Chapter 4: Structural Design and Hierarchy

This chapter describes how to accomplish hierarchy within Verilog using lower-level subsystems. Structural design in Verilog refers to including lower-level subsystems within a higher-level module in order to produce the desired functionality. This is called *hierarchy* and is a good design practice because it enables design partitioning. A purely structural design will not contain any behavioral constructs in the module such as signal assignments, but instead just contain the instantiation and interconnections of other subsystems. A subsystem in Verilog is simply another module that is called by a higher-level module. Each lower-level module that is called is executed concurrently by the calling module.

Learning Outcomes—After completing this chapter, you will be able to:

4.1 Instantiate and map the ports of a lower-level component in Verilog.
4.2 Design a Verilog model for a system that uses hierarchy.

4.1 Structural Design Constructs

4.1.1 Lower-Level Module Instantiation

The term *instantiation* refers to the *use* or *inclusion* of a lower-level module within a system. In Verilog, the syntax for instantiating a lower-level module is as follows.

```
module_name <instance_identifier> (port mapping...);
```

The first portion of the instantiation is the module name that is being called. This must match the lower-level module name exactly, including case. The second portion of the instantiation is an optional instance identifier. Instance identifier are useful when instantiating multiple instances of the same lower-level module. The final portion of the instantiation is the port mapping. There are two techniques to connect signals to the ports of the lower-level module, *explicit* and *positional*.

4.1.2 Port Mapping

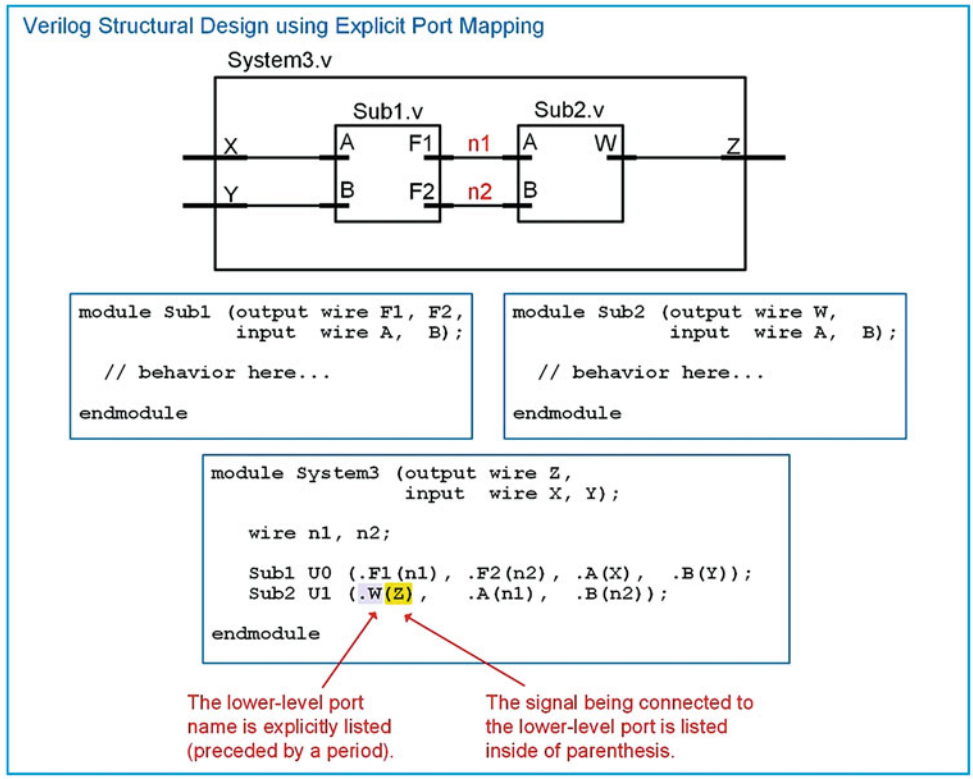
4.1.2.1 Explicit Port Mapping

In explicit port mapping the names of the ports of the lower-level subsystem are provided along with the signals they are being connected to. The lower-level port name is preceded with a period (.) while the signal it is being connected is enclosed within parenthesis. The port connections can be listed in any order since the details of the connection (i.e., port name to signal name) are explicit. Each connection is separated by a comma. The syntax for explicit port mapping is as follows:

```
module_name <instance identifier> (.port_name1(signal1), .port_name2(signal2),
etc.);
```

Example 4.1 shows how to design a Verilog model of a hierarchical system that consists of two lower-level modules.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-030-10552-5_4



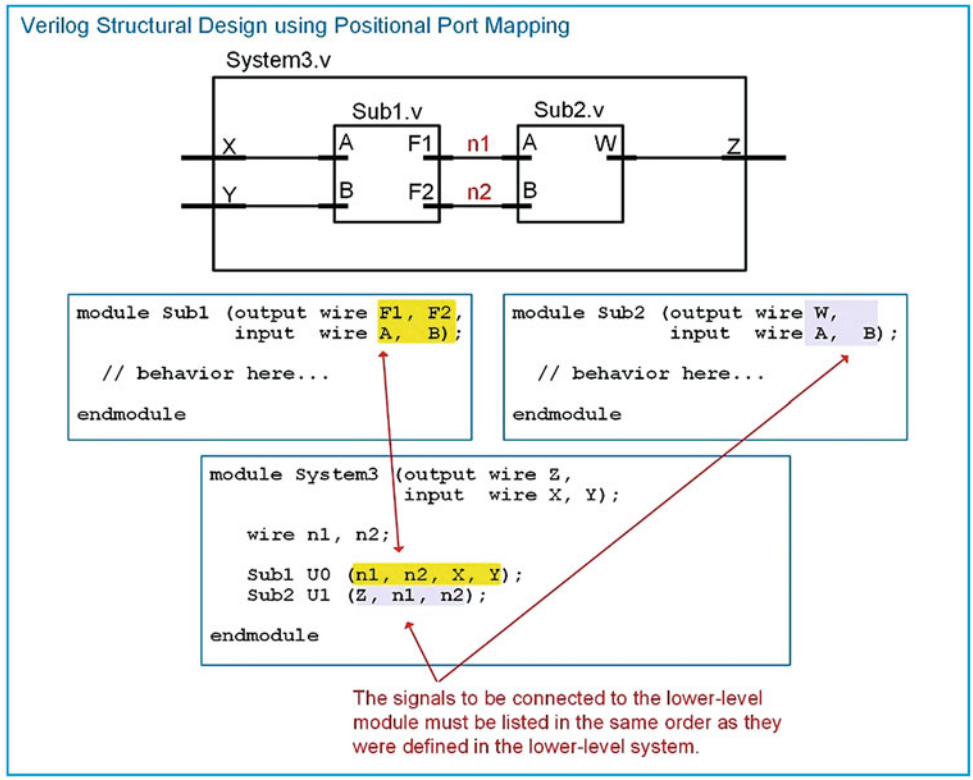
Example 4.1
Verilog structural design using explicit port mapping

4.1.2.2 Positional Port Mapping

In positional port mapping the names of the ports of the lower-level modules are not explicitly listed. Instead, the signals to be connected to the lower-level system are listed in the same order in which the ports were defined in the subsystem. Each signal name is separated by a comma. This approach requires less text to describe the connection but can also lead to misconnections due to inadvertent mistakes in the signal order. The syntax for positional port mapping is as follows:

```
module_name : <instance_identifier> (signal1, signal2, etc.);
```

Example 4.2 shows how to create the same structural Verilog model as in Example 4.1, but using positional port mapping instead.



Example 4.2
Verilog structural design using positional port mapping

4.1.3 Gate-Level Primitives

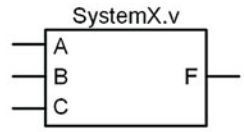
Verilog provides the ability to model basic logic functionality through the use of *primitives*. A primitive is a logic operation that is simple enough that it doesn't require explicit modeling. An example of this behavior can be a basic logic gate or even a truth table. Verilog provides a set of *gate-level primitives* to model simple logic operations. These gate-level primitives are **not()**, **and()**, **nand()**, **or()**, **nor()**, **xor()**, and **xnor()**. Each of these primitives are instantiated as lower-level subsystems with positional port mapping. The port order for each primitive has the output listed first followed by the input(s). The output and each of the inputs are scalars. Gate-level primitives do not need to be explicitly created as they are provided as part of the Verilog standard. One of the benefits of using gate-level primitives is that the number of inputs is easily scaled as each primitive can accommodate an increasing number of inputs automatically. Furthermore, modeling using this approach essentially provides a gate-level netlist, so it represents a very low-level, detailed gate-level implementation that is ready for technology mapping. Example 4.3 shows how to use gate-level primitives to model the behavior of a combinational logic circuit.

Example: Modeling Combinational Logic using Gate Level Primitives

Implement the following truth table using gate level primitives.

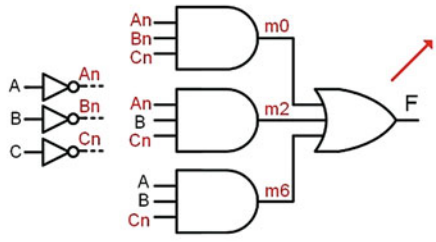
A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Let's call the design SystemX and implement its logic as a canonical SOP logic expression.



$F = \sum_{A,B,C}(0,2,6) = A'\cdot B'\cdot C' + A'\cdot B\cdot C' + A\cdot B\cdot C'$

The corresponding logic diagram is as follows. From this, we can create the Verilog model directly with gate level primitives.



The output is always listed first in the port mapping when using gate level primites.

```
module SystemX  (output wire F,
                 input  wire A, B, C);

   wire  An, Bn, Cn;  // internal nets
   wire  m0, m2, m6;

   not U0 (An, A);            // Not's
   not U1 (Bn, B);
   not U2 (Cn, C);

   and U3 (m0, An, Bn, Cn);   // AND's
   and U4 (m2, An, B,  Cn);
   and U5 (m6, A,  B,  Cn);

   or  U6 (F, m0, m2, m6);    // OR

endmodule
```

Example 4.3
Modeling combinational logic circuits using gate-level primitives

4.1.4 User-Defined Primitives

A **user-defined primitive** (UDP) is a system that describes the behavior of a low-level component using a logic table. This is very useful for creating combinational logic functionality that will be used numerous times. UDPs are also useful for large truth tables where it is more convenient to list the functionality in table form. UDPs are lower-level subsystems that are intended to be instantiated in higher-level modules just like gate-level primitives, with the exception that the UPD needs to be created in its own file. The syntax for a UDP is as follows:

```
primitive primitive_name (output output_name,
                          input input_name1, input_name2, ...);
  table
    in1_val in2_val ... : out_val;
    in1_val in2_val ... : out_val;
                :
  endtable
endprimitive
```

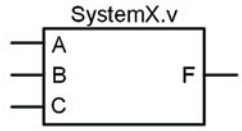
A UDP must list its output(s) first in the port definition. It also does not require types to be defined for the ports. For combinational logic UDPs, all ports are assumed to be of type wire. Example 4.4 shows how to design a user-defined primitive to implement a combinational logic circuit.

Example: Modeling Combinational Logic with a User-Defined Level Primitives

Implement the following truth table with a user-defined primitives.

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Let's call the design SystemX. We will create a simple module for SystemX that defines the ports and then calls the UDP.



The user-defined primitive will be called SystemX_UDP and will contain the table describing the desired functionality.

```
module SystemX (output wire F,
                input  wire A, B, C);

  SystemX_UDP U0 (F, A, B, C);

endmodule
```

The top-level module simply instantiates the UDP.

```
primitive SystemX_UDP (output F,
                       input  A, B, C);

   table
    // A B C : F
       0 0 0 : 1;
       0 0 1 : 0;
       0 1 0 : 1;
       0 1 1 : 0;
       1 0 0 : 0;
       1 0 1 : 0;
       1 1 0 : 1;
       1 1 1 : 0;
   endtable

endprimitive
```

UDPs require that the output be listed first in the port definition.

It is helpful to insert a comment above the table values to list the location of the port names within the table.

Notice that the inputs are listed first, in the order they appear in the port declaration, followed by a ":" and the output.

Example 4.4
Modeling combinational logic circuits with a user-defined primitive

4.1.5 Adding Delay to Primitives

Delay can be added to primitives using the same approach as described in Sect. 3.4. The delay is inserted after the primitive name but before the instance name.

Example:

```
not #2 U0 (An, A);                 // Gate level primitive for an inverter with delay
                                      of 2.
and #3 U3 (m0, An, Bn, Cn);        // Gate level primitive for an AND gate with delay
                                      of 3.
SystemX_UDP #1 U0 (F, A, B, C);    // UDP with a delay of 1.
```

Concept Check

CC4.1 Does the use of lower-level sub-modules model concurrent functionality? Why?

(A) No. Since the lower-level behavior of the module being instantiated may contain non-concurrent behavior, it is not known what functionality will be modeled.

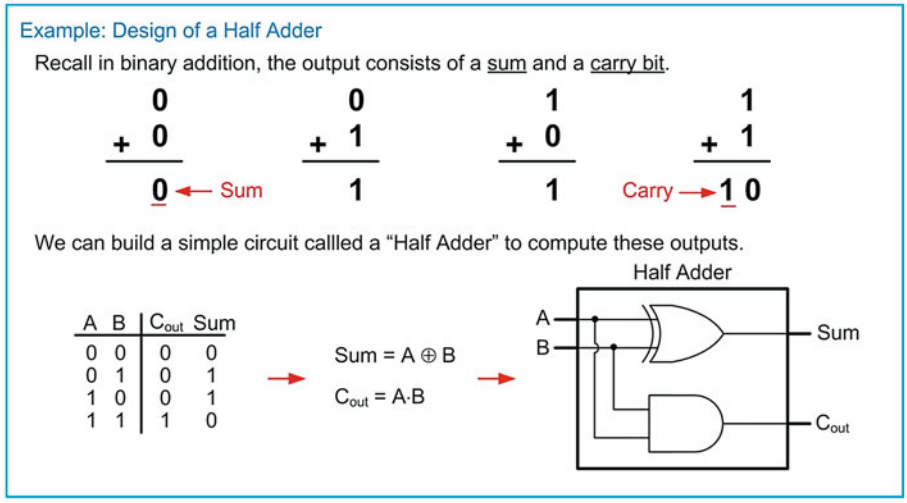
(B) Yes. The modules are treated like independent sub-systems whose behavior runs in parallel just as if separate parts were placed in a design.

4.2 Structural Design Example: Ripple Carry Adder

This section gives an example of a structural design that implements a simple binary adder.

4.2.1 Half Adders

When creating an adder, it is desirable to design incremental subsystems that can be reused. This reduces design effort and minimizes troubleshooting complexity. The most basic component in the adder is called a *half adder*. This circuit computes the sum and carry out on two input arguments. The reason it is called a half adder instead of a full adder is because it does not accommodate a *carry in* during the computation, thus it does not provide all of the necessary functionality required for a positional adder. Example 4.5 shows the design of a half adder. Notice that two combinational logic circuits are required in order to produce the sum (the XOR gate) and the carry out (the AND gate). These two gates are in parallel to each other; thus, the delay through the half adder is due to only one level of logic.

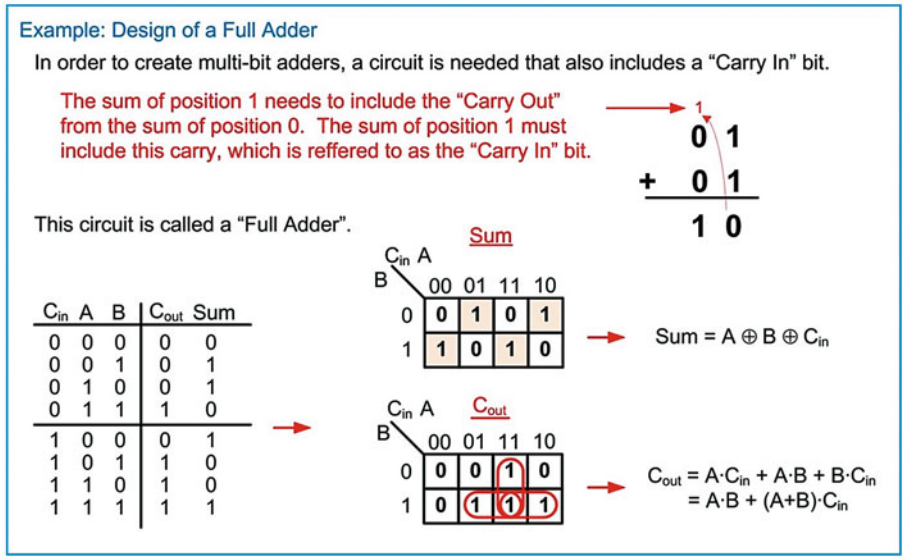


Example 4.5
Design of a half adder

4.2.2 Full Adders

A full adder is a circuit that still produces a sum and carry out, but considers three inputs in the computations (A, B, and C_{in}). Example 4.6 shows the design of a full adder using the classical design

approach. This step is shown to illustrate why it is possible to reuse half adders to create the full adder. In order to do this, it is necessary to have the minimal sum of products logic expression.



Example 4.6
Design of a full adder

As mentioned before, it is desirable to reuse design components as we construct more complex systems. One such design reuse approach is to create a full adder using two half adders. This is straightforward for the sum output since the logic is simply two cascaded XOR gates (Sum = $A \oplus B \oplus C_{in}$). The carry out is not as straightforward. Notice that the expression for C_{out} derived in Example 4.6 contains the term (A + B). If this term could be manipulated to use an XOR gate instead, it would allow the full adder to take advantage of existing circuitry in the system. Figure 4.1 shows a derivation of an equivalency that allows (A + B) to be replaced with ($A \oplus B$) in the C_{out} logic expression.

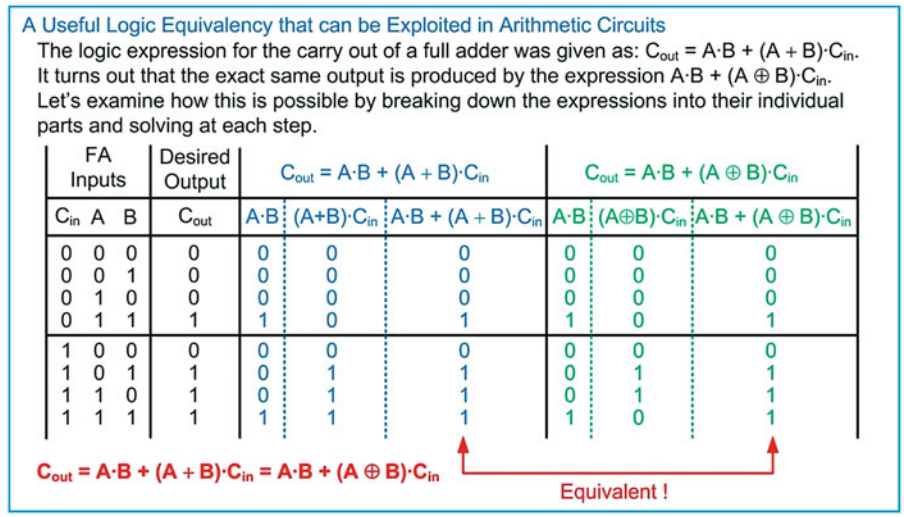
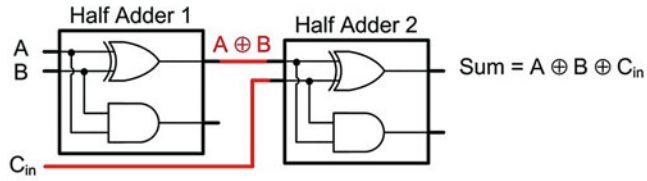


Fig. 4.1
A Useful logic equivalency that can be exploited in arithmetic circuits

The ability to implement the carry out logic using the expression $C_{out} = A{\cdot}B + (A \oplus B){\cdot}C_{in}$ allows us to implement a full adder with two half adders and the addition of a single OR gate. Example 4.7 shows this approach. In this new configuration, the sum is produced in two levels of logic while the carry out is produced in three levels of logic.

Example – Design of a Full Adder Out of Two Half Adders

It is often desirable to create a full adder out of two half adders in order to re-use existing design components. The "Sum" of the full adder can be created by using two cascaded XOR gates provided by the half adders.



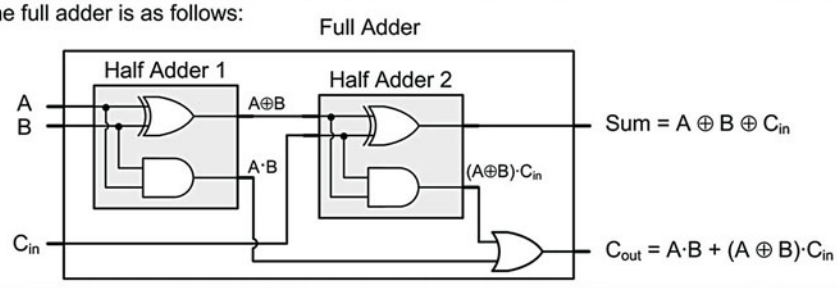
The expression for the "Carry Out" of the full adder is:

$$C_{out} = A{\cdot}B + (A + B){\cdot}C_{in}$$

or

$$C_{out} = A{\cdot}B + (A \oplus B){\cdot}C_{in}$$

Notice that the carry out of Half Adder 1 produces the A·B term in this expression. Also notice that the carry out of Half Adder 2 produces the $(A \oplus B){\cdot}C_{in}$ term. The only remaining logic needed to create the carry out of the full adder is an OR gate. The final logic diagram for the full adder is as follows:



Example 4.7
Design of a full adder out of half adders

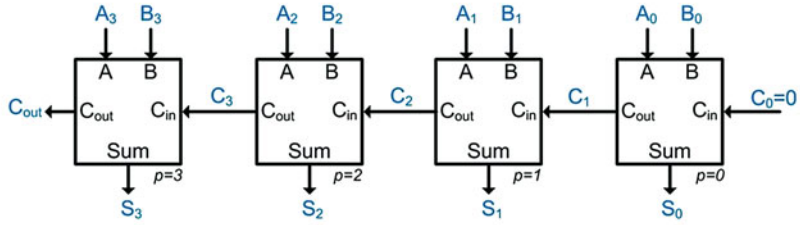
4.2.3 Ripple Carry Adder (RCA)

The full adder can now be used in the creation of multi-bit adders. The simplest topology exploiting the full adder is called a *ripple carry adder* (RCA). In this approach, full adders are used to create the sum and carry out of each bit position. The carry out of each full adder is used as the carry in for the next higher position. Since each subsequent full adder needs to wait for the carry to be produced by the preceding stage, the carry is said to *ripple* through the circuit, thus giving this approach its name.

Example 4.8 shows how to design a 4-bit ripple carry adder using a chain of full adders. Notice that the carry in for the full adder in position 0 is tied to a logic 0. The 0 input has no impact on the result of the sum but enables a full adder to be used in the 0th position.

Example: Design of a 4-Bit Ripple Carry Adder (RCA)

Full adders can be cascaded together to form a multi-bit adder. The symbols are typically drawn in the following fashion to mirror a positional number system.



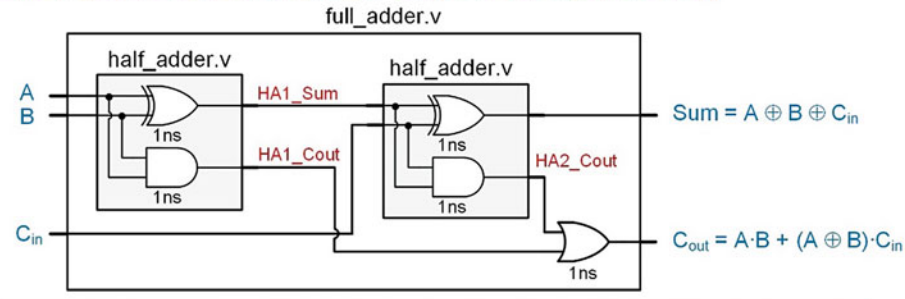
The sum of position 1 cannot complete until it receives the carry in (C_1) from the sum in position 0. The position 2 sum cannot complete until it receives the carry in (C_2) from the sum in position 1, etc. In this way, the carry "ripples" through the circuit from right to left. This configuration is known as a Ripple Carry Adder (RCA).

Example 4.8
Design of a 4-bit ripple carry adder (RCA)

4.2.4 Structural Model of a Ripple Carry Adder in Verilog

Now that the hierarchical design of the RCA is complete, we can now model it in Verilog as a system of lower-level modules. Example 4.9 shows the structural model for a full adder in Verilog consisting of two half adders. The full adder is created by instantiating two versions of the half adder as subsystems. The half adder in this example is implemented using gate-level primitives. In this example, all gates are modeled with a delay of 1 ns.

Example: Structural Model of a Full Adder Using Two Half Adders in Verilog




```
`timescale 1ns/1ps

module half_adder (output wire Sum, Cout,
                   input  wire A, B);

   xor  #1 U1 (Sum, A, B);
   and  #1 U2 (Cout, A, B);

endmodule
```

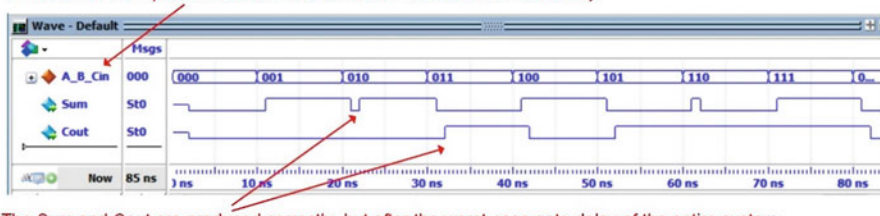
Gate level primitives with delay are used to build the half adder.

```
`timescale 1ns/1ps

module full_adder (output wire Sum, Cout,
                   input  wire A, B, Cin);

  wire HA1_Sum, HA1_Cout, HA2_Cout;

  half_adder U1 (.Sum(HA1_Sum), .Cout(HA1_Cout), .A(A),        .B(B));
  half_adder U2 (.Sum(Sum),     .Cout(HA2_Cout), .A(HA1_Sum), .B(Cin));

  or  #1 U3 (Cout, HA2_Cout, HA1_Cout);

endmodule
```

Two half adders are instantiated in the full adder.

One additional gate level primitive is needed to complete the full adder.

A vector for the inputs is created in the simulation waveform for readability.

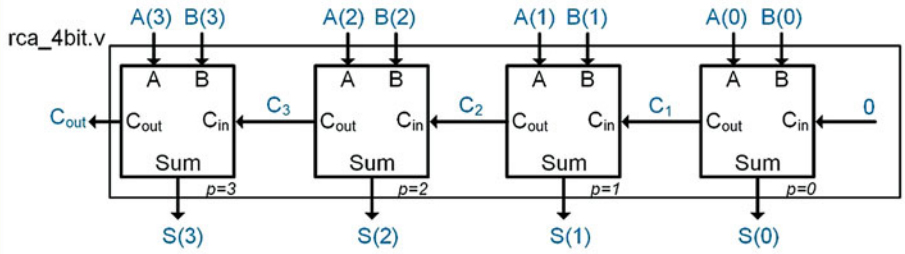


The Sum and Cout are produced correctly, but after the worst-case gate delay of the entire system.

Example 4.9
Structural model of a full adder using two half adders

Example 4.10 shows the structural model of a 4-bit ripple carry adder in Verilog. The RCA is created by instantiating four full adders. Notice that a logic 1'b0 can be directly inserted into the port map of the first full adder to model the behavior of $C_0 = 0$.

Example: Structural Model of a 4-Bit Ripple Carry Adder in Verilog




```
`timescale 1ns/1ps

module rca_4bit (output wire [3:0] Sum,
                 output wire       Cout,
                 input  wire [3:0] A, B);

  wire C1, C2, C3;

  full_adder U1 (.Sum(Sum[0]), .Cout(C1),   .A(A[0]), .B(B[0]), .Cin(1'b0));
  full_adder U2 (.Sum(Sum[1]), .Cout(C2),   .A(A[1]), .B(B[1]), .Cin(C1));
  full_adder U3 (.Sum(Sum[2]), .Cout(C3),   .A(A[2]), .B(B[2]), .Cin(C2));
  full_adder U4 (.Sum(Sum[3]), .Cout(Cout), .A(A[3]), .B(B[3]), .Cin(C3));

endmodule
```

A fixed value can be inserted into the port map of a sub-system. This handles the Cin port for the first full_adder.

Example 4.10
Structural model of a 4-bit ripple carry adder in Verilog

Concept Check

CC4.2 Why is the use of hierarchy considered a good design practice?

(A) Hierarchy allows the design to be broken into smaller pieces, each with simpler functionality that can be verified independently prior to being used in a higher-level system.

(B) Hierarchy allows a large system to be broken into smaller subsystems that can be designed by multiple engineers, thus decreasing the overall development time.

(C) Hierarchy allows a large system to be broken down into smaller subsystems that can be more easily understood so that debugging is more manageable.

(D) All of the above.

Summary

- Instantiating other modules from within a higher-level module is how Verilog implements hierarchy. A lower-level module can be instantiated as many times as desired. An instance identifier is useful is keeping track of each instantiation.
- The ports of the component can be connected using either *explicit* or *positional port mapping*.
- Verilog subsystems are also treated as concurrent subsystems.
- Gate-level primitives are provided in Verilog to implement basic logic functions (not, and, nand, or, nor, xor, xnor). These primitives are instantiated just like any other lower-level subsystem.
- User-Defined Primitives are supported in Verilog that allow the functionality of a circuit to be described in table form.

Exercise Problems

Section 4.1: Structural Design Constructs

4.1.1 How many times can a lower-level module be instantiated?

4.1.2 Which port mapping technique is more compact, explicit or positional?

4.1.3 Which port mapping technique is less prone to connection errors because the names of the lower-level ports are listed within the mapping?

4.1.4 Would it make sense to design a lower-level module to implement an AND gate in Verilog?

4.1.5 When would it makes more sense to build a user-defined primitive instead of modeling the logic using continuous assignments?

Section 4.2: Structural Design Examples

4.2.1 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 4.2. Use a structural design approach based on gate-level primitives. This is considered *structural* because you will need to instantiate the gate-level primitives just like a traditional subsystem; however, you don't need to create the gate-level modules as they are already built into the Verilog standard. You will need to determine a logic expression for the system prior to connecting the gate-level primitives. You can use whatever approach you prefer to create the logic expression (i.e., canonical SOP/POS, minimized SOP/POS). Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

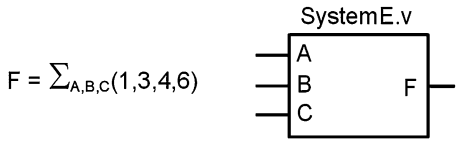


Fig. 4.2
System E Functionality

4.2.2 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 4.2. Use a structural design approach based on a user-defined primitive. This is considered *structural* because you will need to instantiate the user-defined primitive just like a traditional subsystem. You will need to create both the upper level module and the lower-level UDP. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

4.2.3 Design a Verilog model to implement the behavior described by the 3-input maxterm list shown in Fig. 4.3. Use a structural design approach based on gate-level primitives. This is considered *structural* because you will need to instantiate the gate-level primitives just like a traditional subsystem; however, you don't need to create the gate-level modules as they are already built into the Verilog standard. You will need to determine a logic expression for the system prior to connecting the gate-level primitives. You can use whatever approach you prefer to create the logic expression (i.e., canonical SOP/POS, minimized SOP/POS). Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

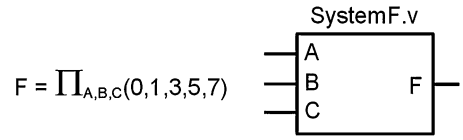


Fig. 4.3
System F Functionality

4.2.4 Design a Verilog model to implement the behavior described by the 3-input maxterm list shown in Fig. 4.3. Use a structural design approach based on a user-defined primitive. This is considered *structural* because you will need to instantiate the user-defined primitive just like a traditional subsystem. You will need to create both the upper level module and the lower-level UDP. Declare your module and ports to match the block diagram provided. Use the type wire for your ports. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

4.2.5 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 4.4. Use a structural design approach based on gate-level primitives. This is considered *structural* because you will need to instantiate the gate-level primitives just like a traditional subsystem; however, you don't need to create the gate-level modules as they are already built into the Verilog standard. You will need to determine a logic expression for the system prior to connecting the gate-level primitives. You can use whatever approach you prefer to create the logic expression (i.e., canonical SOP/POS, minimized SOP/POS). Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

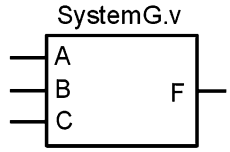


Fig. 4.4
System G Functionality

4.2.6 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 4.4. Use a structural design approach based on a user-defined primitive. This is considered *structural* because you will need to instantiate the user-defined primitive just like a traditional subsystem. You will need to create both the upper level module and the lower-level UDP. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

4.2.7 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 4.5. Use a structural design approach based on gate-level primitives. This is considered *structural* because you will need to instantiate the gate-level primitives just like a traditional subsystem; however, you don't need to create the gate-level modules as they are already built into the Verilog standard. You will need to determine a logic expression for the system prior to connecting the gate-level primitives. You can use whatever approach you prefer to create the logic expression (i.e., canonical SOP/POS, minimized SOP/POS). Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

$F = \sum_{A,B,C,D}(1,3,9,11)$

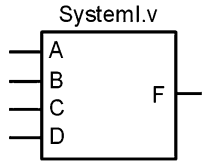


Fig. 4.5
System I Functionality

4.2.8 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 4.5. Use a structural design approach based on a user-defined primitive. This is considered *structural* because you will need to instantiate the user-defined primitive just like a traditional subsystem. You will need to create both the upper level module and the lower-level UDP. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

4.2.9 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 4.6. Use a structural design approach based on gate-level primitives. This is considered *structural* because you will need to instantiate the gate-level primitives just like a traditional subsystem; however, you don't need to create the gate-level modules as they are already built into the Verilog standard. You will need to determine a logic expression for the system prior to connecting the gate-level primitives. You can use whatever approach you prefer to create the logic expression (i.e., canonical SOP/POS, minimized SOP/POS). Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

$F = \prod_{A,B,C,D}(0,1,2,3,6,8,9,10,11,14)$

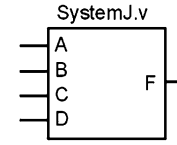


Fig. 4.6
System J Functionality

4.2.10 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 4.6. Use a structural design approach based on a user-defined primitive. This is considered *structural* because you will need to instantiate the user-defined primitive just like a traditional subsystem. You will need to create both the upper level module and the lower-level UDP. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

4.2.11 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 4.7. Use a structural design approach based on gate-level primitives. This is considered *structural* because you will need to instantiate the gate-level primitives just like a traditional subsystem; however, you don't need to create the gate-level modules as they are already built into the Verilog standard. You will need to determine a logic expression for the system prior to connecting the gate-level primitives. You can use whatever approach you prefer to create the logic expression (i.e., canonical SOP/POS, minimized SOP/POS).

Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

A	B	C	D	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

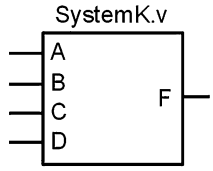


Fig. 4.7
System K Functionality

4.2.12 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 4.7. Use a structural design approach based on a user-defined primitive. This is considered *structural* because you will need to instantiate the user-defined primitive just like a traditional subsystem. You will need to create both the upper level module and the lower-level UDP. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.