GATE-LEVEL MODELING

- A design at a low level of abstraction gate level
- Most digital design is now done at gate level or higher levels of abstraction
- At gate level, the circuit is described in terms of gates (e.g., and, nand)
- Hardware design at this level is intuitive for a user with a basic knowledge of digital logic design because it is possible to see a one-to-one correspondence between the logic circuit diagram and the Verilog description
- Actually, the lowest level of abstraction is switch- (transistor-) level modeling
- However, with designs getting very complex, very few hardware designers work at switch level

GATE-LEVEL MODELING - LEARNING OBJECTIVES

- Identify logic gate primitives provided in Verilog
- Understand instantiation of gates, gate symbols, and truth tables for and/or and buf/not type gates
- Understand how to construct a Verilog description from the logic diagram of the circuit
- Describe rise, fall, and turn-off delays in the gate-level design
- Explain min, max, and typ delays in the gate-level design

GATE TYPES

- A logic circuit can be designed by use of logic gates
- Verilog supports basic logic gates as predefined primitives
- These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition
- All logic circuits can be designed by using basic gates
- There are two classes of basic gates:
  - and/or gates and
  - buf/not gates

And/Or Gates

- And/or gates have one scalar output and multiple scalar inputs

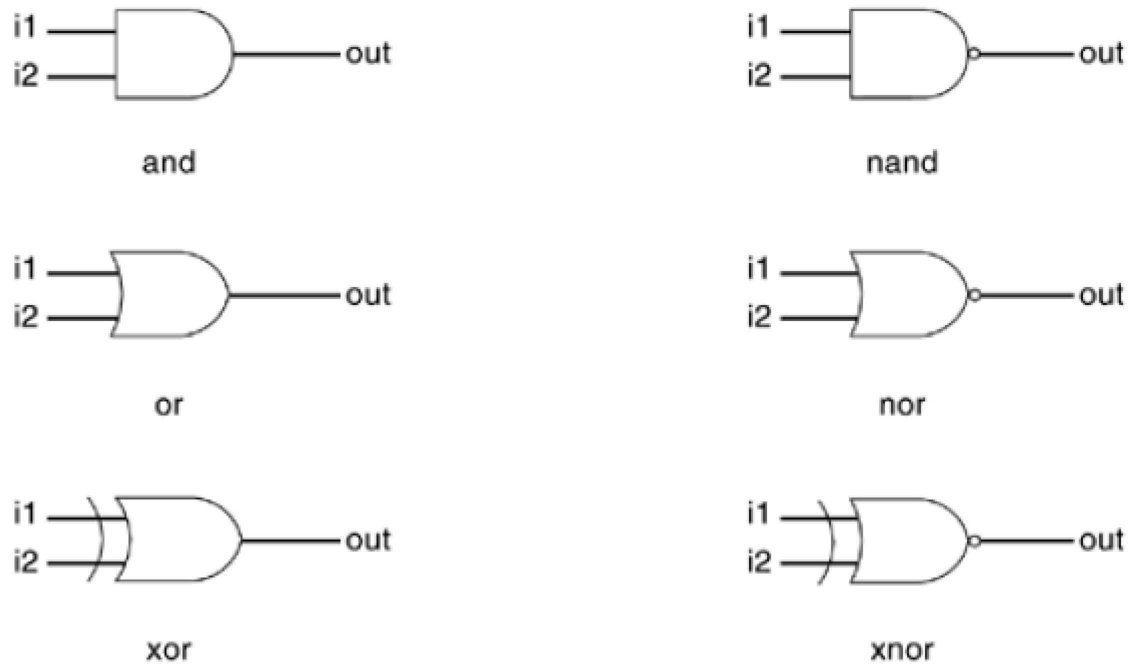
- The first terminal in the list of gate terminals is an output and the other terminals are inputs

- The output of a gate is evaluated as soon as one of the inputs changes

- The and/or gates available in Verilog are shown below
  - and or xor
  - nand nor xnor

And/Or Gates

Figure 1. Basic Gates



- We consider gates with two inputs. The output terminal is denoted by out. Input terminals are denoted by i1 and i2

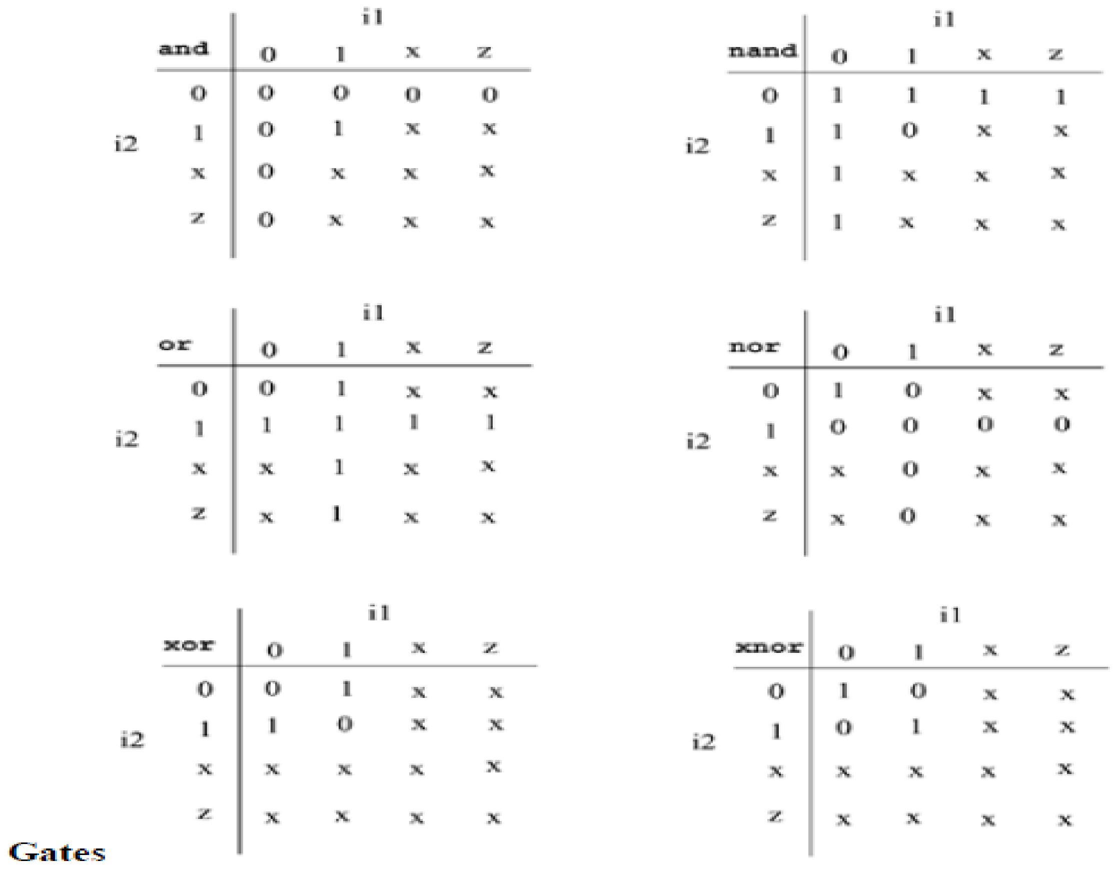
Example 1: Gate Instantiation of And/Or Gates

```
wire OUT, IN1, IN2;
// basic gate instantiations.
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);
or or1(OUT, IN1, IN2);
nor nor1(OUT, IN1, IN2);
xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);
// More than two inputs; 3 input nand gate
nand na1_3inp(OUT, IN1, IN2, IN3);
// gate instantiation without instance name
and (OUT, IN1, IN2); // legal gate instantiation
```

TRUTH TABLES FOR THESE GATES

- The truth tables for these gates define how outputs for the gates are computed from the inputs
- Truth tables are defined assuming two inputs
- Outputs of gates with more than two inputs are computed by applying the truth table iteratively

Table 1. Truth Tables for And/Or



		i1			
	and	0	1	x	z
	0	0	0	0	0
i2	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x

		i1			
	nand	0	1	x	z
	0	1	1	1	1
i2	1	1	0	x	x
	x	1	x	x	x
	z	1	x	x	x

		i1			
	or	0	1	x	z
	0	0	1	x	x
i2	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x

		i1			
	nor	0	1	x	z
	0	1	0	x	x
i2	1	0	0	0	0
	x	x	0	x	x
	z	x	0	x	x

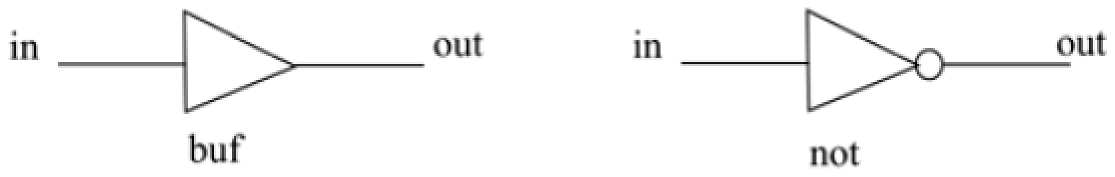
		i1			
	xor	0	1	x	z
	0	0	1	x	x
i2	1	1	0	x	x
	x	x	x	x	x
	z	x	x	x	x

		i1			
	xnor	0	1	x	z
	0	1	0	x	x
i2	1	0	1	x	x
	x	x	x	x	x
	z	x	x	x	x

Gates

Buf/Not Gates

- Buf/not gates have one scalar input and one or more scalar outputs
- The last terminal in the port list is connected to the input. Other terminals are connected to the outputs
- Two basic buf/not gate primitives are provided in Verilog
  - buf not

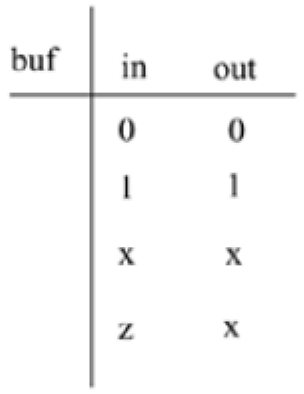


Example 2: Gate Instantiations of Buf/Not Gates

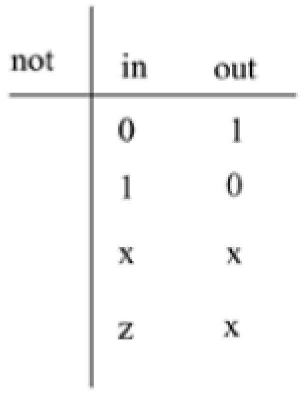
```
// basic gate instantiations.
buf b1(OUT1, IN);
not n1(OUT1, IN);

// More than two outputs
buf b1_2out(OUT1, OUT2, IN);

// gate instantiation without instance name
not (OUT1, IN); // legal gate instantiation
```

Truth Tables for Buf/Not Gates

Table 2. Truth Tables for Buf/Not Gates



buf	in	out
	0	0
	1	1
	x	x
	z	x



not	in	out
	0	1
	1	0
	x	x
	z	x

Gates with an additional control signal on buf and not gates

- **Bufif/notif**
  - bufif1 notif1
  - bufif0 notif0

- These gates are used when a signal is to be driven only when the control signal is asserted

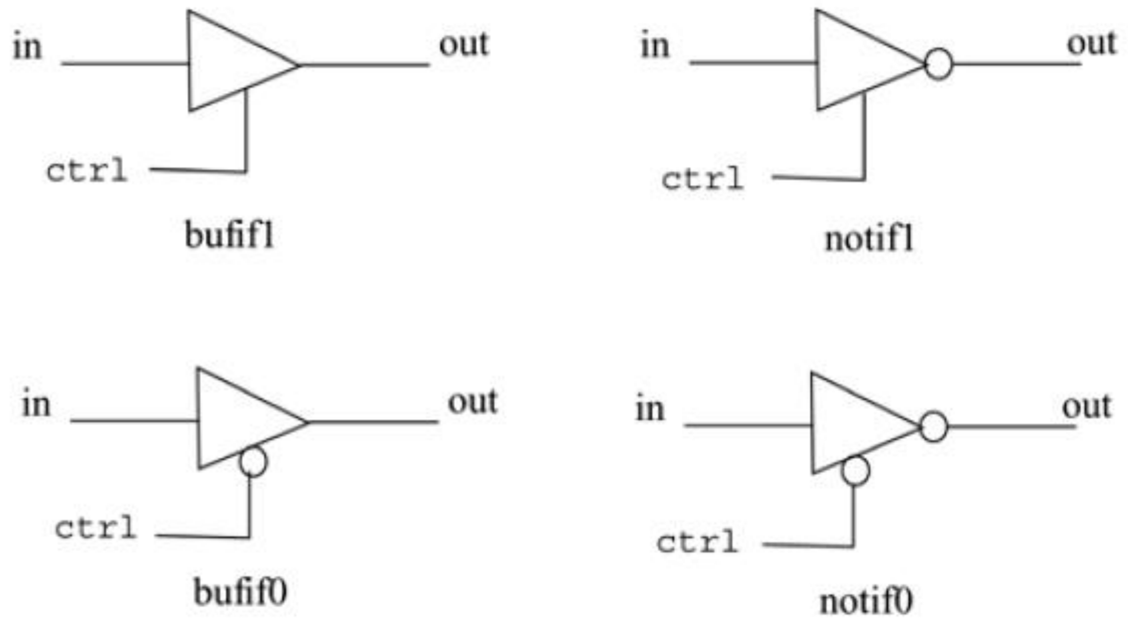
- Such a situation is applicable when multiple drivers drive the signal

- These gates propagate only if their control signal is asserted

- They propagate z if their control signal is deasserted

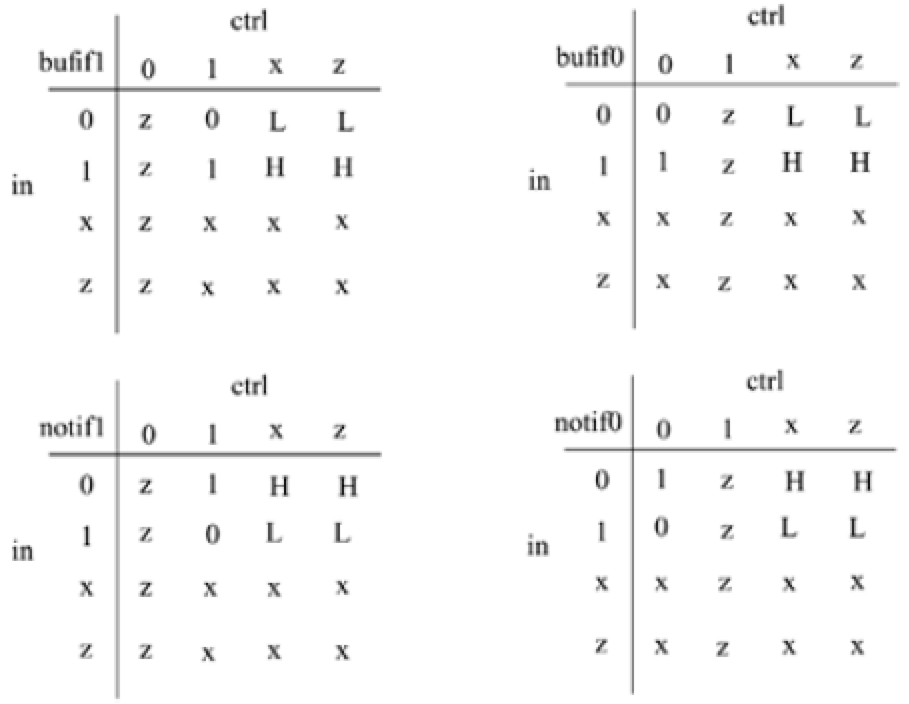
- Symbols for bufif/notif are shown in Figure 3

Figure 3. Gates Bufif and Notif



Truth Tables for Bufif/Notif Gates

Table 3. Truth Tables for Bufif/Notif Gates



		ctrl			
	bufif1	0	1	x	z
	0	z	0	L	L
in	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

		ctrl			
	bufif0	0	1	x	z
	0	0	z	L	L
in	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

		ctrl			
	notif1	0	1	x	z
	0	z	1	H	H
in	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

		ctrl			
	notif0	0	1	x	z
	0	1	z	H	H
in	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

Example 3: Gate Instantiations of Bufif/Notif Gates

- These drivers are designed to drive the signal on mutually exclusive control signals.

- Example 3 shows examples of instantiation of bufif and notif gates.

- **Example 3 Gate Instantiations of Bufif/Notif Gates**

1. //Instantiation of bufif gates.
2. bufif1 b1 (out, in, ctrl);
3. bufif0 b0 (out, in, ctrl);
4. //Instantiation of notif gates
5. notif1 n1 (out, in, ctrl);
6. notif0 n0 (out, in, ctrl);

Array of Instances

- There are many situations when repetitive instances are required
- These instances differ from each other only by the index of the vector to which they are connected
- To simplify specification of such instances, Verilog HDL allows an array of primitive instances to be defined
- Example 4 shows an example of an array of instances

Example 4: Simple Array of Primitive Instances

1. **Example 4 Simple Array of Primitive Instances**
2. wire [7:0] OUT, IN1, IN2;
3. // basic gate instantiations.
4. nand n_gate[7:0](OUT, IN1, IN2);
5. // This is equivalent to the following 8 instantiations
6. nand n_gate0(OUT[0], IN1[0], IN2[0]);
7. nand n_gate1(OUT[1], IN1[1], IN2[1]);
8. nand n_gate2(OUT[2], IN1[2], IN2[2]);
9. nand n_gate3(OUT[3], IN1[3], IN2[3]);
10. nand n_gate4(OUT[4], IN1[4], IN2[4]);
11. nand n_gate5(OUT[5], IN1[5], IN2[5]);
12. nand n_gate6(OUT[6], IN1[6], IN2[6]);
13. nand n_gate7(OUT[7], IN1[7], IN2[7]);

Gate-level multiplexer

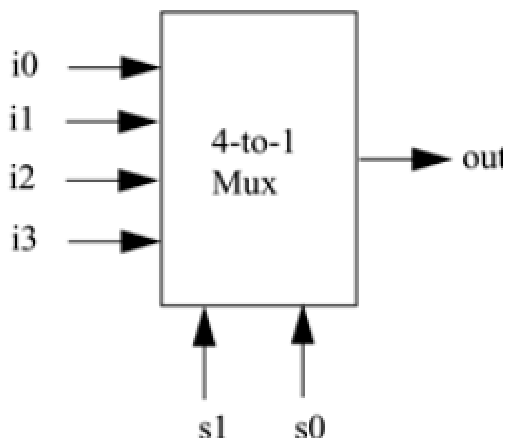
- We will design a 4-to-1 multiplexer with 2 select signals
- Multiplexers serve a useful purpose in logic design
- They can connect two or more sources to a single destination
- They can also be used to implement boolean functions
- We will assume for this example that signals s1 and s0 do not get the value x or z
- The I/O diagram and the truth table for the multiplexer are shown in Figure 4
- The I/O diagram will be useful in setting up the port list for the multiplexer

Gate-level multiplexer

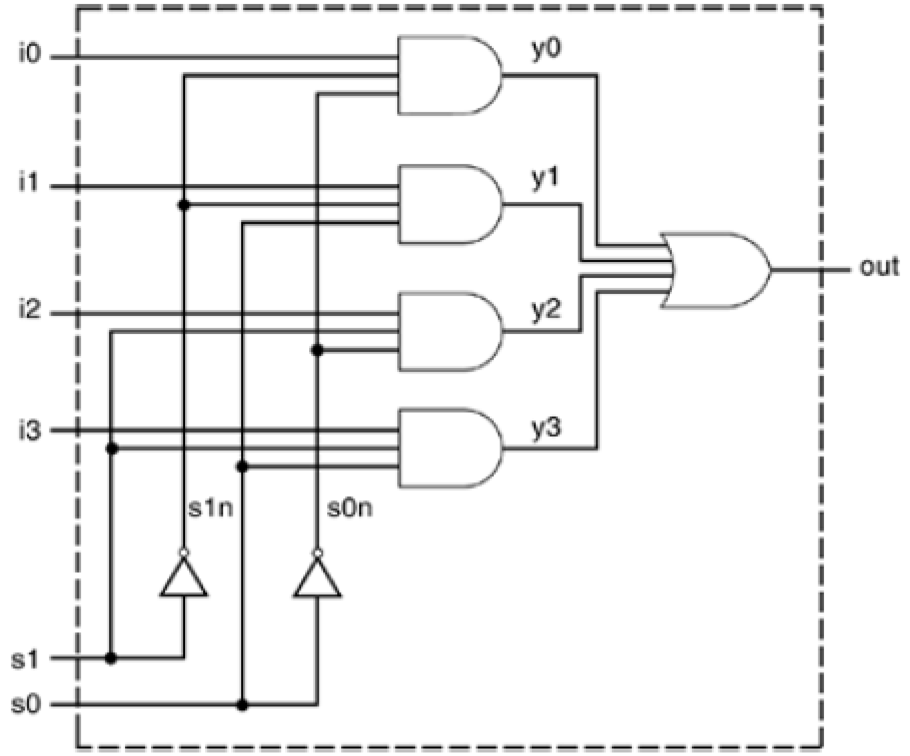
–Cont'd

Figure 5. Logic Diagram for Multiplexer

Figure 4. 4-to-1 Multiplexer



s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3



Gate-level multiplexer

- The logic diagram has a one-to-one correspondence with the Verilog description
- The Verilog description for the multiplexer is shown in Example 5
- Two intermediate nets, s0n and s1n, are created; they are complements of input signals s1 and s0
- Internal nets y0, y1, y2, y3 are also required
- Note that instance names are not specified for primitive gates, not, and, and or
- Instance names are optional for Verilog primitives but are mandatory for instances of user-defined modules

Example 5.1: Verilog Description of Multiplexer

```verilog
// Module 4-to-1 multiplexer. Port list is taken exactly from the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
// Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;
// Gate instantiations
// Create s1n and s0n signals.
not (s1n, s1);
not (s0n, s0);
// 3-input and gates instantiated
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);
// 4-input or gate instantiated
or (out, y0, y1, y2, y3);
endmodule
```

Example 5.2 Stimulus for Multiplexer

```
// Define the stimulus module (no ports)
module stimulus;
// Declare variables to be connected
// to inputs
reg IN0, IN1, IN2, IN3;
reg S1, S0;
// Declare output wire
wire OUTPUT;
// Instantiate the multiplexer
mux4_to_1 mymux(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);
```

Example 5.2 Stimulus for Multiplexer –Cont’d

```
// Stimulate the inputs
// Define the stimulus module (no ports)
initial
begin
// set input lines
IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
#1 $display("IN0= %b, IN1= %b, IN2= %b, IN3= %b\n",IN0,IN1,IN2,IN3);
// choose IN0
S1 = 0; S0 = 0;
#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
// choose IN1
S1 = 0; S0 = 1;
#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
```

Example 5.2 Stimulus for Multiplexer –Cont’d

```
// choose IN2
S1 = 1; S0 = 0;
#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
// choose IN3
S1 = 1; S0 = 1;
#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
end
endmodule
```

Example 5.2 Stimulus for Multiplexer

- The output of the simulation is shown below
- Each combination of the select signals is tested

1. IN0= 1, IN1= 0, IN2= 1, IN3= 0
2. S1 = 0, S0 = 0, OUTPUT = 1
3. S1 = 0, S0 = 1, OUTPUT = 0
4. S1 = 1, S0 = 0, OUTPUT = 1
5. S1 = 1, S0 = 1, OUTPUT = 0

4-BIT RIPPLE CARRY FULL ADDER

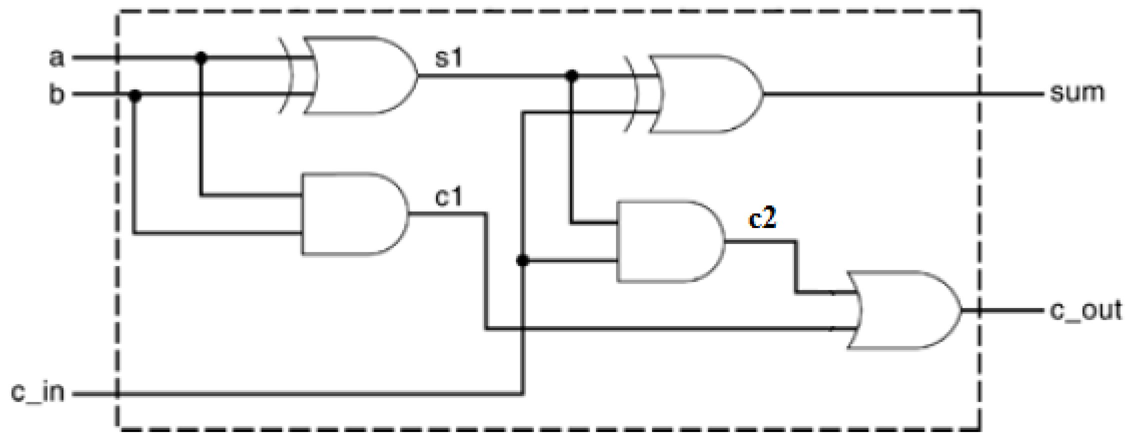
- In this example, we design a 4-bit full adder
- We use primitive logic gates, and we apply stimulus to the 4-bit full adder to check functionality
- For the sake of simplicity, we will implement a ripple carry adder
- The basic building block is a 1-bit full adder
- The mathematical equations for a 1-bit full adder are shown below

$$\text{sum} = (a \oplus b \oplus \text{cin})$$

$$\text{cout} = (a \cdot b) + \text{cin} \cdot (a \oplus b)$$

Example 7: Verilog Description for 1-bit Full Adder

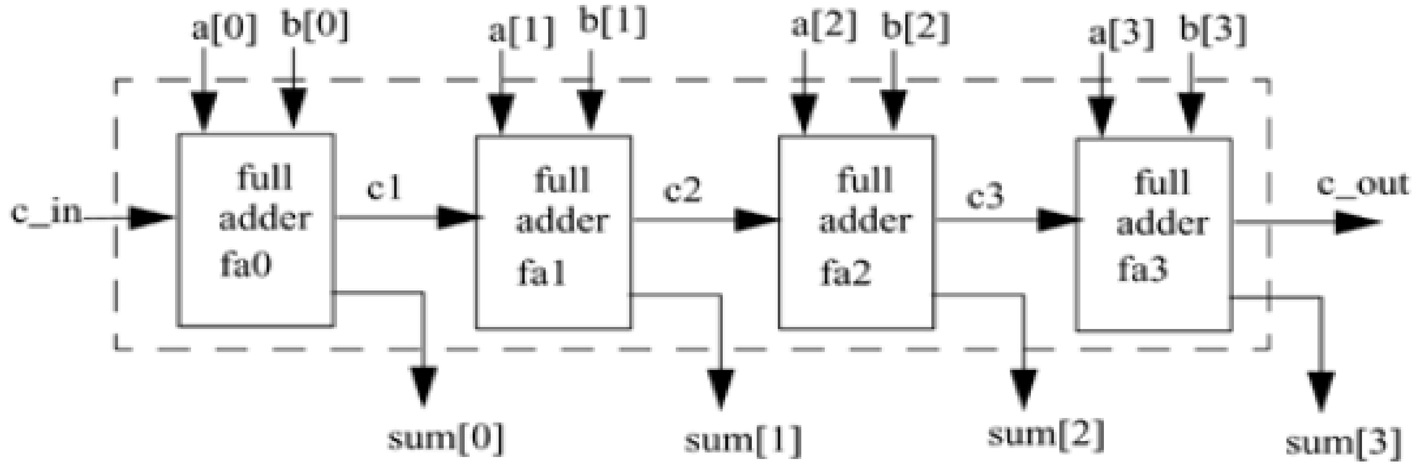
Figure 6. 1-bit Full Adder



```
// Define a 1-bit full adder
module fulladd(sum, c_out, a, b, c_in);
// I/O port declarations
output sum, c_out;
input a, b, c_in;
// Internal nets
wire s1, c1, c2;
// Instantiate logic gate primitives
xor (s1, a, b);
and (c1, a, b);
xor (sum, s1, c_in);
and (c2, s1, c_in);
or (c_out, c2, c1);
endmodule
```

A 4-BIT RIPPLE CARRY FULL ADDER CONSTRUCTED FROM FOUR 1-BIT FULL ADDERS

Figure 7. 4-bit Ripple Carry Full Adder



A 4-BIT RIPPLE CARRY FULL ADDER CONSTRUCTED FROM FOUR 1-BIT FULL ADDERS –Cont'd

- This structure can be translated to Verilog as shown in Example 8.1
- Note that the port names used in a 1-bit full adder and a 4-bit full adder are the same but they represent different elements.
- The element sum in a 1-bit adder is a scalar quantity and the element sum in the 4-bit full adder is a 4-bit vector quantity
- Verilog keeps names local to a module. Names are not visible outside the module unless hierarchical name referencing is used
- Also note that instance names must be specified when defined modules are instantiated, but when instantiating Verilog primitives, the instance names are optional

EXAMPLE 8.1: VERILOG DESCRIPTION FOR 4-BIT RIPPLE CARRY FULL ADDER

```
// Define a 4-bit full adder
module fulladd4(sum, c_out, a, b, c_in);
// I/O port declarations
output [3:0] sum;
output c_out;
input[3:0] a, b;
input c_in;
// Internal nets
wire c1, c2, c3;
// Instantiate four 1-bit full adders.
fulladd fa0(sum[0], c1, a[0], b[0], c_in);
fulladd fa1(sum[1], c2, a[1], b[1], c1);
fulladd fa2(sum[2], c3, a[2], b[2], c2);
fulladd fa3(sum[3], c_out, a[3], b[3], c3);
endmodule
```

Example 8.2 Stimulus for 4-bit Ripple Carry Full Adder

```
// Define the stimulus (top level module)
module stimulus;
// Set up variables
reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;
// Instantiate the 4-bit full adder. call it FA1_4
fulladd4 FA1_4(SUM, C_OUT, A, B, C_IN);
// Set up the monitoring for the signal values
initial
begin
$monitor($time," A= %b, B=%b, C_IN= %b, --- C_OUT= %b, SUM= %b\n", A, B, C_IN, C_OUT, SUM);
end
// Stimulate inputs
initial
begin
A = 4'd0; B = 4'd0; C_IN = 1'b0;
#5 A = 4'd3; B = 4'd4;
#5 A = 4'd2; B = 4'd5;
#5 A = 4'd9; B = 4'd9;
#5 A = 4'd10; B = 4'd15;
#5 A = 4'd10; B = 4'd5; C_IN = 1'b1;
end
endmodule
```

Example 8.2 Stimulus for 4-bit Ripple Carry Full Adder –Cont’d

- The output of the simulation is shown below

1. 0 A= 0000, B=0000, C_IN= 0, --- C_OUT= 0, SUM= 0000
2. 5 A= 0011, B=0100, C_IN= 0, --- C_OUT= 0, SUM= 0111
3. 10 A= 0010, B=0101, C_IN= 0, --- C_OUT= 0, SUM= 0111
4. 15 A= 1001, B=1001, C_IN= 0, --- C_OUT= 1, SUM= 0010
5. 20 A= 1010, B=1111, C_IN= 0, --- C_OUT= 1, SUM= 1001
6. 25 A= 1010, B=0101, C_IN= 1,, C_OUT= 1, SUM= 0000