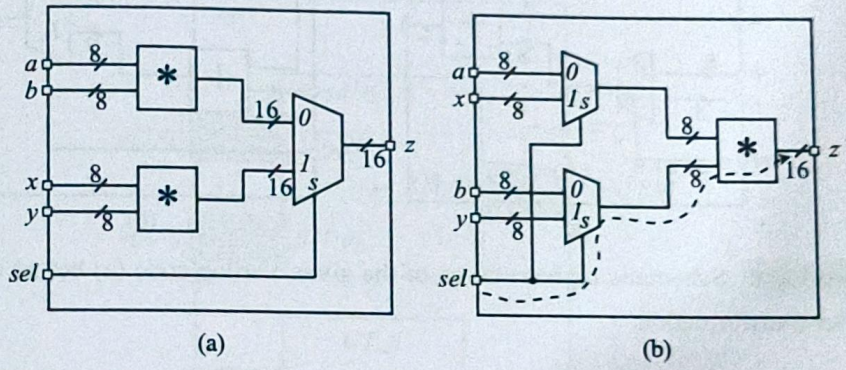

Figure 10.17 Schematic representation of the given Verilog code (a) before transformation, (b) after transformation

In this implementation, two multipliers are used. After multiplication, the result from one of the multipliers is selected. However, we can eliminate a multiplier by transforming the circuit, as shown in Figure 10.17(b). In this case, we select the operands that need to be multiplied before multiplying. Since multipliers are expensive resources in terms of area, this transformation helps us save area significantly. However, note that the multiplier now appears on the path through the select pins of the multiplexers (as indicated by the dashed line). Hence, the delay through the select pins of the multiplexers will increase due to this transformation, and we need to ensure that this path does not violate the timing constraint (in addition to other paths through the multiplier).

2. **Speculation:** Sometimes, we can improve performance by adding extra resources or *unsharing resources*. We illustrate it in the following example.

Example 10.16 Consider the following piece of Verilog code. Assume that a, b, c, and y are of 8 bits.

```
if (sel == 1'b0)
    y = b;
else
    y = c;
z = a+y;
```

We can represent it schematically using Figure 10.18(a). Assume that the path through the *sel* port is critical, as indicated by the dashed line. Hence, the adder is on the critical path. It computes either $(a+b)$ or $(a+c)$ depending on the value of *sel*.

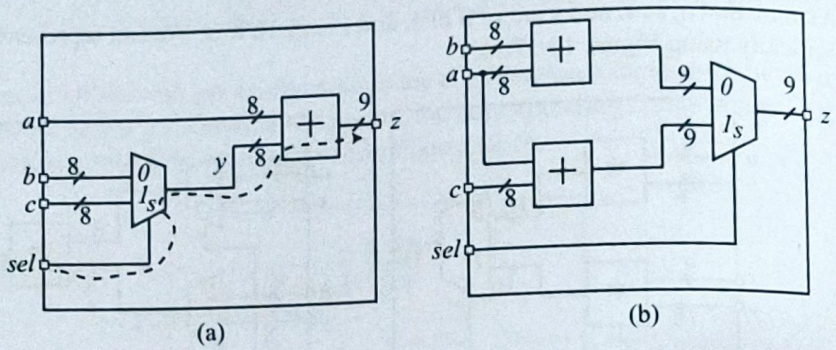

Figure 10.18 Schematic representation of the given Verilog code (a) before transformation, (b) after transformation

We can remove the adder from the critical path by performing addition before selection, as shown in Figure 10.18(b). In this implementation, two adders are used. After addition, the result from one of the adders is selected. As a result, the adders are not on the path through the port *sel*, and the critical path delay reduces. Note that an 8-bit adder typically exhibits a considerable delay. Therefore, the above transformation can reduce the critical path delay significantly. However, this transformation needs an extra adder and incurs an area penalty.

In the above transformation, we perform addition irrespective of the sum being needed (the result of only one adder will be finally selected). This transformation is an example of an optimization technique known as *speculation* or *eager computation*. It improves performance by employing more resources.

Mapping and architecture selection: After optimizing arithmetic and relational operators at the operator level, we map them to predefined modules implementing these operators. For example, an RTL synthesis tool can have a set of *internal parameterized modules* implementing arithmetic operators (+, −, *, /) and relational operators (==, >, >=, <, <=). It will instantiate these internal modules for arithmetic and relational operators, and choose the right set of parameters for the parameterized modules.

Example 10.17 Consider the following Verilog code.

```
module top(a, b, c, d, sum, carry, comp);
    input [7:0]a, b, c, d;
    output [7:0]sum;
    output carry, comp;

    assign {carry, sum} = a + b;
    assign comp = (c > d);
endmodule
```

The code contains operators + and >. An RTL synthesis tool will map the operator + to an internal module *ADD_8* and > to an internal module *GT_8*. As a result, we will obtain a circuit shown in Figure 10.19.

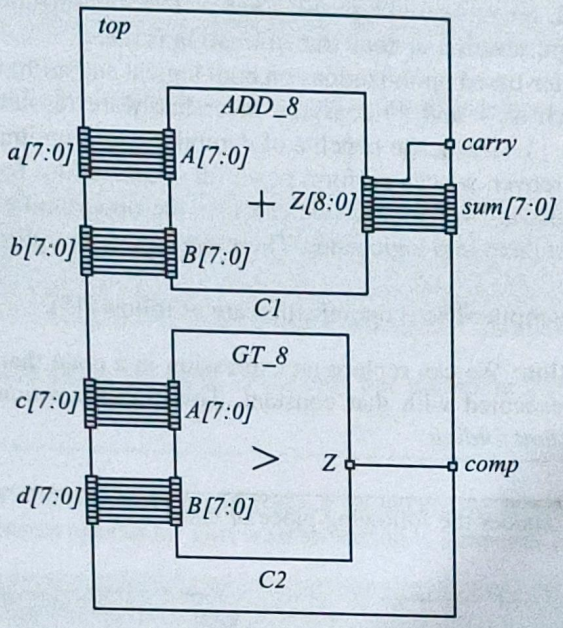

Figure 10.19 Schematic of the synthesized circuit for the given Verilog code (*ADD_8* and *GT_8* represent internal modules of an RTL synthesis tool). A tool can choose any name for these modules

The following points are noteworthy regarding the internal modules corresponding to the arithmetic operators:

1. The implementation of these modules is tool-specific. However, the logical functionality will be the same (as defined by the semantics of the language).

2. In subsequent stages of logic synthesis, the instances of these modules get *flattened* (module hierarchy gets dissolved), optimized, and mapped to the cells of the given technology libraries.
3. An RTL synthesis tool can have multiple implementations with varying architectures for the same operator. These architectures can represent different performance–area trade-offs. For example, it can have two implementations for the adder *ADD_8*: ripple carry adder (RCA) and carry-look ahead adder (CLA). An RCA has a smaller area, while CLA is faster. A synthesis tool can switch the architecture of the internal modules depending on whether the adder instance is timing critical or not. Such architectural switching can be done in the later stages of logic synthesis also. However, it should be done before flattening the internal modules because once they get flattened, it becomes challenging to reconstruct their original boundary and carry out the transformation.

10.5 COMPILER OPTIMIZATION

RTL synthesis tools employ various compiler optimization techniques by adapting them to the RTL code. These optimizations can be applied to the parse tree or the internal model of the RTL created after elaboration. An RTL synthesis tool typically applies these optimizations in passes, and in each pass, a specific type of optimization or code transformation is done.

We can apply compiler-based optimizations on both logical and arithmetic operators. However, arithmetic operators (such as + and *) consume more hardware resources compared to logical operators (such as & and |). Hence, the benefits of compiler-based optimizations are more for the arithmetic operators. Moreover, we can perform powerful optimizations for logical operators in the later stages of logic synthesis.[4] In contrast, we can lose the opportunities to optimize arithmetic operators once we convert them into logic gates. Therefore, compiler optimizations are targeted for arithmetic operators.

A few examples of compiler-based optimization are as follows [5].

1. **Constant propagation:** We can replace an expression in a code that evaluates to a *constant* every time it gets executed with that constant. This transformation is known as *constant propagation* or *constant folding*.

Example 10.18 Consider the following piece of code.

```
a = 8*8;
b = (a*1024)/32;
c = (b+32+b+32);
```

We can identify *a* as a constant and compute its value during compile time. Further, we can substitute its value in *b*. We can again identify *b* as a constant and propagate it to *c*. We can also compute *c* at the compile time. Thus, the above code transforms to the following code after constant propagation.

[4] We will describe logic optimization in Chapter 12 ("Logic optimization").

```
a = 64;
b = 2048;
c = 4160;
```

. **Common subexpression elimination:** We can replace identical subexpression in multiple expressions with a single variable if it reduces the cost, such as area. A synthesis tool typically performs common subexpression elimination for the *arithmetic* operators during RTL synthesis, and leaves the task of common *logical* subexpression elimination to the subsequent logic optimization.

Example 10.19 Consider the following piece of code.

```
x = p+a*b;
y = q+a*b;
```

We can identify $a*b$ as a common subexpression and replace it with a variable c, as shown below.

```
c = a*b;
x = p+c;
y = q+c;
```

The above transformation allows us to use only one multiplier instead of two. Thus, it can help in saving area.

. **Strength reduction:** Sometimes we can replace an expensive arithmetic operation with an equivalent less expensive operation. This transformation is known as *strength reduction*.

Example 10.20 Consider the following piece of code.

```
x = a*64;
y = b/4;
z = c*17;
```

It consists of expensive multiplication and division operators. We can carry out multiplication by 2 (or by powers of 2) by shifting left. Similarly, we can carry out division by 2 (or by powers of 2) by shifting right. The shifting operation is less expensive than multiplication and division. Therefore, we can transform the code as shown below and save area.

```
x = a<<6;
y = b>>2;
z = (c<<4)+c;
```

Note that we have transformed multiplication by 17 as shift and addition. Typically, such shifting and addition are less expensive than performing multiplication.

4. **Tree height reduction:** We can exploit the parallelism of hardware to improve the speed of computation for long arithmetic expressions.

Assume that we represent a given arithmetic expression as a *binary tree* (*binary* because we typically implement arithmetic operators, such as adder and multiplier, as two-input arithmetic modules). The height of the tree represents the longest chain of operation and indicates the maximum delay in computation. We can employ the commutative and distributive properties of arithmetic operators to restructure the tree such that the height gets reduced. As a result, the maximum delay in computation gets reduced. Note that such restructuring is possible because separate hardware can work in parallel.

Example 10.21 Consider the expression $x=a+b+c+d$. We can perform the computation using two-input adders, as shown in Figure 10.20(a). Note that the computation is performed sequentially by three adders. Hence, the maximum delay in computing x is three adders' delay.

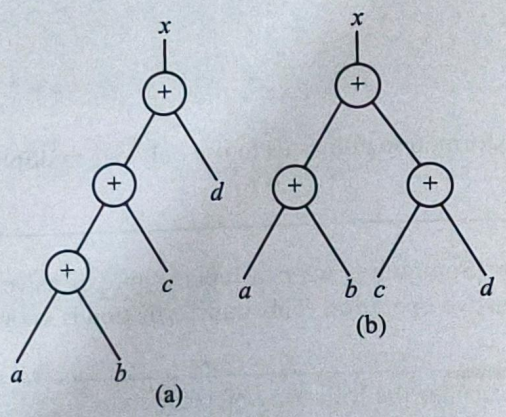

Figure 10.20 Binary tree for the given expression (a) before transformation, (b) after transformation

We can reduce the height of the expression tree as shown in Figure 10.20(b). Note that the number of adders required is the same in both implementations. However, the computations of $(a+b)$ and $(c+d)$ are being performed parallelly. Hence, the maximum delay in computing x reduces to two adders' delay.

5. **Dead code elimination:** We can remove the portion of a code that will never be executed without impacting the design functionality. Similarly, we can remove the portion of a code whose result is never used. We refer to these transformations as *dead code elimination*. Note that a designer never introduces a dead code intentionally. Nevertheless, dead code can appear in an RTL code due to previous transformations. By eliminating dead code, we can save hardware resources.

Example 10.22 Consider the following piece of code.

```
debug = 0;
if (debug == 1) begin // dead code
    x = a; // dead code
end // dead code
```

The RTL synthesis tool can deduce that the value of *debug* is 0 when it reaches the `if` statement. Hence, it can eliminate the `if` statement and everything within it as a dead code.

6. **Copy propagation:** We can sometimes use a copy assignment of the form $x = a$ for optimizing. We can replace the occurrences of the LHS (i.e., x) with the RHS (i.e., a) in the subsequent code, unless the value of a changes. This transformation is known as *copy propagation*.

Example 10.23 Consider the following piece of code.

```
x = a;
y = x+5;
z = x+9;
```

We can transform the code using copy propagation as follows:

```
x = a;
y = a+5;
z = a+9;
```

Now, the copy assignment becomes a dead code because its result is never used, and we can eliminate it as follows:

```
y = a+5;
z = a+9;
```

10.6 FSM SYNTHESIS

During RTL synthesis, FSMs need to be synthesized to hardware. The size and performance of the hardware depend on the number of states, the state encoding scheme, and the implementation of the combinational logic defining the state transitions and the output logic. Initially, an RTL synthesis tool can choose a simple encoding scheme such as one-hot encoding or what exists in the RTL code. In the later stages of logic synthesis, it can perform FSM optimization.[5]

Example 10.24 Consider the state diagram shown in Figure 10.21. There are three states {*S0*, *S1*, *S2*}. The state *S0* is the *initial state* or the *reset state*. The FSM takes 1 bit as input, and the state transitions are defined to detect a nonoverlapping sequence of 101. On detecting the 101 sequence, it produces 1 and goes to the reset state. For all other transitions, it produces 0. We can write the Verilog code for this FSM as shown below.[6] We have used asynchronous reset for the state elements.

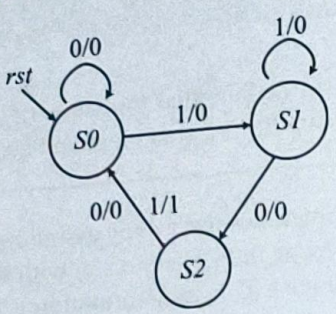

Figure 10.21 State diagram for a 101 nonoverlapping sequence detector

```
module MyFSM (in, clk, rst, out) ;
    input in, clk, rst;
    output out;
    reg out;

    reg [2:0] cS; // state elements of the FSM
    reg [2:0] nS; // holds the next state

    // state encoding (one-hot)
    parameter S0 = 3'b001, S1 = 3'b010, S2 = 3'b100;
```

[5] We will explain FSM optimization in Chapter 12 ("Logic optimization").

[6] The Verilog code consists of initialization (variable declaration and state encoding), combinational logic for state transition, output combinational logic, and state transition. It often improves the readability and debuggability of the Verilog code if we write these parts of an FSM in separate blocks.

```
    // combinational logic for state transition
    always @(*) begin
        case (cS)
            S0: if (in==1'b1) nS = S1; else nS = S0;
            S1: if (in==1'b0) nS = S2; else nS = S1;
            S2: nS = S0;
            default: nS = S0;
        endcase
    end

    // combinational logic for output
    always @(*) begin
        case (cS)
            S0: out = 1'b0;
            S1: out = 1'b0;
            S2: if (in==1'b1) out = 1'b1; else out = 1'b0;
            default: out = 1'b0;
        endcase
    end

    // state transition on clock edges
    always @(posedge clk or posedge rst) begin
        if (rst) cS <= S0;
        else cS <= nS;
    end
endmodule
```

An RTL synthesis tool can infer flip-flops for the state elements *cS[2:0]* using the edge-triggered `always` block. It will synthesize the code to three flip-flops (one for each bit of *cS*), as shown in Figure 10.22. Additionally, the RTL synthesis tool will infer the next state logic using assignments to the next state variable *nS[2:0]* in the first `case` statement. The logic derived using this `case` statement will drive the D-pins of the state element through the signals *nS[2:0]*. For example, we can observe from the statement "S1: if ..." that the FSM reaches the state *S2* (i.e., the next state bit *nS[2]* becomes 1), when the current state is *S1* and *in*=0.

The state *S1* is represented by 3'b010, or equivalently *cS[0]'.cS[1].cS[2]'*. Hence, the following logic will be inferred for *nS[2]*:

$$nS[2] = cS[0]'.cS[1].cS[2]'.in'$$

Similarly, an RTL synthesis tool can infer the output logic using the second `case` statement. We can observe that *out*=1 when the current state is *S2* (represented by 3'b100) and *in*=1. In all other cases, *out*=0. Hence, an RTL synthesis tool will infer the following logic for the output:

$$out = cS[0]'.cS[1]'.cS[2].in$$

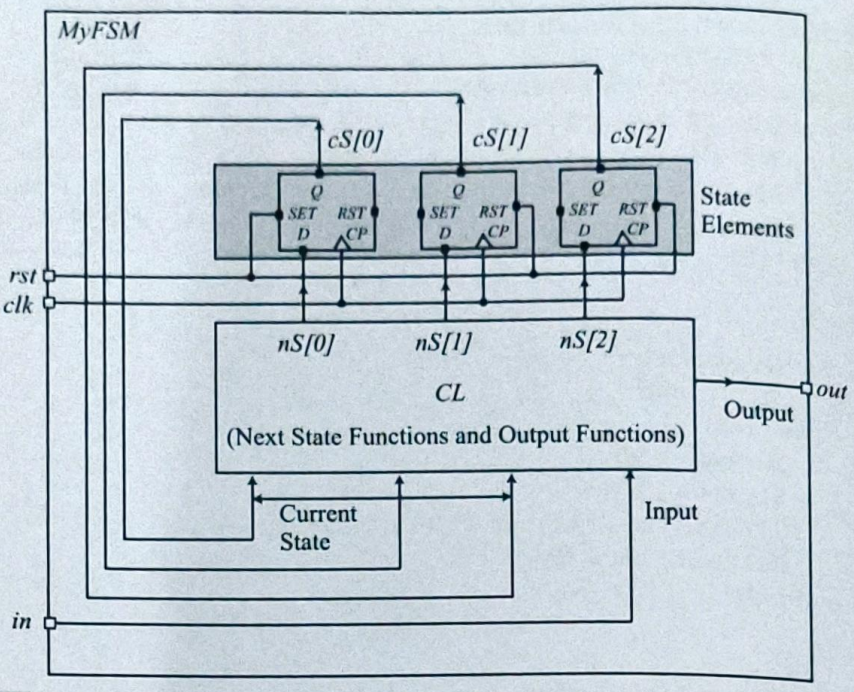

Figure 10.22 Schematic representation of the circuit for the given Verilog code (the next state and output logic functions are shown as combinational logic block *CL*)

An RTL synthesis tool can also possibly recognize the above code as an FSM and carry out FSM-based optimization such as state minimization and re-encoding. However, identifying an FSM in a Verilog code is a challenging problem because there are too many ways to represent an FSM. Therefore, tools use various heuristics to identify FSMs. Note that identifying an FSM is required not only in RTL synthesis, but also for other purposes such as functional verification of RTL using formal methods.

We know that the *next state* of an FSM depends on the *current state*. Additionally, the signals generated by the state elements propagate through the combinational logic and return to them, forming a *loop*. Hence, we can identify state elements by determining such loops in an RTL model [6]. However, such loops can occur even for sub-circuits that are not FSMs. Hence, we need to use heuristics to filter out such false matches.

RTL synthesis tools typically employ some pattern of RTL constructs or coding styles to recognize an FSM. For example, tools can search for Verilog `case` statements in a combinational `always` block and consider the variables compared in that statement as potential state elements [7]. This heuristic will work in the previous example. Additionally, a designer can provide some hints to the tool for recognizing an FSM. Moreover, we can follow a coding style that allows easy identification of an FSM [8].

10.7 RECENT TRENDS

RTL synthesis is a matured technology. Nevertheless, advancements are required to keep pace with the increased design complexity and tighter power, performance, and area (PPA) constraints.

We can take the help of SystemVerilog features to avoid simulation–synthesis mismatches, inadvertent modeling errors, and writing self-documenting codes [9]. For example, we can use explicit constructs such as `always_comb`, `always_latch`, and `always_ff` to avoid modeling errors. Additionally, we can define FSMs using `enum` that allows easy identification of the state elements.

The PPA of a design strongly depends on RTL synthesis. Optimizing data paths involving arithmetic operators is critical for meeting the PPA targets. The techniques such as resource sharing help us in minimizing circuit area. Nevertheless, we need to perform timing-aware data path optimization, especially for designs with long data paths. Typically, synthesis tools provide mechanisms and options to a designer for controlling and guiding their RTL optimizations.

After RTL synthesis, we need to verify the functional equivalence of the RTL and the generated netlist. It can be challenging when the boundaries of data path elements get dissolved or we optimize across data path elements. Also note that a synthesis tool is permitted to *refine* an `x`-valued (don't care) RTL signal to either 0 or 1 in the synthesized netlist. Such refinements also pose challenges because they make the problem more complicated for the equivalence checking tools.

REVIEW QUESTIONS

10.1 How are syntax errors detected in a Verilog code during RTL synthesis?

10.2 What do you understand by elaborating an RTL design? What kinds of errors can be detected by an RTL synthesis tool during elaboration?

10.3 When do we need to carry out uniquification of a design?

10.4 Why is delay specification in Verilog statements ignored by an RTL synthesis tool?

10.5 How are `assign` statements synthesized?

10.6 Why is it necessary that the number of iterations in a `for` loop in a Verilog code be fixed for RTL synthesis?

10.7 When can a latch be inferred in an `always` block?

10.8 How is the synthesis of Verilog `function` performed in an RTL model?

10.9 Why is it easier to carry out arithmetic operator-based optimizations at the RTL than at the gate level?

10.10 Comment on the impact of architecture chosen to implement an arithmetic operator on the performance–area trade-off.

10.11 Comment on the area and timing trade-off in the following optimization techniques:

(a) Resource sharing

(b) Speculation

(c) Common subexpression elimination

(d) Strength reduction

(e) Tree height reduction

10.12 Why is it challenging to extract FSMs from a given Verilog code? How can we write a Verilog RTL model that makes this task easy for a tool?

10.13 What are merits of using `always_comb`, `always_latch`, and `always_ff` constructs over simple `always` in an RTL code?

10.14 Consider the following Verilog code.

```
module MyFSM (in, clk, rst, out) ;
    input in, clk, rst;
    output out;
    reg out;

    reg [1:0] cS;
    reg [1:0] nS;
    parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 3'b11;

    always @(*) begin
        case (cS)
            S0: if (in==1'b1) nS = S1; else nS = S0;
            S1: if (in==1'b0) nS = S2; else nS = S1;
            S2: if (in==1'b1) nS = S3; else nS = S0;
            S3: nS = S0;
            default: nS = S0;
        endcase
    end

    always @(*) begin
        case (cS)
            S0: out = 1'b0;
            S1: out = 1'b0;
            S2: out = 1'b0;
            S3: if (in==1'b0) out = 1'b1; else out = 1'b0;
            default: out = 1'b0;
        endcase
    end

    always @(posedge clk or posedge rst) begin
        if (rst) cS <= S0;
        else cS <= nS;
    end

endmodule
```

Draw the state diagram for the FSM represented by it.

TOOL-BASED ACTIVITY

In this activity, use any logic synthesis tool, including open-source tools such as Yosys [10, 11]. You can use any library, including freely available technology libraries [12].

1. Write a Verilog module containing the code shown in Ex. 10.15. Synthesize it with a relaxed timing constraint. Examine the netlist and analyze how/whether the optimization is performed for the arithmetic operators.
2. Repeat the above activity for the code shown in Ex. 10.19.
3. Write a Verilog code shown in Ex. 10.24. Synthesize with a relaxed timing constraint. Observe the netlist and analyze how are the next state and the output functions implemented in the netlist.

REFERENCES

[1] "IEEE standard Verilog hardware description language." *IEEE Std 1364-2001* (2001), pp. 1–792.

[2] M. Budiu and S. C. Goldstein. "Pegasus: An efficient intermediate representation." Tech. rep., Carnegie-Mellon University Pittsburgh PA School of Computer Science, 2002.

[3] C. E. Cummings et al. "Nonblocking assignments in verilog synthesis, coding styles that kill!" *SNUG (Synopsys Users Group) 2000 User Papers*, 2000.

[4] D. Mills and C. E. Cummings. "RTL coding styles that yield simulation and synthesis mismatches." *SNUG (Synopsys Users Group) 1999 Proceedings* (1999).

[5] A. V. Aho, R. Sethi, and J. D. Ullman. "Compilers, principles, techniques." *Addison Wesley* 7, no. 8 (1986), p. 9.

[6] C.-N. J. Liu and J.-Y. Jou. "An automatic controller extractor for HDL descriptions at the RTL." *IEEE Design & Test of Computers* 17, no. 3 (2000), pp. 72–77.

[7] P. Jamieson and J. Rose. "A verilog RTL synthesis tool for heterogeneous FPGAs." *International Conference on Field Programmable Logic and Applications 2005* (2005), pp. 305–310, IEEE.

[8] T.-H. Wang and T. Edsall. "Practical FSM analysis for Verilog." *Proceedings International Verilog HDL Conference and VHDL International Users Forum* (1998), pp. 52–58, IEEE.

[9] "IEEE standard for SystemVerilog—Unified hardware design, specification, and verification language." *IEEE STD 1800-2009* (2009), pp. 1–1285.

[10] C. Wolf, J. Glaser, and J. Kepler. "Yosys—A free Verilog synthesis suite." *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)* (2013).

[11] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, et al. "Toward an open-source digital flow: First learnings from the OpenROAD project." *Proceedings of the 56th Annual Design Automation Conference 2019* (2019), pp. 1–4.

[12] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen. "Open cell library in 15nm FreePDK technology." *Proceedings of the 2015 Symposium on International Symposium on Physical Design* (2015), pp. 171–178.