Chapter 3: Modeling Concurrent Functionality in Verilog

This chapter presents a set of built-in operators that will allow basic logic expressions to be modeled within a Verilog module. This chapter then presents a series of combinational logic model examples.

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

3.1 Describe the various built-in operators within Verilog.
3.2 Design a Verilog model for a combinational logic circuit using continuous assignment and logical operators.
3.3 Design a Verilog model for a combinational logic circuit using continuous assignment and conditional operators.
3.4 Design a Verilog model for a combinational logic circuit using continuous assignment with delay.

3.1 Verilog Operators

There are a variety of predefined operators in the Verilog standard. It is important to note that operators are defined to work on specific data types and that not all operators are synthesizable.

3.1.1 Assignment Operator

Verilog uses the equal sign (=) to denote an assignment. The left-hand side (LHS) of the assignment is the target signal. The right-hand side (RHS) contains the input arguments and can contain both signals, constants, and operators.

Example:

```
F1 = A;       // F1 is assigned the signal A
F2 = 8'hAA;   // F2 is an 8-bit vector and is assigned the value 10101010₂
```

3.1.2 Continuous Assignment

Verilog uses the keyword **assign** to denote a continuous signal assignment. After this keyword, an assignment is made using the = symbol. The left-hand side (LHS) of the assignment is the target signal and must be a net type. The right-hand side (RHS) contains the input arguments and can contain nets, regs, constants, and operators. A continuous assignment models combinational logic. Any change to the RHS of the expression will result in an update to the LHS target net. The net being assigned to must be declared prior to the first continuous assignment. Multiple continuous assignments can be made to the same net. When this happens, the assignment containing signals with the highest drive strength will take priority.

Example:

```
assign F1 = A;       // F1 is updated anytime A changes, where A is a signal
assign F2 = 1'b0;    // F2 is assigned the value 0
assign F3 = 4'hAA;   // F3 is an 8-bit vector and is assigned the value 10101010₂
```

Each individual assignment will be executed concurrently and synthesized as separate logic circuits. Consider the following example.

B. J. LaMeres, *Quick Start Guide to Verilog*, https://doi.org/10.1007/978-3-030-10552-5_3

Example:

```
assign X = A;
assign Y = B;
assign Z = C;
```

When simulated, these three lines of Verilog will make three separate signal assignments at the exact same time. This is different from a programming language that will first assign A to X, then B to Y, and finally C to Z. In Verilog this functionality is identical to three separate wires. This description will be directly synthesized into three separate wires.

Below is another example of how continuous signal assignments in Verilog differ from a sequentially executed programming language.

Example:

```
assign A = B;
assign B = C;
```

In a Verilog simulation, the signal assignments of C to B and B to A will take place at the same time. This means during synthesis, the signal B will be eliminated from the design since this functionality describes two wires in series. Automated synthesis tools will eliminate this unnecessary signal name. This is not the same functionality that would result if this example was implemented as a sequentially executed computer program. A computer program would execute the assignment of B to A first, then assign the value of C to B second. In this way, B represents a storage element that is passed to A before it is updated with C.

3.1.3 Bitwise Logical Operators

Bitwise operators perform logic functions on individual bits. The inputs to the operation are single bits and the output is a single bit. In the case where the inputs are vectors, each bit in the first vector is operated on by the bit in the same position from the second vector. If the vectors are not the same length, the shorter vector is padded with leading zeros to make both lengths equal. Verilog contains the following bitwise operators:

Syntax	Operation
~	Negation
&	AND
|	OR
^	XOR
~^ or ^~	XNOR
<<	Logical shift left (fill empty LSB location with zero)
>>	Logical shift right (fill empty MSB location with zero)

Example:

```
~X          // invert each bit in X
X & Y       // AND each bit of X with each bit of Y
X | Y       // OR each bit of X with each bit of Y
X ^ Y       // XOR each bit of X with each bit of Y
X ~^ Y      // XNOR each bit of X with each bit of Y
X << 3      // Shift X left 3 times and fill with zeros
Y >> 2      // Shift Y right 2 times and fill with zeros
```

3.1.4 Reduction Logic Operators

A *reduction* operator is one that uses each bit of a vector as individual inputs into a logic operation and produces a single-bit output. Verilog contains the following reduction logic operators.

Syntax	Operation
&	AND all bits in the vector together (1-bit result)
~&	NAND all bits in the vector together (1-bit result)
|	OR all bits in the vector together (1-bit result)
~|	NOR all bits in the vector together (1-bit result)
^	XOR all bits in the vector together (1-bit result)
~^ or **^~**	XNOR all bits in the vector together (1-bit result)

Example:

```
&X      // AND all bits in vector X together
~&X     // NAND all bits in vector X together
|X      // OR all bits in vector X together
~|X     // NOR all bits in vector X together
^X      // XOR all bits in vector X together
~^X     // XNOR all bits in vector X together
```

3.1.5 Boolean Logic Operators

A Boolean logic operator is one that returns a value of TRUE (1) or FALSE (0) based on a logic operation of the input operations. These operations are used in decision statements.

Syntax	Operation
!	Negation
&&	AND
||	OR

Example:

```
!X        // TRUE if all values in X are 0, FALSE otherwise
X && Y    // TRUE if the bitwise AND of X and Y results in all ones, FALSE otherwise
X || Y    // TRUE if the bitwise OR of X and Y results in all ones, FALSE otherwise
```

3.1.6 Relational Operators

A relational operator is one that returns a value of TRUE (1) or FALSE (0) based on a comparison of two inputs.

Syntax	Description
==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Example:

```
X == Y     // TRUE if X is equal to Y, FALSE otherwise
X != Y     // TRUE if X is not equal to Y, FALSE otherwise
X < Y      // TRUE if X is less than Y, FALSE otherwise
X > Y      // TRUE if X is greater than Y, FALSE otherwise
X <= Y     // TRUE if X is less than or equal to Y, FALSE otherwise
X >= Y     // TRUE if X is greater than or equal to Y, FALSE otherwise
```

3.1.7 Conditional Operators

Verilog contains a conditional operator that can be used to provide a more intuitive approach to modeling logic statements. The keyword for the conditional operator is **?** with the following syntax:

```
<target_net> = <Boolean_condition> ? <true_assignment> : <false_assignment>;
```

This operator specifies a Boolean condition in which if evaluated TRUE, the *true_assignment* will be assigned to the target. If the Boolean condition is evaluated FALSE, the *false_assignment* portion of the operator will be assigned to the target. The values in this assignment can be signals or logic values. The Boolean condition can be any combination of the Boolean operators described above. Nested conditional operators can also be implemented by inserting subsequent conditional operators in place of the *false_value*.

Example:

```
F = (A == 1'b0) ? 1'b1 : 1'b0;                  // If A is a zero, F=1, otherwise F=0.
                                                //    This models an inverter.

F = (sel == 1'b0) ? A : B;                      // If sel is a zero, F=A, otherwise F=B.
                                                //    This models a selectable switch.

F = ((A == 1'b0) && (B == 1'b0)) ? 1'b'0 :      // Nested conditional statements.
    ((A == 1'b0) && (B == 1'b1)) ? 1'b'1 :      //    This models an XOR gate.
    ((A == 1'b1) && (B == 1'b0)) ? 1'b'1 :
    ((A == 1'b1) && (B == 1'b1)) ? 1'b'0;

F = ( !C && (!A || B) ) ? 1'b1 : 1'b0;          // This models the logic expression
                                                //    F = C'·(A'+B).
```

3.1.8 Concatenation Operator

In Verilog, the curly brackets (i.e., **{}**) are used to concatenate multiple signals. The target of this operation must be the same size of the sum of the sizes of the input arguments.

Example:

```
Bus1[7:0] = {Bus2[7:4], Bus3[3:0]};  // Assuming Bus1, Bus2, and Bus3 are all 8-bit
                                     // vectors, this operation takes the upper
                                        4-bits of
                                     // Bus2, concatenates them with the lower
                                        4-bits of
                                     // Bus3, and assigns the 8-bit combination
                                        to Bus1.

BusC = {BusA, BusB};                 // If BusA and BusB are 4-bits, then BusC
                                     //  must be 8-bits.

BusC[7:0] = {4'b0000, BusA};         // This pads the 4-bit vector BusA with
                                        4x leading
                                     // zeros and assigns to the 8-bit vector BusC.
```

3.1.9 Replication Operator

Verilog provides the ability to concatenate a vector with itself through the *replication operator*. This operator uses double curly brackets (i.e., **{{}}**) and an integer indicating the number of replications to be performed. The replication syntax is as follows:

```
{<number_of_replications>{<vector_name_to_be_replicated>}}
```

Example:

```
BusX = {4{Bus1}};          // This is equivalent to: BusX = {Bus1, Bus1, Bus1, Bus1};
BusY = {2{A,B}};           // This is equivalent to: BusY = {A, B, A, B};
BusZ = {Bus1, {2{Bus2}}}; // This is equivalent to: BusZ = {Bus1, Bus2, Bus2};
```

3.1.10 Numerical Operators

Verilog also provides a set of numerical operators as follows:

Syntax	Operation
+	Addition
–	Subtraction (when placed between arguments)
–	2's complement negation (when placed in front of an argument)
*****	Multiplication
/	Division
%	Modulus
**	Raise to the power
<<<	Shift to the left, fill with zeros
<<<	Shift to the right, fill with sign bit

Example:

```
X + Y      // Add X to Y
X - Y      // Subtract Y from X
-X         // Take the two's complement negation of X
X * Y      // Multiply X by Y
X / Y      // Divide X by Y
X % Y      // Modulus X/Y
X ** Y     // Raise X to the power of Y
X <<< 3    // Shift X left 3 times, fill with zeros
X >>> 2    // Shift X right 2 times, fill with sign bit
```

Verilog will allow the use of these operators on arguments of different sizes, types, and signs. The rules of the operations are as follows:

- If two vectors are of different sizes, the smaller vector is expanded to the size of the larger vector.
  - If the smaller vector is unsigned, it is padded with zeros.
  - If the smaller vector is signed, it is padded with the sign bit.
- If one of the arguments is real, then the arithmetic will take place using real numbers.
- If one of the arguments is unsigned, then all arguments will be treated as unsigned.

Example 3.1 shows the behavioral model for a 4-bit adder in Verilog using a combination of operators including continuous assignment, numerical addition, and concatenation. Note that when adding two *n*-bit arguments the sum produced will be *n* + 1 bits. This can be handled in Verilog by concatenating the Cout and Sum outputs on the LHS of the assignment. The entire add operation can be

accomplished in a single continuous assignment that contains both the concatenation and addition operators. When using continuous assignment, the LHS must be a net data type. This means the outputs Cout and Sum need to be declared as type wire.

Example: Behavioral Model of a 4-Bit Adder in Verilog

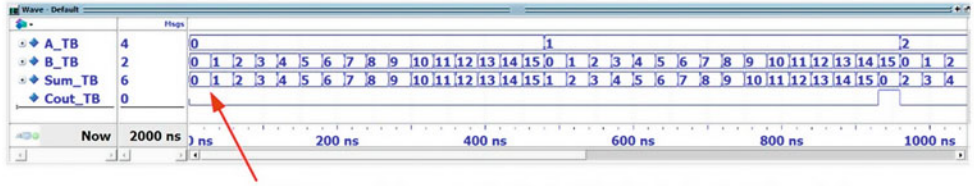
```
module adder_4bit (output wire [3:0] Sum,
                   output wire       Cout,
                   input  wire [3:0] A, B);

   assign {Cout, Sum} = A + B;

endmodule
```

When using continuous assignment, the LHS needs to be a net data type.

The addition of two 4-bit numbers will result in a 5-bit sum. Cout and Sum are concatenated on the RHS of the assignment to accomidate 5-bits.



Since no delay was included in the behavioral model, the outputs are produced instantaneously.

Example 3.1
Behavioral model of a 4-bit adder in Verilog

3.1.11 Operator Precedence

The following is the order of precedence of the Verilog operators. If two operators of the same type appear in an expression without parenthesis to dictate the order of precedence, the precedence will be determined by executing from the operations from left to right.

Operators	Precedence	Notes
! ~ + –	Highest	Bitwise/Unary
{} {{}}		Concatenation/Replication
()	↓	No operation, just parenthesis
**		Power
* / %		Binary Multiply/Divide/Modulo
+ –	↓	Binary Addition/Subtraction
<< >> <<< >>>		Shift Operators
< <= > >=		Greater/Less than Comparisons
== !=	↓	Equality/Inequality Comparisons
& ~&		AND/NAND Operators
^ ~^		XOR/XNOR Operators
| ~|	↓	OR/NOR Operators
&&		Boolean AND
||		Boolean OR
?:	Lowest	Conditional Operator

Concept Check

CC3.1 For the expression: F = !A & (B I !C); What is the order of execution of the bitwise operations?

(A) Negate → OR → AND

(B) Negate → AND → OR

(C) OR → Negate → AND

(D) OR → AND → Negate

3.2 Continuous Assignment with Logical Operators

When modeling synthesizable logic, it is important to remember that Verilog is a hardware description language, not a programming language. In a programming language, the lines of code are executed sequentially as they appear in the source file. In Verilog, the lines of code represent the behavior of real hardware. Thus, the assignments are executed concurrently unless specifically noted otherwise. Each of the bitwise logical operators described in Sect. 3.1.3 can be used in conjunction with continuous signal assignments to create individual combinational logic circuits.

3.2.1 Logical Operator Example: SOP Circuit

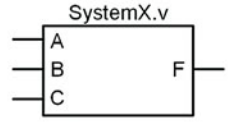
Example 3.2 shows how to design a Verilog model of a combinational logic circuit using continuous assignment and logical operators. Note that in this example the logic expressions must first be determined by hand prior to modeling in Verilog.

Example: Modeling Combinational Logic using Continuous Assignment with Logical Operators

Implement the following truth table using continuous assignment with logical operators.

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

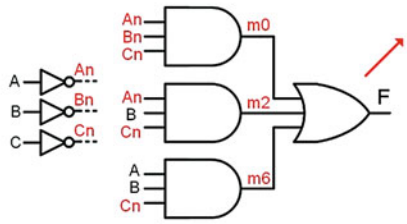
Let's call the module *SystemX*. First, let's declare the ports. The module will have three inputs (A, B, C) and one output (F). We'll use the type wire for all inputs/outputs so that this will synthesize directly into real circuitry.



Now we can design the behavior. We will create a canonical sum of products logic expression for this truth table using minterms.

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

Drawing out the logic diagram will help us understand which internal signals need to be declared for the interim connections. Since there is a need for the complement of each of the inputs, the first set of logic will be three inverters. We'll need to create three wires to hold the inverted versions of the inputs. Let's call them An, Bn and Cn. We'll also need three wires to hold the outputs of the AND gates. Let's call them m0, m2 and m6. Using these internal wires, the port names, and logical operators, we can describe the behavior of the logic expression above.




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

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

   assign An = ~A;             // Not's
   assign Bn = ~B;
   assign Cn = ~C;

   assign m0 = An & Bn & Cn;  // AND's
   assign m2 = An & B  & Cn;
   assign m6 = A  & B  & Cn;

   assign F  = m0 | m2 | m6;  // OR

endmodule
```

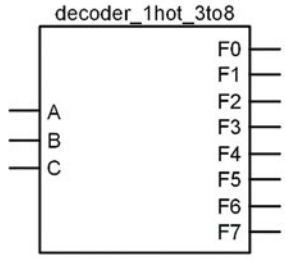
Example 3.2
Combinational logic using continuous assignment with logical operators

3.2.2 Logical Operator Example: One-Hot Decoder

A one-hot decoder is a circuit that has n inputs and 2^n outputs. Each output will assert for one and only one input code. Since there are 2^n outputs, there will always be one and only one output asserted at any given time. Example 3.3 shows how to model a 3-to-8 one-hot decoder in Verilog with continuous assignment and logic operators.

Example: 3-to-8 One-Hot Decoder – Verilog Modeling using Logical Operators

The block diagram and truth table for this system are as follows:



A	B	C	F7	F6	F5	F4	F3	F2	F1	F0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

To implement this in Verilog using logical operators, we must first determine the logic that will be used in the continuous assignment. Again, since each logic function only has one input code corresponding to an output of '1', the minterm can be used to implement the logic.

$F0 = \sum_{A,B,C}(0) = A'\cdot B'\cdot C'$ $\quad F4 = \sum_{A,B,C}(4) = A\cdot B'\cdot C'$

$F1 = \sum_{A,B,C}(1) = A'\cdot B'\cdot C$ $\quad F5 = \sum_{A,B,C}(5) = A\cdot B'\cdot C$

$F2 = \sum_{A,B,C}(2) = A'\cdot B\cdot C'$ $\quad F6 = \sum_{A,B,C}(6) = A\cdot B\cdot C'$

$F3 = \sum_{A,B,C}(3) = A'\cdot B\cdot C$ $\quad F7 = \sum_{A,B,C}(7) = A\cdot B\cdot C$

In Verilog, each of the outputs requires a separate continuous assignment.

```
module decoder_1hot_3to8
  (output wire F0, F1, F2, F3, F4, F5, F6, F7,
   input  wire A, B, C);

  assign F0 = ~A & ~B & ~C;
  assign F1 = ~A & ~B &  C;
  assign F2 = ~A &  B & ~C;
  assign F3 = ~A &  B &  C;
  assign F4 =  A & ~B & ~C;
  assign F5 =  A & ~B &  C;
  assign F6 =  A &  B & ~C;
  assign F7 =  A &  B &  C;

endmodule
```

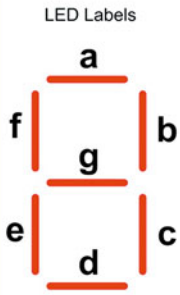
Example 3.3
3-to-8 One-hot decoder—Verilog modeling using logical operators

3.2.3 Logical Operator Example: 7-Segment Display Decoder

A 7-segment display decoder is a circuit used to drive character displays that are commonly found in applications such as digital clocks and household appliances. A character display is made up of seven individual LEDs, typically labeled a–g. The input to the decoder is the binary equivalent of the decimal or Hex character that is to be displayed. The output of the decoder is the arrangement of LEDs that will form the character. Decoders with 2-inputs can drive characters "0" to "3." Decoders with 3-inputs can drive characters "0" to "7." Decoders with 4-inputs can drive characters "0" to "F" with the case of the Hex characters being "A, b, c or C, d, E and F."

Let's look at an example of how to design a 3-input, 7-segment decoder in Verilog. The first step in the process is to create the truth table for the outputs that will drive the LEDs in the display. We'll call these outputs F_a, F_b, . . ., F_g. Example 3.4 shows how to construct the truth table for the 7-segment display decoder. In this table, a logic 1 corresponds to the LED being ON.

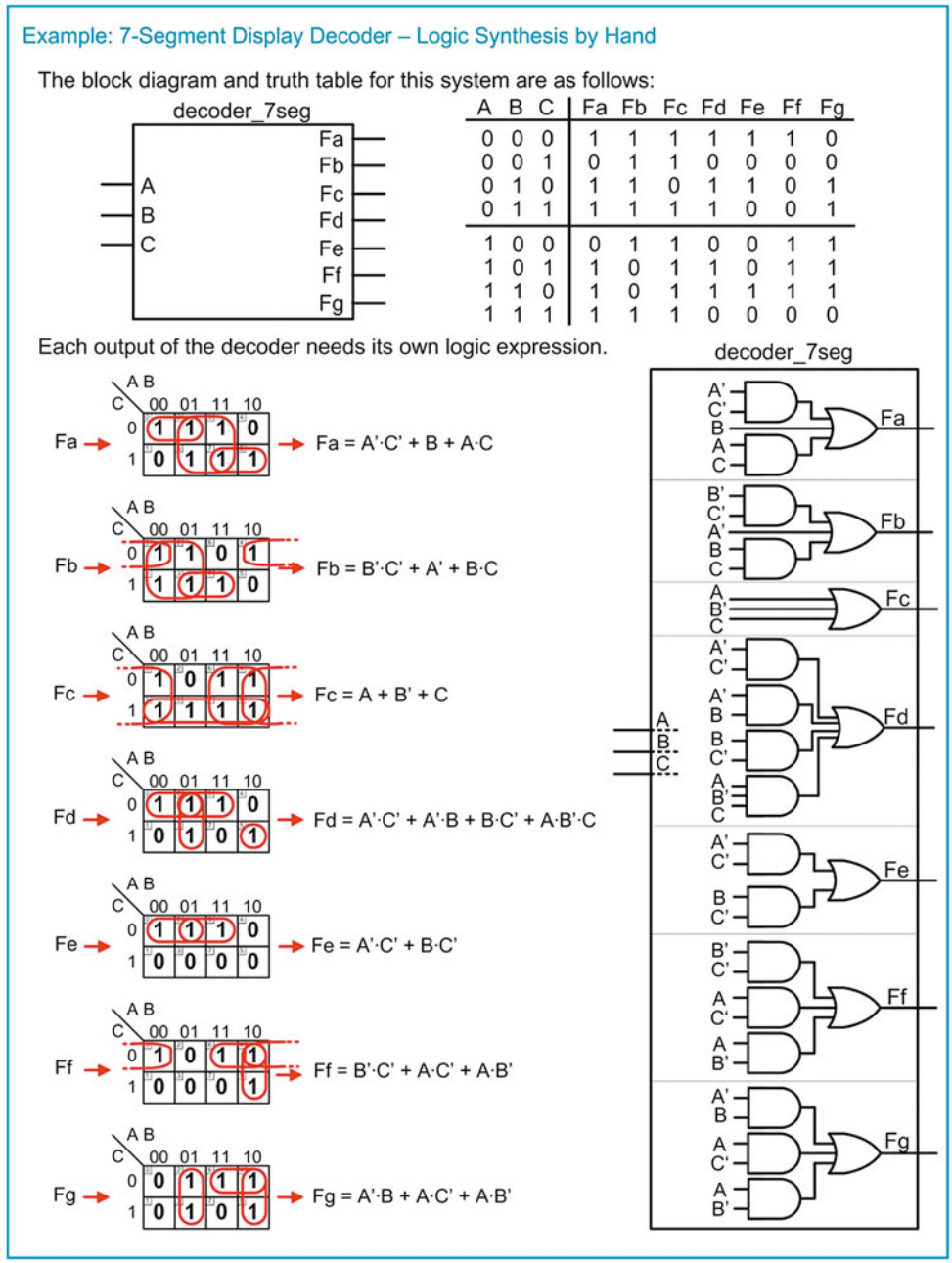
Example: 7-Segment Display Decoder - Truth Table



A	B	C		F_a	F_b	F_c	F_d	F_e	F_f	F_g
0	0	0		1	1	1	1	1	1	0
0	0	1		0	1	1	0	0	0	0
0	1	0		1	1	0	1	1	0	1
0	1	1		1	1	1	1	0	0	1
1	0	0		0	1	1	0	0	1	1
1	0	1		1	0	1	1	0	1	1
1	1	0		1	0	1	1	1	1	1
1	1	1		1	1	1	0	0	0	0

Example 3.4
7-Segment display decoder—truth table

If we wish to model this decoder using logical operators, we need to first create the seven separate combinational logic expressions for each output. Each of the outputs (F_a – F_g) can be put into a 3-input K-map to find the minimized logic expression. Example 3.5 shows the derivation of the logic expressions for the decoder from the truth table in Example 3.4 using Karnaugh maps.

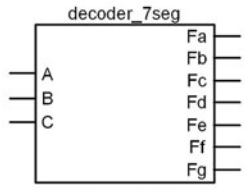


Example 3.5
7-Segment display decoder—logic synthesis by hand

Now these seven logic expressions can be modeled in Verilog. Example 3.6 shows how to model the 7-segment decoder in Verilog using continuous assignment with logic operators.

Example: 7-Segment Display Decoder – Verilog Modeling using Logical Operators

The block diagram and truth table for this system are as follows:



A	B	C	Fa	Fb	Fc	Fd	Fe	Ff	Fg
0	0	0	1	1	1	1	1	1	0
0	0	1	0	1	1	0	0	0	0
0	1	0	1	1	0	1	1	0	1
0	1	1	1	1	1	1	0	0	1
1	0	0	0	1	1	0	0	1	1
1	0	1	1	0	1	1	0	1	1
1	1	0	1	0	1	1	1	1	1
1	1	1	1	1	1	0	0	0	0

```
module decoder_7seg (output wire Fa, Fb, Fc, Fd, Fe, Ff, Fg,
                     input  wire A, B, C);

   assign Fa = (~A & ~C) | (B) | (A & C);
   assign Fb = (~B & ~C) | (~A) | (B & C);
   assign Fc = (A) | (~B) | (C);
   assign Fd = (~A & ~C) | (~A & B) | (B & ~C) | (A & ~B & C);
   assign Fe = (~A & ~C) | (B & ~C);
   assign Ff = (~B & ~C) | (A & ~C) | (A & ~B);
   assign Fg = (~A & B) | (A & ~C) | (A & ~B);

endmodule
```

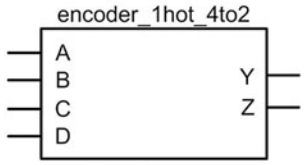
Example 3.6
7-Segment display decoder—Verilog modeling using logical operators

3.2.4 Logical Operator Example: One-Hot Encoder

A one-hot binary encoder has n outputs and 2^n inputs. The output will be an n-bit, binary code which corresponds to an assertion on one and only one of the inputs. Example 3.7 shows the process of designing a 4-to-2 binary encoder by hand (i.e., using the classical digital design approach) in order to find the logic expression to model in Verilog using logical operators.

Example: 4-to-2 Binary Encoder – Logic Synthesis by Hand

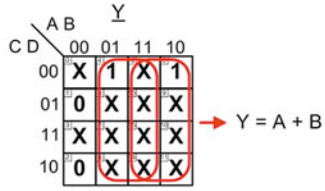
The block diagram and truth table for this system are as follows:

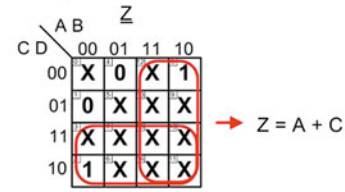


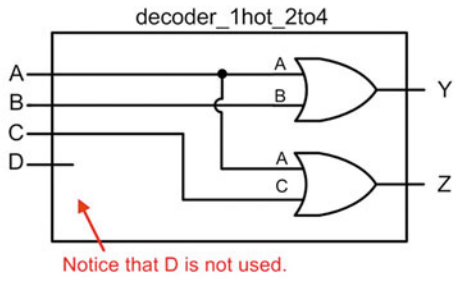
A	B	C	D	Y	Z
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

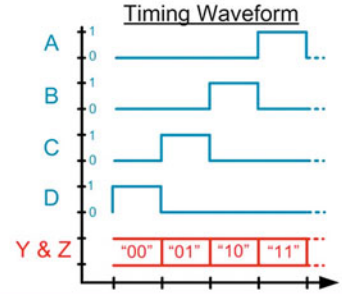
D => "00"
C => "01"
B => "10"
A => "11"

When designing this circuit, each output needs to have its own separate combinational logic circuit. When constructing the K-maps for Y and Z, each will have 4-inputs (A, B, C, D). The output values for many of the input codes are not specified in the above truth table. As such, we can use Don't Cares (X) to simplify the logic.







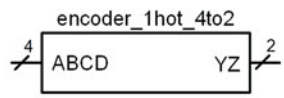


Example 3.7
4-to-2 Binary encoder—logic synthesis by hand

Example 3.8 shows how to model the encoder with continuous assignments and logical operators using the logic expressions from Example 3.7.

Example: 4-to-2 Binary Encoder – Verilog Modeling using Logical Operators

The block diagram and truth table for this system are as follows:



ABCD	YZ
"0 0 0 1"	"00"
"0 0 1 0"	"01"
"0 1 0 0"	"10"
"1 0 0 0"	"11"

The following implements the behavior of the encoder with continuous assignment and logical operators.

```
module encoder_1hot_4to2 (output wire [1:0] YZ,
                          input  wire [3:0] ABCD);

   assign YZ[1] = ABCD[3] | ABCD[2];
   assign YZ[0] = ABCD[3] | ABCD[1];

endmodule
```

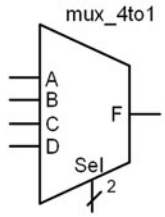
Example 3.8
4-to-2 Binary encoder—Verilog modeling using logical operators

3.2.5 Logical Operator Example: Multiplexer

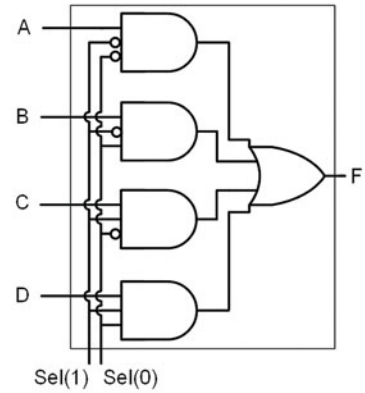
A multiplexer is a circuit that passes one of its multiple inputs to a single output based on a select input. This can be thought of as a digital switch. The multiplexer has n select lines, 2^n inputs, and 1 output. Example 3.9 shows the process of modeling a 4-to-1 multiplexer using continuous signal assignments and logical operators.

Example: 4-to-1 Multiplexer – Verilog Modeling using Logical Operators

The symbol and truth table for a 4-to-1 multiplexer are shown below. This can be implemented using a simple sum of products form based on the identity theorem and the appropriate inversions of the select line.



Sel	F
"00"	A
"01"	B
"10"	C
"11"	D



The following shows how to model the behavior of the mux using continuous assignment and logical operators.

```
module mux_4to1 (output wire F,
                 input  wire A, B, C, D,
                 input  wire [1:0] Sel);

   assign F = (A & ~Sel[1] & ~Sel[0]) |
              (B & ~Sel[1] &  Sel[0]) |
              (C &  Sel[1] & ~Sel[0]) |
              (D &  Sel[1] &  Sel[0]);

endmodule
```

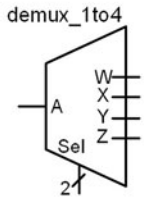
Example 3.9
4-to-1 Multiplexer—Verilog modeling using logical operators

3.2.6 Logical Operator Example: Demultiplexer

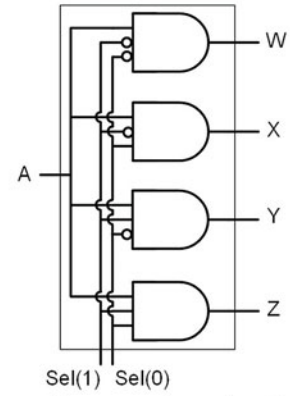
A demultiplexer works in a complementary fashion to a multiplexer. A demultiplexer has one input that is routed to one of its multiple outputs. The output that is active is dictated by a select input. A demux has n select lines that chooses to route the input to one of its 2^n outputs. When an output is not selected, it outputs a logic 0. Example 3.10 shows how to model the demultiplexer in Verilog using continuous assignments and logical operators.

Example: 1-to-4 Demultiplexer – Verilog Modeling using Logical Operators

The symbol and truth table for the 1-to-4 demultiplexer are shown below. This can be implemented using set of simple product terms based on the identity theorem and the appropriate inversions of the select line.



Sel	W	X	Y	Z
"00"	A	0	0	0
"01"	0	A	0	0
"10"	0	0	A	0
"11"	0	0	0	A



The following shows the behavior of the demux using continuous assignments with logical operators.

```
module demux_1to4 (output wire W, X, Y, Z,
                   input  wire A,
                   input  wire [1:0] Sel);

   assign W = (A & ~Sel[1] & ~Sel[0]);
   assign X = (A & ~Sel[1] &  Sel[0]);
   assign Y = (A &  Sel[1] & ~Sel[0]);
   assign Z = (A &  Sel[1] &  Sel[0]);

endmodule
```

Example 3.10
1-to-4 Demultiplexer—Verilog modeling using logical operators

Concept Check

CC3.2 Why does modeling combinational logic in its canonical form with continuous assignment and logical operators defeat the purpose of the modern digital design flow?

(A) It requires the designer to first create the circuit using the classical digital design approach and then enter it into the HDL in a form that is essentially a text-based netlist. This doesn't take advantage of the abstraction capabilities and automated synthesis in the modern flow.

(B) It cannot be synthesized because the order of precedence of the logical operators in Verilog doesn't match the precedence defined in Boolean algebra.

(C) The circuit is in its simplest form, so there is no work for the synthesizer to do.

(D) It doesn't allow an *else* clause to cover the outputs for any remaining input codes not explicitly listed.

3.3 Continuous Assignment with Conditional Operators

Logical operators are good for describing the behavior of small circuits; however, in the prior examples we still needed to create the canonical sum of products logic expression by hand before describing the functionality with logical operators. The true power of an HDL is when the behavior of the system can be

described fully without requiring any hand design. The conditional operator allows us to describe a continuous assignment using Boolean conditions that effect the values of the result. In this approach, we use the conditional operator (**?**) in conjunction with the continuous assignment keyword **assign**.

3.3.1 Conditional Operator Example: SOP Circuit

Example 3.11 shows how to design a Verilog model of a combinational logic circuit using continuous assignment with conditional operators. Note that this example uses the same truth table as in Example 3.2 to illustrate a comparison between approaches.

Example: Modeling Combinational Logic using Continuous Assignment with Conditional Operators (1)

Implement the following truth table using a continuous assignment with conditional operators.

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

We can implement the entire truth table in its current form by nesting conditional operators to explicitly list out each possible input code and its corresponding output as follows:

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

   assign F = ((A == 1'b0) && (B == 1'b0) && (C == 1'b0)) ? 1'b1 :
              ((A == 1'b0) && (B == 1'b0) && (C == 1'b1)) ? 1'b0 :
              ((A == 1'b0) && (B == 1'b1) && (C == 1'b0)) ? 1'b1 :
              ((A == 1'b0) && (B == 1'b1) && (C == 1'b1)) ? 1'b0 :
              ((A == 1'b1) && (B == 1'b0) && (C == 1'b0)) ? 1'b0 :
              ((A == 1'b1) && (B == 1'b0) && (C == 1'b1)) ? 1'b0 :
              ((A == 1'b1) && (B == 1'b1) && (C == 1'b0)) ? 1'b1 :
              ((A == 1'b1) && (B == 1'b1) && (C == 1'b1)) ? 1'b0 :
               1'b0;

endmodule
```

We can reduce the length of this model by only explicitly listing the input conditions for when the output is TRUE and allowing the final FALSE value to cover all other inputs.

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

   assign F = ((A == 1'b0) && (B == 1'b0) && (C == 1'b0)) ? 1'b1 :
              ((A == 1'b0) && (B == 1'b1) && (C == 1'b0)) ? 1'b1 :
              ((A == 1'b1) && (B == 1'b1) && (C == 1'b0)) ? 1'b1 :
               1'b0;

endmodule
```

Example 3.11
Combinational logic using continuous assignment with conditional operators (1)

In the prior example, the conditional operator was based on a truth table. Conditional operators can also be used to model logic expressions. Example 3.12 shows how to design a Verilog model of a combinational logic circuit when the logic expression is already known. Note that this example again uses the same truth table as in Examples 3.2 and 3.11 to illustrate a comparison between approaches.

Example: Modeling Combinational Logic using Continuous Assignment with Conditional Operators (2)

Implement the following truth table using a continuous assignment with conditional operators.

In this example, a K-map was used to find a minimized logic expression of:

$$F = C' \cdot (A' + B)$$

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

We can implement the conditional operator using input variables and Boolean operators to directly model the logic expression.

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

   assign F = ( !C && (!A || B) ) ? 1'b1 : 1'b0;

endmodule
```

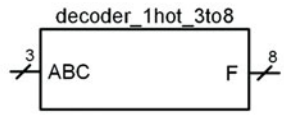
Example 3.12
Combinational logic using continuous assignment with conditional operators (2)

3.3.2 Conditional Operator Example: One-Hot Decoder

Example 3.13 shows how to model the 3-to-8 one-hot decoder in Verilog using continuous assignment with conditional operators. This description of a one-hot decoder can be simplified by using vector notation for the ports.

Example: 3-to-8 One-Hot Decoder – Verilog Modeling using Conditional Operators

The block diagram and truth table for this system are as follows. Notice that the input and output ports now use vectors in order to create a more compact description.



ABC	F(7)	F(6)	F(5)	F(4)	F(3)	F(2)	F(1)	F(0)
"000"	0	0	0	0	0	0	0	1
"001"	0	0	0	0	0	0	1	0
"010"	0	0	0	0	0	1	0	0
"011"	0	0	0	0	1	0	0	0
"100"	0	0	0	1	0	0	0	0
"101"	0	0	1	0	0	0	0	0
"110"	0	1	0	0	0	0	0	0
"111"	1	0	0	0	0	0	0	0

The following shows a technique to model the decoder using continuous assignment with conditional operators. Note that the output will be "unknown" (X) if the input code is not one of the eight possible binary input values.

```
module decoder_1hot_3to8 (output wire [7:0] F,
                          input  wire [2:0] ABC);

   assign F = (ABC == 3'b000) ? 8'b0000_0001 :
              (ABC == 3'b001) ? 8'b0000_0010 :
              (ABC == 3'b010) ? 8'b0000_0100 :
              (ABC == 3'b011) ? 8'b0000_1000 :
              (ABC == 3'b100) ? 8'b0001_0000 :
              (ABC == 3'b101) ? 8'b0010_0000 :
              (ABC == 3'b110) ? 8'b0100_0000 :
              (ABC == 3'b111) ? 8'b1000_0000 :
             8'bXXXX_XXXX;

endmodule
```

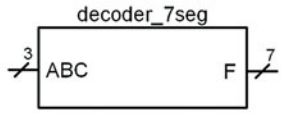
Example 3.13
3-to-8 One-hot decoder—Verilog modeling using conditional operators

3.3.3 Conditional Operator Example: 7-Segment Display Decoder

Example 3.14 shows how to model the 7-segment decoder in Verilog using continuous assignment with conditional operators. Again, a more compact description of the decoder can be accomplished if the ports are described as vectors.

Example: 7-Segment Decoder – Verilog Modeling using Conditional Operators

The block diagram and truth table for this system are as follows:



	a	b	c	d	e	f	g
ABC	F(6)	F(5)	F(4)	F(3)	F(2)	F(1)	F(0)
"000"	1	1	1	1	1	1	0
"001"	0	1	1	0	0	0	0
"010"	1	1	0	1	1	0	1
"011"	1	1	1	1	0	0	1
"100"	0	1	1	0	0	1	1
"101"	1	0	1	1	0	1	1
"110"	1	0	1	1	1	1	1
"111"	1	1	1	0	0	0	0

The following shows a technique to model the decoder using continuous assignment with conditional operators.

```
module decoder_7seg (output wire [6:0] F,
                     input  wire [2:0] ABC);

   assign F = (ABC == 3'b000) ? 7'b111_1110 :
              (ABC == 3'b001) ? 7'b011_0000 :
              (ABC == 3'b010) ? 7'b110_1101 :
              (ABC == 3'b011) ? 7'b111_1001 :
              (ABC == 3'b100) ? 7'b011_0011 :
              (ABC == 3'b101) ? 7'b101_1011 :
              (ABC == 3'b110) ? 7'b101_1111 :
              (ABC == 3'b111) ? 7'b111_0000 :
              8'bXXXX_XXXX;

endmodule
```

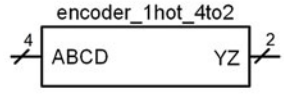
Example 3.14
7-Segment display decoder—Verilog modeling using conditional operators

3.3.4 Conditional Operator Example: One-Hot Decoder

Example 3.15 shows how to model the encoder with continuous assignments and conditional operators. Notice that using this approach does not require synthesizing the logic expressions by hand but rather can model the functionality directly from the truth table.

Example: 4-to-2 Binary Encoder – Verilog Modeling using Conditional Operators

The block diagram and truth table for this system are as follows:



ABCD	YZ
"0 0 0 1"	"00"
"0 0 1 0"	"01"
"0 1 0 0"	"10"
"1 0 0 0"	"11"

The following implements the behavior of the encoder with continuous assignment and conditional operators.

```
module encoder_1hot_4to2 (output wire [1:0] YZ,
                          input  wire [3:0] ABCD);

   assign YZ = (ABCD == 4'b0001) ? 2'b00 :
               (ABCD == 4'b0010) ? 2'b01 :
               (ABCD == 4'b0100) ? 2'b10 :
               (ABCD == 4'b1000) ? 2'b11 :
               2'bXX;
endmodule
```

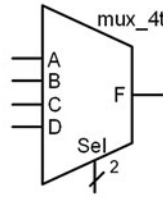
Example 3.15
4-to-2 Binary encoder—Verilog modeling using conditional operators

3.3.5 Conditional Operator Example: Multiplexer

Example 3.16 shows the process of modeling a 4-to-1 multiplexer using continuous signal assignments and conditional operators. Notice that this approach can also be implemented directly from the truth table.

Example: 4-to-1 Multiplexer – Verilog Modeling using Conditional Operators

The symbol and truth table for the 4-to-1 multiplexer are as follows:



Sel	F
"00"	A
"01"	B
"10"	C
"11"	D

The following shows how to model the behavior of the mux using continuous assignment and conditional operators.

```
module mux_4to1 (output wire F,
                 input  wire A, B, C, D,
                 input  wire [1:0] Sel);

   assign F = (Sel == 2'b00) ? A :
              (Sel == 2'b01) ? B :
              (Sel == 2'b10) ? C :
              (Sel == 2'b11) ? D :
              1'bX;

endmodule
```

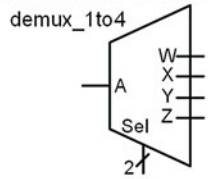
Example 3.16
4-to-1 Multiplexer—Verilog modeling using conditional operators

3.3.6 Conditional Operator Example: Demultiplexer

Example 3.17 shows how to model the demultiplexer in Verilog using continuous assignments and conditional operators. Notice that this approach can be implemented directly from the truth table as well.

Example: 1-to-4 Demultiplexer – Verilog Modeling using Conditional Operators

The symbol and truth table for the 1-to-4 demultiplexer are as follows:



Sel	W	X	Y	Z
"00"	A	0	0	0
"01"	0	A	0	0
"10"	0	0	A	0
"11"	0	0	0	A

The following shows the behavior of the demux using continuous assignments with conditional operators.

```
module demux_1to4 (output wire W, X, Y, Z,
                   input  wire A,
                   input  wire [1:0] Sel);

   assign W = (Sel == 2'b00) ? A : 1'b0;
   assign X = (Sel == 2'b01) ? A : 1'b0;
   assign Y = (Sel == 2'b10) ? A : 1'b0;
   assign Z = (Sel == 2'b11) ? A : 1'b0;

endmodule
```

Example 3.17
1-to-4 Demultiplexer—Verilog modeling using conditional operators

Concept Check

CC3.3 Why does a continuous signal assignment with conditional operators better reflect the modern digital design flow compared to using logical operators?

(A) It allows the logic to be modeled directly from its functional description as opposed to from the final logic expressions, which must be determined prior to HDL modeling. This allows the continuous signal assignment approach to take advantage of automated synthesis and avoids any hand design.

(B) A conditional operator has a final clause that covers any input cases not explicitly listed. This makes it more like a programming language operator.

(C) A conditional operator has a final clause that covers any input cases not explicitly listed. This allows a final assignment of "X," which provides the ability to assign any outputs not explicitly listed to be treated as "unknowns".

(D) The conditional operators can model the entire logic circuit in one assignment while the logical operator approach often takes multiple separate assignments.

3.4 Continuous Assignment with Delay

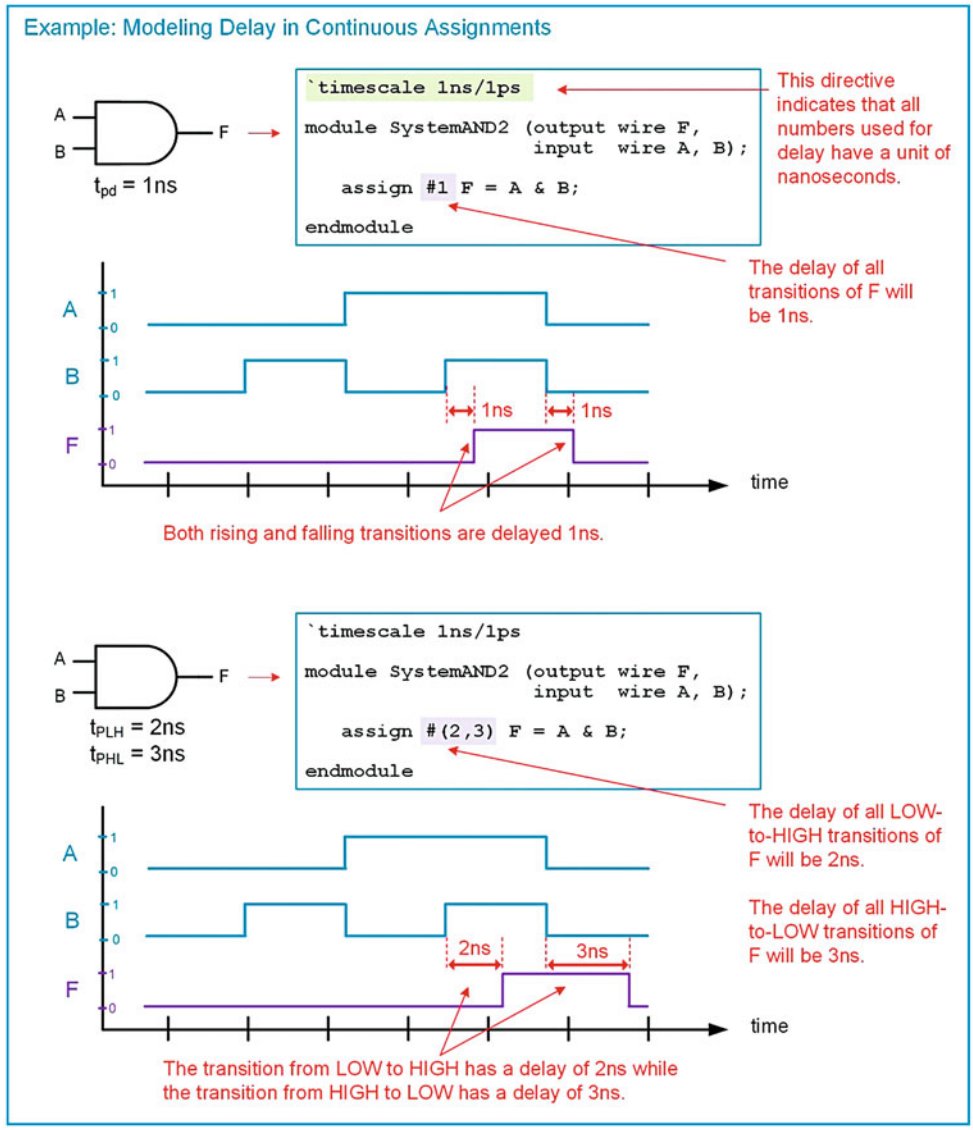
Verilog provides the ability to model gate delays when using a continuous assignment. The **#** is used to indicate a delayed assignment. For combinational logic circuits, the delay can be specified for all transitions, for rising and falling transitions separately, and for rising, falling, and transitions to the value *off* separately. A transition to *off* refers to a transition to Z. If only one delay parameter is specified, it is used to model all delays. If two delay parameters are specified, the first parameter is used for the rise time delay while the second is used to model the fall time delay. If three parameters are specified, the third parameter is used to model the transition to off. Parenthesis are optional but recommended when using multiple delay parameters.

```
assign #(<del_all>)                        <target_net> = <RHS_nets, operators,
                                           etc...>;
assign #(<del_rise, del_fall>)             <target_net> = <RHS_nets, operators,
                                           etc...>;
assign #(<del_rise, del_fall, del_off>)    <target_net> = <RHS_nets, operators,
                                           etc...>;
```

Example:

```
assign #1        F = A;   // Delay of 1 on all transitions.
assign #(2,3)    F = A;   // Delay of 2 for rising transitions and 3 for falling.
assign #(2,3,4)  F = A;   // Delay of 2 for rising, 3 for falling, and 4 for
                             off transition.
```

When using delay, it is typical to include the **`timescale** directive to provide the units of the delay being specified. Example 3.18 shows a graphical depiction of using delay with continuous assignments when modeling combinational logic circuits.



Example 3.18
Modeling delay in continuous assignments

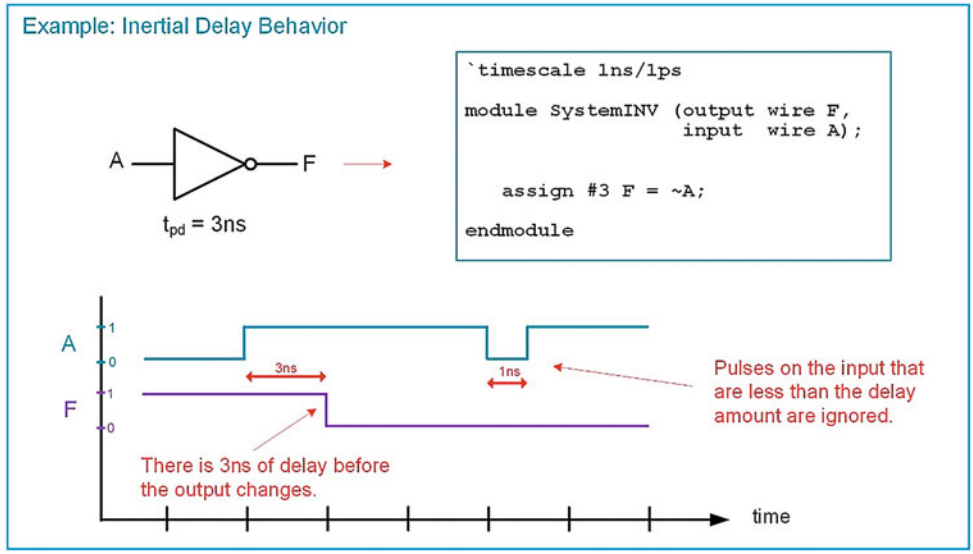
Verilog also provides a mechanism to model a range of delays that are selected by a switch set in the CAD compiler. There are three delays categories that can be specified: *minimum*, *typical*, and *maximum*. The delays are separated by a "**:**". The following is the syntax of how to use the delay range capability.

```
assign #(<min>:<typ>:<max>) <target_net> = <RHS_nets, operators, etc...>;
```

Example:

```
assign #(1:2:3)                F = A;  // Specifying a range of delays for all
                                          transitions.
assign #(1:1:2, 2:2:3)         F = A;  // Specifying a range of delays for
                                          rising/falling.
assign #(1:1:2, 2:2:3, 4:4:5)  F = A;  // Specifying a range of delays for
                                          each transition.
```

The delay modeling capability in continuous assignment is designed to model the behavior of real combinational logic with respect to short duration pulses. When a pulse is shorter than the delay of the combinational logic gate, the pulse is ignored. Ignoring brief input pulses on the input accurately models the behavior of on-chip gates. When the input pulse is faster than the delay of the gate, the output of the gate does not have time to respond. As a result, there will not be a logic change on the output. This is called *inertial delay* modeling and is the default behavior when using continuous assignments. Example 3.19 shows a graphical depiction of inertial delay behavior in Verilog.



Example 3.19
Inertial delay modeling when using continuous assignment

Concept Check

CC3.4 Can a delayed signal assignment impact multiple continuous signal assignments?

(A) Yes. If a signal assignment with delay is made to a signal that is also used as an input in a separate continuous signal assignment, then the delay will propagate through both assignments.

(B) No. Only the assignment in which the delay is used will experience the delay.

Summary

- *Concurrency* is the term that describes operations being performed in parallel. This allows real-world system behavior to be modeled.
- Verilog provides the *continuous assignment* operator to support modeling concurrent combinational logic operations.
- Complex logic circuits can be implemented by using continuous assignment with *logical operators* or *conditional operators*.
- Delay can also be included in continuous assignments.
- Verilog supports a variety of delay models including delay for all transitions, separate delay for rising and falling transitions, separate delay for rising, falling, and transitions to off, and finally support for a min:typ:max delay that is selected by a compiler switch.

Exercise Problems

Section 3.1: Verilog Operators

3.1.1 What is the purpose of the continuous assignment operator?

3.1.2 If two continuous assignments are made to the same net, which one will take priority?

3.1.3 What is the difference between a bitwise logical AND (&) operation and a reduction AND (&) operation?

3.1.4 How is a conditional operator (?) similar to an if/then programming construct?

3.1.5 How many bits will the target vector F need to be if the following concatenation assignment is made?

F = {4'hA, 2'b00};

3.1.6 How many bits will the target vector F need to be if the following replication assignment is made?

F = {3{4'hA}};

3.1.7 When adding two unsigned vectors of different sizes using the + numerical operator, what happens to the smaller vector prior to the addition?

3.1.8 What operation has the highest precedence operation in Verilog?

Section 3.2: Continuous Assignment with Logical Operators

3.2.1 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 3.1. Use continuous assignment with logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

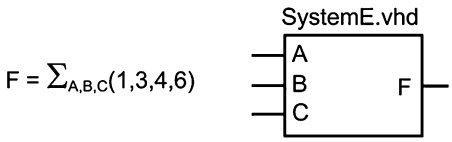


Fig. 3.1
System E Functionality

3.2.2 Design a Verilog model to implement the behavior described by the 3-input maxterm list shown in Fig. 3.2. Use continuous assignment with logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

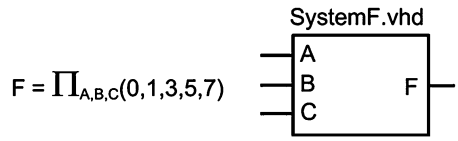


Fig. 3.2
System F Functionality

3.2.3 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 3.3. Use continuous assignment with logical operators. 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

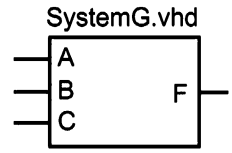


Fig. 3.3
System G Functionality

3.2.4 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 3.4. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

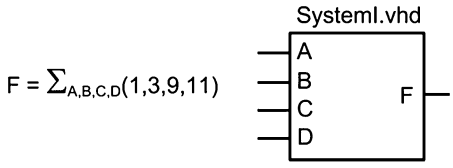


Fig. 3.4
System I Functionality

3.2.5 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 3.5. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

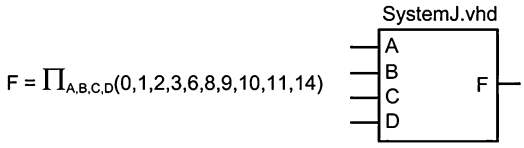


Fig. 3.5
System J Functionality

3.2.6 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 3.6. Use continuous assignment and logical operators. 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

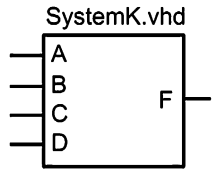
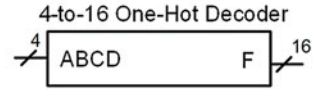


Fig. 3.6
System K Functionality

3.2.7 Design a Verilog model for the 4-to-16 one-hot decoder shown in Fig. 3.7. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided.

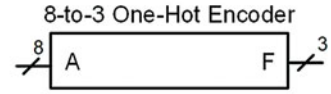


ABCD[3:0]	F[15:0]
0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 1 1	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 1 0 0	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 1 0 1	0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 1 1 0	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 1 1 1	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
1 0 0 0	0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
1 0 0 1	0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
1 0 1 0	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
1 0 1 1	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 1	0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0	0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```
module decoder_1hot_4to16
  (output wire [15:0] F,
   input  wire [3:0]  ABCD);
                      :
```

Fig. 3.7
4-to-16 One-Hot Decoder Functionality

3.2.8 Design a Verilog model for the 8-to-3 one-hot encoder shown in Fig. 3.8. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided.

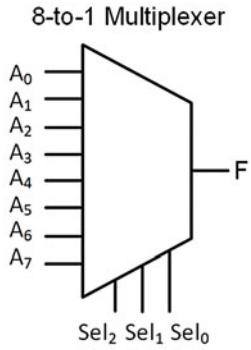


A[7:0]	F[2:0]
0 0 0 0 0 0 0 1	0 0 0
0 0 0 0 0 0 1 0	0 0 1
0 0 0 0 0 1 0 0	0 1 0
0 0 0 0 1 0 0 0	0 1 1
0 0 0 1 0 0 0 0	1 0 0
0 0 1 0 0 0 0 0	1 0 1
0 1 0 0 0 0 0 0	1 1 0
1 0 0 0 0 0 0 0	1 1 1

```
module encoder_8to3_binary
  (output wire [2:0] F,
   input  wire [7:0] A);
                    :
```

Fig. 3.8
8-to-3 One-Hot Encoder Functionality

3.2.9 Design a Verilog model for the 8-to-1 multiplexer shown in Fig. 3.9. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided.

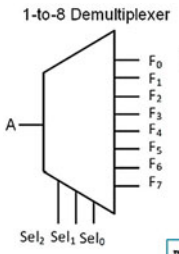


Sel_2	Sel_1	Sel_0	F
0	0	0	A_0
0	0	1	A_1
0	1	0	A_2
0	1	1	A_3
1	0	0	A_4
1	0	1	A_5
1	1	0	A_6
1	1	1	A_7

```
module mux_8to1
  (output wire F,
   input  wire [7:0] A,
   input  wire [2:0] Sel);
                     :
```

Fig. 3.9
8-to-1 Multiplexer Functionality

3.2.10 Design a Verilog model for the 1-to-8 demultiplexer shown in Fig. 3.10. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided.



Sel_2	Sel_1	Sel_0	F_7	F_6	F_5	F_4	F_3	F_2	F_1	F_0
0	0	0	0	0	0	0	0	0	0	A
0	0	1	0	0	0	0	0	0	A	0
0	1	0	0	0	0	0	0	A	0	0
0	1	1	0	0	0	0	A	0	0	0
1	0	0	0	0	0	A	0	0	0	0
1	0	1	0	0	A	0	0	0	0	0
1	1	0	0	A	0	0	0	0	0	0
1	1	1	A	0	0	0	0	0	0	0

```
module demux_1to8
  (output wire [7:0] F,
   input  wire A,
   input  wire [2:0] Sel);
                     :
```

Fig. 3.10
1-to-8 Demultiplexer Functionality

Section 3.3: Continuous Assignment with Conditional Operators

3.3.1 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 3.1. Use continuous assignment with conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.3.2 Design a Verilog model to implement the behavior described by the 3-input maxterm list shown in Fig. 3.2. Use continuous assignment with conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.3.3 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 3.3. Use continuous assignment with conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.3.4 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 3.4. Use continuous assignment and conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.3.5 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 3.5. Use continuous assignment and conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.3.6 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 3.6. Use continuous assignment and conditional operators. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.3.7 Design a Verilog model for the 4-to-16 one-hot decoder shown in Fig. 3.7. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided.

3.3.8 Design a Verilog model for the 8-to-3 one-hot encoder shown in Fig. 3.8. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided.

3.3.9 Design a Verilog model for the 8-to-1 multiplexer shown in Fig. 3.9. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided.

3.3.10 Design a Verilog model for the 1-to-8 demultiplexer shown in Fig. 3.10. Use continuous assignment and logical operators. Declare your module and ports to match the block diagram provided.

Section 3.4: Continuous Assignment with Delay

3.4.1 Design a Verilog model to implement the behavior described by the 3-input minterm list shown in Fig. 3.1. Use continuous assignment with logical operators and give each logic operation 1 ns of delay. Declare your module and

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

3.4.2 Design a Verilog model to implement the behavior described by the 3-input maxterm list shown in Fig. 3.2. Use continuous assignment with logical operators and give each rising transition a delay of 1 ns and each falling transition a delay of 2 ns. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.4.3 Design a Verilog model to implement the behavior described by the 3-input truth table shown in Fig. 3.3. Use continuous assignment with conditional operators and give the entire logic operation a delay of 3 ns. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.4.4 Design a Verilog model to implement the behavior described by the 4-input minterm list shown in Fig. 3.4. Use continuous assignment with conditional operators and give rising transitions a delay of 3 ns and falling transitions a delay of 2 ns. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.4.5 Design a Verilog model to implement the behavior described by the 4-input maxterm list shown in Fig. 3.5. Use continuous assignment and logical operators and give each logic operation a delay of 1, 2, and 3 ns, respectively, for the operation's min:typ:max behavior. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.

3.4.6 Design a Verilog model to implement the behavior described by the 4-input truth table shown in Fig. 3.6. Use continuous assignment and conditional operators and give the entire operation a delay of 1, 2, and 3 ns, respectively, for the operation's min:typ:max behavior. Declare your module and ports to match the block diagram provided. Use the type wire for your ports.