Digital Logic Design and HDL Coding

Verilog® HDL

logic modeling

Today’s program agenda

- Expressions and operand
- Operators
- Logic Gates
- Three state gates
- Gate-level modeling
- User defined Primitives (UDPs)

Learning objectives

- What is an expression and operand? How operands, operators are used to form an expression? Understand the types of operators, their precedence in modeling logic logic circuits.
- Learn about digital logic gates, its truth tables, and how they are instantiated in a verilog design.
- Understand function of three state gates, why there are needed in modeling circuits, and how they are used in verilog design.
- Learn about rules to define UDPs, types of UDPs and how UDPs are used to model digital logic.

Expressions, Operators and Operand

- Expressions are simply a combination of operands and operators which produce a result.
- Operands provide the data evaluated by the expression.
- Operators act on operands to produce desired results.

Example: Y = A + B - C

In the above statement A + B - C forms the expression. A, B, and C are the operands and + and - are the arithmetic operators.

Operands

An operand can be any of of the verilog Data Types

- number (including real)

```
Example:   real x, y, z; // x, y and z declared as real

           z = x - y ; // x and y are real operands
```

- net, net bit-select, net part select

```
Example:   wire  [7:0]  temp1; // temp1  is declared as net data type

           wire  [7:0]  temp2; // temp2  is declared as net data type

wire  temp_out[3:0]  = temp1[3:0] ^ temp2 [3:0]; //temp1[3:0] and temp2[3:0] are part-select net operands
```

Operands continued

- register, register bit-select • register part-select

Example:

```
reg [15:0] count1,count2;       // count1/count2 are declared as reg data type
reg [7 :0] count_out;     // count_out is declared as reg data type
count_out[7:0] = count1[7:0] & count2[7:0];      //count11[7:0] and count2[7:0] are part-select
                                                  // register operands
```

- integer

Example:

```
integer count, final-count; // Both count and final_count are declared as integer  data type
final-count = count + l;     //count is an integer operand
```

Operands continued

- functions

Example:

```
wire   a; // a is defined as wire data type
assign a = b & c & check_bc(b, c); // check_bc is a function

function check_bc; // function definition
input b,c;
check_bc = b ^ c;
endfunction
```

An operand can be a memory element or system defined function that returns any of the above operand types discussed.

Verilog Operators

Verilog operators operate on several data types (operands) to produce an output. Verilog supports many different types of operators and are listed below

- Arithmetic operator
- logical operator
- Relational operator
- Equality operator
- Bitwise operator
- Reduction operator
- Shift operator
- Concatenation and Replication operator
- Conditional operator

Verilog Operators

Arithmetic Operator: There different arithmetic operators in Verilog listed in table below.The + and - can be used as either unary (-x) or binary (x-y) operators.

Operator symbol	Operation performed	Number of operands
+	add	two
-	subtract	two
*	multiply	two
/	divide	two
%	modulus	two
**	power	two

Verilog Operators

Arithmetic Operator example:

```
Assume a = 4’b0011; b = 4’b0100; d = 6; e = 4; f = 2;

a + b  //add a and b; evaluates to 4’b0111

b - a         //subtract a from b; evaluates to 4’b0001

a * b  //multiply a and b; evaluates to 4’b1100

d / e         /*divide d by e, evaluates to 4’b0001.

               Truncates fractional part */

e ** f       // to the power f, evaluates to 4’b1111
```

Modulus operator yields the remainder from division of two numbers

```
        3 % 2; // evaluates to 1

        16 % 4; // evaluates to 0

        -7 % 2; /* evaluates to -1,
takes sign of first operand */

        7 % -2; /* evaluates to 1, takes
sign of first operand */
```

Verilog Operators

Logical Operator: Logical operators evaluates to a 1 bit value(0: if relation is false, 1: if relation is true, or x: if relation is unknown).

Operator symbol	Operation performed	Number of operands
!	logical negation	one
&&	logical and	two
||	logical or	two

Example1: a = 2’b11 and b = 2’b00

(a && b) //evaluates to zero   (!a) // evaluates to 0

(b || a) //evaluates to one   (!b) // evaluates to 1

Example2: a = 2’b0x; b = 2’b10;

(a && b) // evaluates to x

Note: If any operand bit is x or z, it evaluates to x

Verilog Operators

Example with expression:

if ((a == 4) && (b == 5)) //evaluates to 1 only if both a == 4 and b ==5 are true

if ((4’b1100 > 4’b1011) || (4’b0111 == 4’b1000)) //evaluates to 1 if any one comparison is true

Logical operators are typically used in conditional statements (if ... else) , since they work with expressions.An example of if ….. else statement is shown below

```
wire [3:0]       x, y, z;    // x, y, and z are vectors and wire data type
reg  a;                   // variable a is declared as register data type scalar
      if ((x ==y) && (z))   // a evaluates to 1 if (x == y) and (z) is non-zero is true
              a = 1’b1;
      else
              a = 1’b0; // a is 0 if (x == y) and (z) is false
```

Verilog Operators

Relational Operator: Relational operators evaluates to a 1 bit value (0: if relation is false, 1: if relation is true, or x: if relation is unknown).

Operator symbol	Operation performed	Number of operands
>	greater than	two
<	less than	two
>=	greater than or equal	two
<=	less than or equal	two

Verilog Operators

Relational Operator Example:

Assume a = 6, b = 4, and x = 4’b1001, y = 4’b1110,v = 4’b1xxx and w = 4’b1z10

```
a <= b    //evaluates to logical zero

a > b     //evaluates to logical one

y >= x    //evaluates to logical 1

y < v     //evaluates to x if any bit is x

y > w     //evaluates to x if any bit is z
```

Verilog Operators

Equality Operator: Equality operators return logical 1 if the expression is true, else 0. Operands are compared bit by bit, zero filling is done if operands are of unequal length. For the == and != operators, the result is x, if either operand contains an x or a z. For the === and !== operators, bits with x and z are included in the comparison and must match for the result to be true

Operator symbol	Operation performed	Number of operands
==	equality	two
!=	inequality	two
===	case equality	two
!==	case inequality	two

Verilog Operators

Example of equality Operator:

let a = 5, b = 4, and w = 4’b1010, y = 4’b1101, z = 4’b1xxz, m = 4’b1xxz, n =’b1xxx

```
a == b    //evaluates to logical 0
w != y    //evaluates to logical 1
w == z    //evaluates to x
z === m //evaluates to logical 1
z === n  //evaluates to logical 0
m !== n  //evaluates to logical 1
```

Verilog Operators

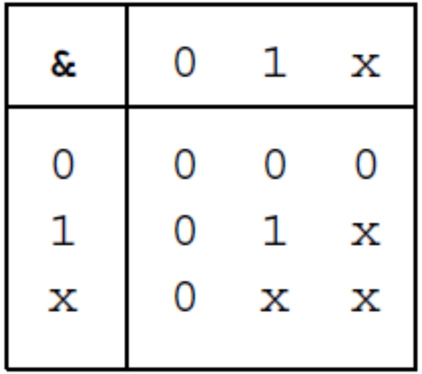
Bitwise Operator: Perform bit-by-bit operation on two operands (except ~). Mismatched length operands are zero extended. A “z” is treated as an “x” in bitwise operation.

Operator symbol	**Operation performed**	**Number of operands**
~	bitwise negation	one
&	bitwise and	two
|	bitwise or	two
^	bitwise xor	two
~^ or ^~	bitwise xnor	two

Verilog Operators

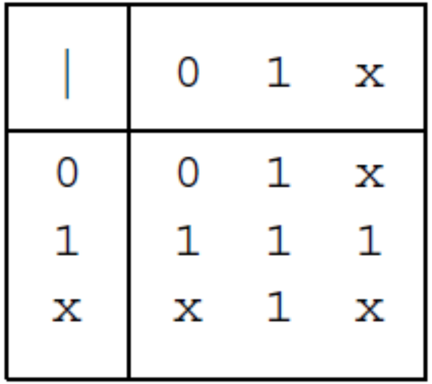
Truth table for bitwise operators are shown below.

bitwise and (&)



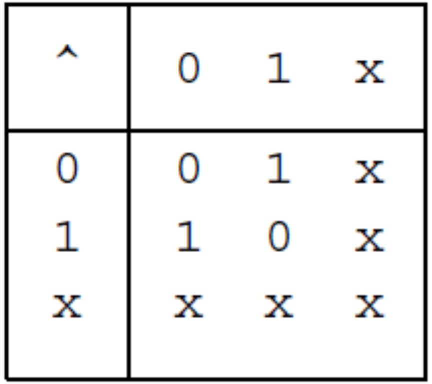
&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bitwise or(|)



|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

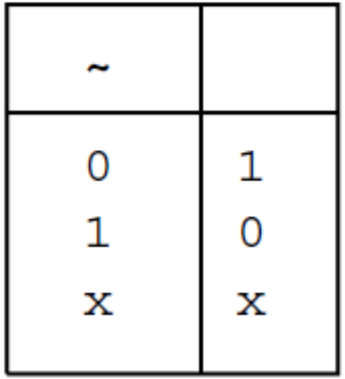
bitwise xor(^)



^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

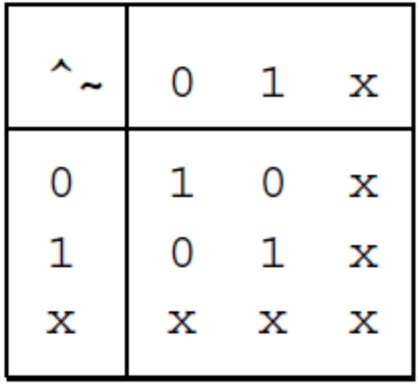
Verilog Operators

bitwise negation(~)



~	
0	1
1	0
x	x

bitwise xnor(~^ or ^~)



^~	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Verilog Operators

Example of bitwise operator:

```
Let a = 4’b1010, b = 4’b0000

~a          //negation, result is 4’b0101

a & b       // bitwise and, result is 4’b0000

a | b       // bitwise or, result is 4’b1010

a ^ b       // bitwise xor, result is 4’b1010

a ~^b       //bitwise xnor, result is 4’b0101
```

Verilog Operators

Reduction Operator: Performs a bitwise operation on all bits of the operand and returns a 1-bit result.Works from right to left, bit by bit and operates only on one operand.

Operator symbol	**Operation performed**	**Number of operands**
&	reduction and	one
~&	reduction nand	one
|	reduction or	one
~|	reduction nor	one
^	reduction xor	one
~^ or ^~	reduction xnor	one

Verilog Operators

Example of reduction operator:

```
let x = 4'b1010

&x   //equivalent to 1 & 0 & 1 & 0 and the result is 1'b0

|x   //equivalent to 1 | 0 | 1 | 0 and the result is 1'b1

^x   //equivalent to 1 ^ 0 ^ 1 ^ 0 and the result is 1'b0
```

more examples:

```
&  4'b1001 = 0, &  4'bx111 = x, &  4'bz111 = x, ~& 4'b1001 = 1, ~& 4'bx001 = 1, ~& 4'bz001 = 1
 |  4'b1001 = 1, |  4'bx000 = x, |  4'bz000 = x, ~| 4'b1001 = 0, ~| 4'bx001 = 0, ~| 4'bz001 = 0
 ^  4'b1001 = 0, ^  4'bx001 = x, ^  4'bz001 = x, ~^ 4'b1001 = 1, ~^ 4'bx001 = x, ~^ 4'bz001 = x
```

Verilog Operators

Shift Operator: Shift operator shifts a vector operand left or right by a specified number of bits, filling vacant bit positions with zeros.Shifts operations do not wrap around.

Operator symbol	Operation performed	Number of operands
>>	right shift	two
<<	left shift	two

Example: Let X = 4’b1100

Y = X >> 1; // Y is 4’b0110. Shift right 1 bit. 0 filled in MSB position

Y = X << 1; // Y is 4’b1000. Shift left 1 bit. 0 filled in LSB position

Y = X << 2; // Y is 4’b0000. Shift left 2 bits. 0 filled in LSB position

Verilog Operators

Concatenation Operator: Provides a mechanism to to append multiple operands. The operands must be sized (unsized operands are not allowed) and expressed as operands in braces separated by commas. Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constant.

Operator symbol	**Operation performed**	**Number of operands**
{ }	concatenation	any number

Example: Let A = 1’b1, B = 2’b00, C = 2’b10, D = 3’b110

Y = {B, C}; // Result Y is 4’b0010

Y = {A, B, C, D, 3’b001}; // Result Y is 11’b10010110001

Y = {A, B[0], C[1]}; // Result Y is 3’b101

Verilog Operators

Replication Operator: Repetitive concatenation of the same number can be expressed by using a replication constant. A replication constant specifies how many times to replicate the number inside the brackets ({ }).

Operator symbol	Operation performed	Number of operands
{ { } }	replication	any number

Example:

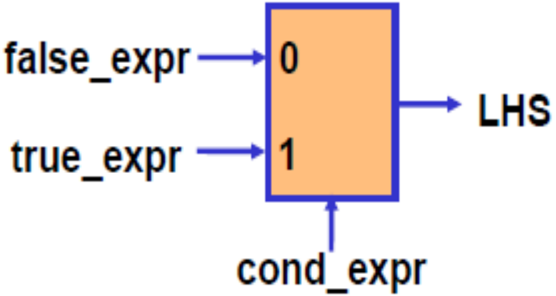
```
reg A;
reg [1:0] B, C;
reg [2:0] D;
wire Y[9:0];
A = 1’b1; B = 2’b00; C = 2’b10; D = 3’b110;
Y = { 4{A}}; // Result Y is 4’b1111
Y = { 4{A}, 2{B}}; // Result Y is 8’b11110000
Y = { 4{A}, 2{B}, C}; // Result Y is 10’b1111000010
```

Verilog Operators

Conditional Operator: The conditional operator (?:) takes three operands.

<LHS> = <conditional_expr> ? <true_expr> : <false expr>

- The conditional expression (condition_expr) is first evaluated.
- If the result is true (logical “1”), then the true_expr is evaluated
- If the result is false (logical “0”), then the false_expr is evaluated
- If the result is “x” (ambiguous), then both true_expr and false_expr are evaluated and their results are compared, bit by bit, to return for each bit position an “x” is the bits are different and the value of the bits if they are the same



Verilog Operators

The action of a conditional operator is similar to a multiplexer. Alternately, it can be compared to the if-else expression.Conditional Operator is an ideal way to model a multiplexer or tri-state buffer.

Example:

```
// Model functionality of a 2-to-1 mux

assign out = sel ? in1 : in0;

// Model functionality of a tristate buffer

assign data_bus = bus_enable ? data_out : 35’bz;
```

Verilog Operators

Conditional operators can be nested. Each true_expr or false_expr can itself be a conditional operation.

Example:

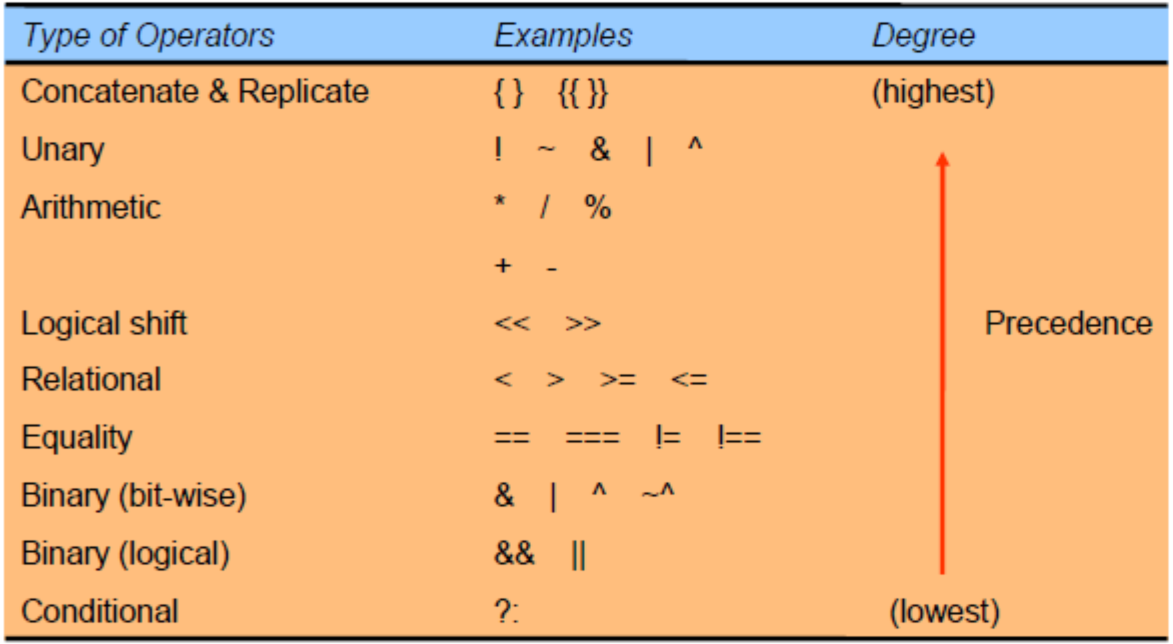
```
// Check that A==3 and control are the two select signals of

// 4-to-1 mux with n, m, x, y as the inputs and out as the output signal

assign out = (A == 3) ? ( control ? x : y ) : ( control ? m : n );
```

Verilog Operators

Operator Precedence: If no parentheses are used to separate parts of expressions, Verilog enforces the following precedence. Operators listed in Table below are in order from highest precedence to lowest precedence. It is recommended that parentheses be used to separate expressions except in case of unary operators or when there is no ambiguity.



Type of Operators	*Examples*	*Degree*
Concatenate & Replicate	{ } {{ }}	(highest)
Unary	! ~ & | ^	
Arithmetic	* / %	
	+ -	
Logical shift	<< >>	Precedence
Relational	< > >= <=	
Equality	== === != !==	
Binary (bit-wise)	& | ^ ~^	
Binary (logical)	&& ||	
Conditional	?:	(lowest)

Gate level modeling

Verilog is both a behavioral and a structural language. There are four levels of abstraction used to describe hardware in verilog. They are **Gate level modeling, switch level modeling**, **data flow modeling** and **behavioral modeling**.

Gate Level Modeling(Structural modeling): At Gate level, the circuit is described in terms of logic gates (and, nand, or, nor, xor, xnor gates) and interconnections between these gates. Gate level modeling are built:

- Using verilog built-in gate primitives (there are 14 different gate primitives in verilog: 8 of them are for logic functions (and, nand, or, nor, xor, xnor, buf and not gates), 4 of them are for modeling tri-state signals (bufif0, bufif1, notif0, and notif1), and 2 of them (pullup and pulldown) are enhance signal strength).These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition.
- Using User Defined Primitives (defined in terms of truth tables)

Gate level modeling

There are two classes of basic gates:. These basic gates are called verilog primitives.

and / or type Gates

- and
- nand
- or
- nor
- xor
- xnor

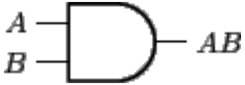
buf/not type gates:

- buf
- not

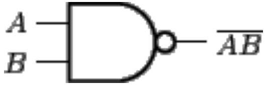
Gate level modeling

and/or type Logic gates have one output and one or more inputs. The first terminal in the terminal list connects to the gate's output and all other terminals connect to its inputs. The output of a gate is evaluated as soon as one of the inputs changes. Symbol for logic gates are shown below.

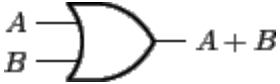
and



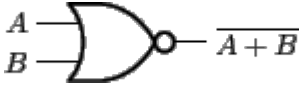
nand



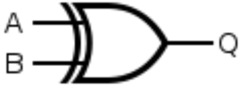
or



nor

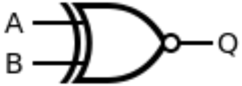


xor



$$Q = A \oplus B$$

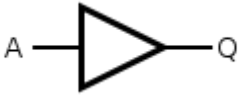
xnor



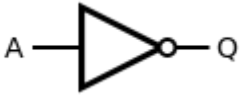
$$Q = \overline{A \oplus B}$$

buf/not type logic gates have one input and one or more output.The last terminal in the port list is connected to the input. Other terminals are connected to the outputs. We will discuss gates that have one input and one output.

buffer

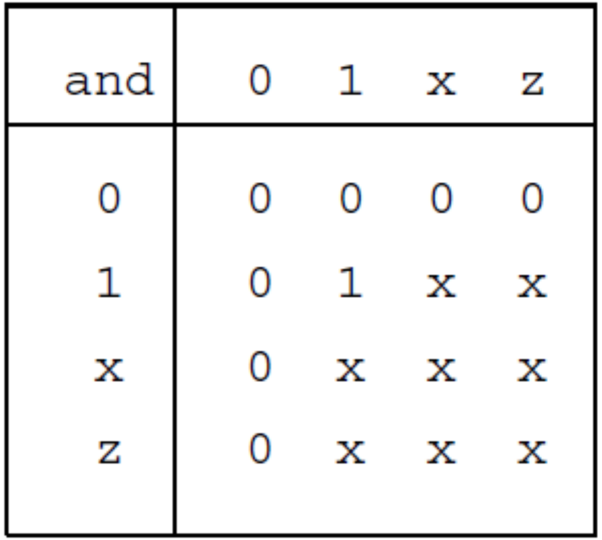


not

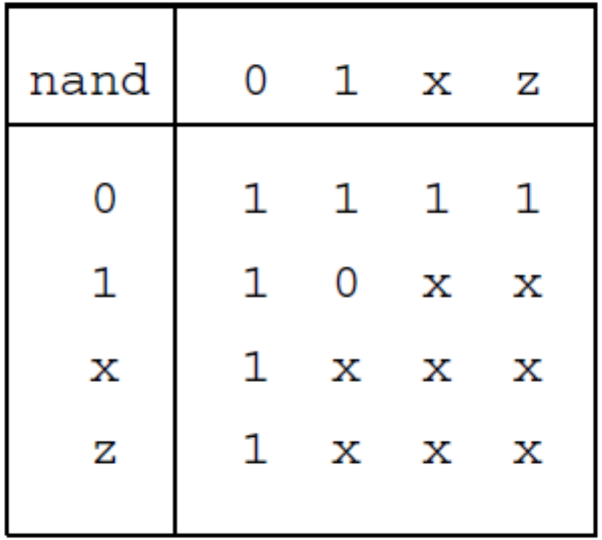


Gate level modeling

The truth table for and/nand gates are shown below.



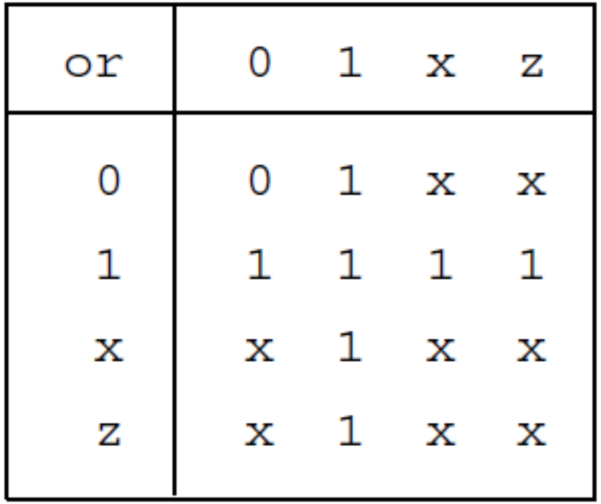
and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x



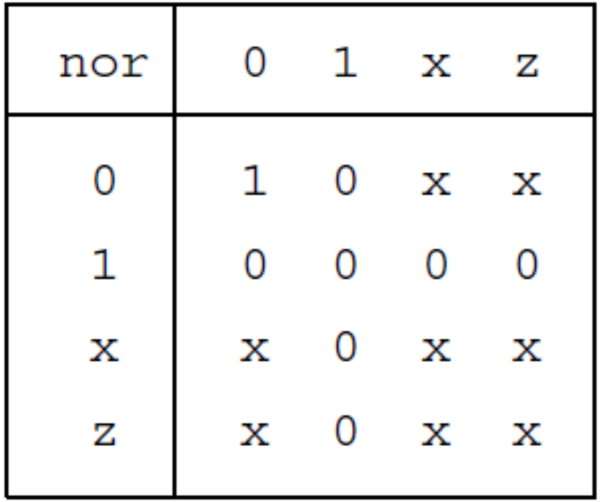
nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

Gate level modeling

Truth table for or/nor gates are shown below.



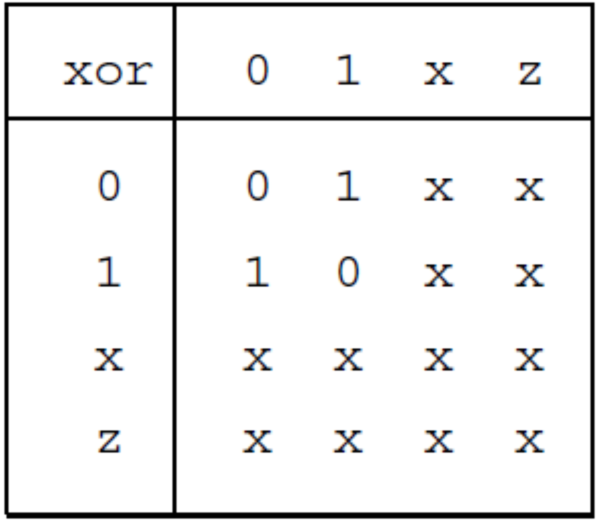
or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x



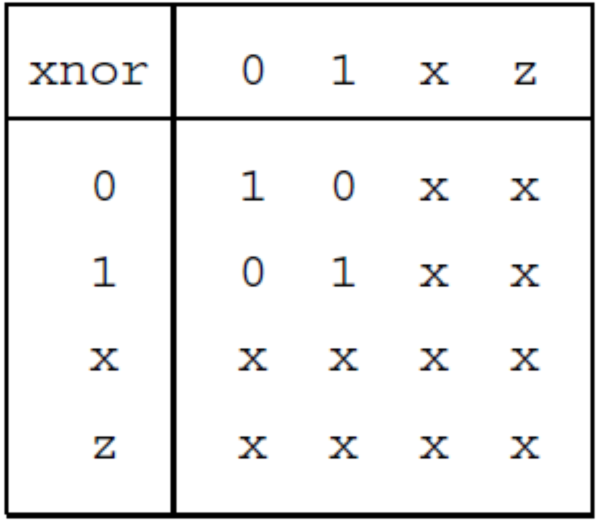
nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

Gate level modeling

Truth table for xor/xnor gates are shown below.



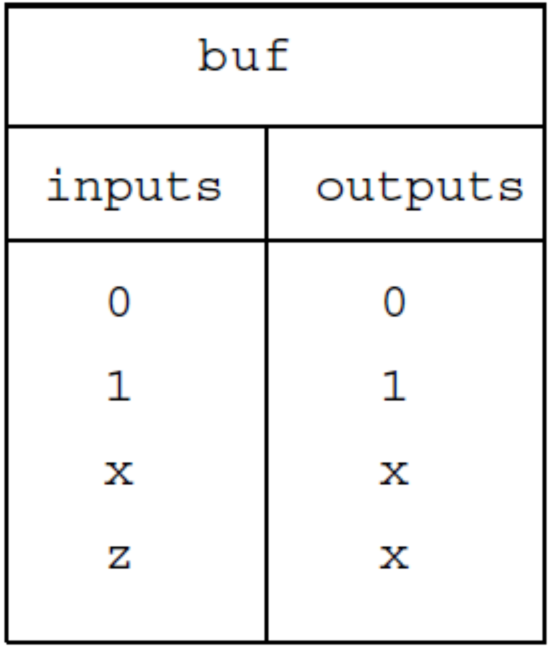
xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x



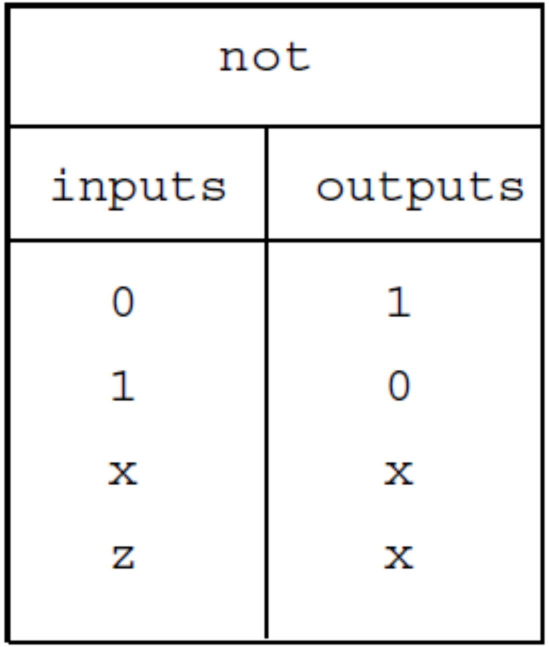
xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Gate level modeling

The truth table for buf/not gate are shown below.



buf	
inputs	outputs
0	0
1	1
x	x
z	x



not	
inputs	outputs
0	1
1	0
x	x
z	x

Gate level modeling

Example: Typical Verilog gate instantiation is:

and #delay instance-name (out, in1, in2, in3, …); /* Note: #delay and instance-name are optional */

```
and #5 g1 (f1, a, g1); //2 input and gate with delay

      // basic buffer and not gate instantiations.

            buf bl(OUT1, IN);

            not nl(OUT1, IN);
```

```
// More than two outputs from buffer

      buf bl_2out(OUTI, OUT2, IN);

// gate instantiation without instance name

      not (OUT1, IN); // legal gate
      instantiation
```

Gate level modeling

More Example of gate instantiation:

```
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 na1-3input (OUT, IN1, IN2, IN3 ) ;

// gate instantiation without instance name

and (OUT, IN1, IN2); // legal gate instantiation
```

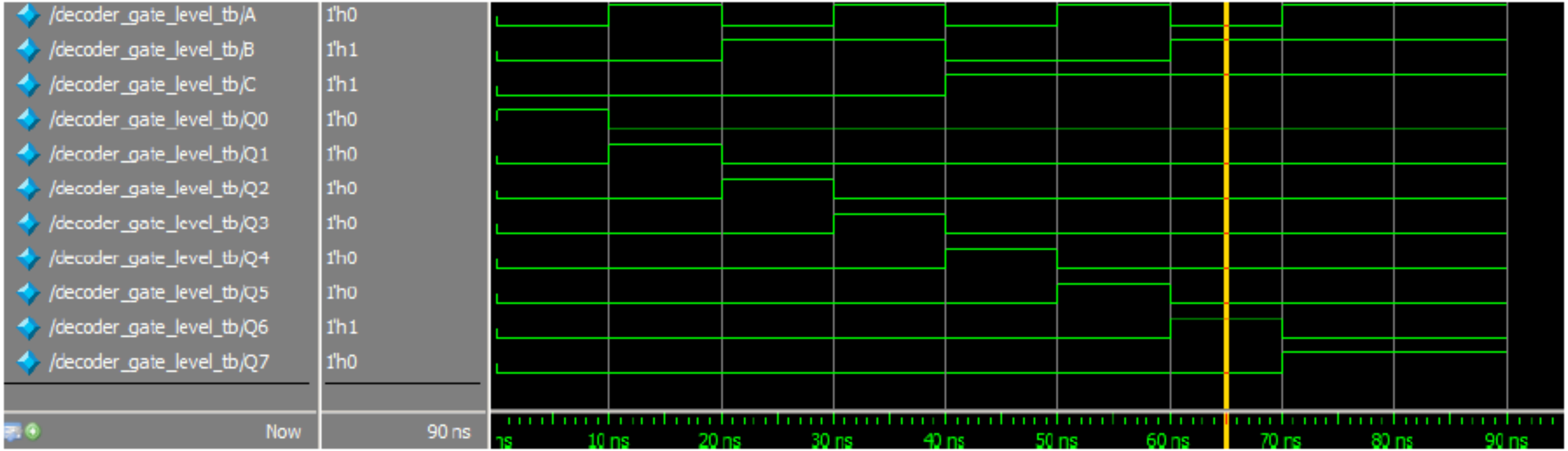
3-to-8 line decoder gate level example

```
// module decoder
module decoder(Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7,A,B,C);  // module input/output port list
output Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7; // outputs
input A,B,C; // inputs
wire s0,s1,s2; // s0 s1 and s2 are not operation of A, B and C
not (s0,A);
not (s1,B);
not (s2,C);
and (Q0,s0,s1,s2);
and (Q1,A,s1,s2);
and (Q2,s0,B,s2);
and (Q3,A,B,s2);
and (Q4,s0,s1,C);
and (Q5,A,s1,C);
and (Q6,s0,B,C);
and (Q7,A,B,C);
endmodule
```

3-to-8 line decoder gate level example test bench

```
module stimulus; // Set up variables
reg A, B, C;
decoder deco1(Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7,A,B,C); // decoder instantiated
initial
        begin $monitor($time," C= %b, B=%b,A=%b,Q=%b%b%b%b%b%b%b%b\n",C,B,A,Q0,Q1,Q2,
        Q3,Q4,Q5,Q6,Q7); end
initial
        begin //Provide input vectors
                C=1'b0; B =1'b0; A = 1'b0;
                #10 C =1'b0; B = 1'b0; A= 1'b1;
                #10 C =1'b0; B = 1'b1; A= 1'b0;
                #10 C =1'b0; B = 1'b1; A= 1'b1;
                #10 C =1'b1; B = 1'b0; A= 1'b0;
                #10 C =1'b1; B = 1'b0; A= 1'b1;
                #10 C =1'b1; B = 1'b1; A= 1'b0;
                #10 C =1'b1; B = 1'b1; A= 1'b1;
                end
                endmodule
```

```
$monitor display output:     # 0 C= 0, B=0, A=0,   Q=10000000
                             # 10 C= 0, B=0, A=1, Q=01000000
                             # 20 C= 0, B=1, A=0, Q=00100000
                             # 30 C= 0, B=1, A=1, Q=00010000
                             # 40 C= 1, B=0, A=0, Q=00001000
                             # 50 C= 1, B=0, A=1, Q=00000100
                             # 60 C= 1, B=1, A=0, Q=00000010
                             # 70 C= 1, B=1, A=1, Q=00000001
```

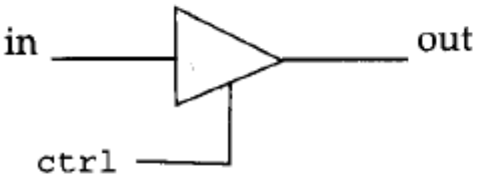


```
/decoder_gate_level_tb/A     1'h0
/decoder_gate_level_tb/B     1'h1
/decoder_gate_level_tb/C     1'h1
/decoder_gate_level_tb/Q0    1'h0
/decoder_gate_level_tb/Q1    1'h0
/decoder_gate_level_tb/Q2    1'h0
/decoder_gate_level_tb/Q3    1'h0
/decoder_gate_level_tb/Q4    1'h0
/decoder_gate_level_tb/Q5    1'h0
/decoder_gate_level_tb/Q6    1'h1
/decoder_gate_level_tb/Q7    1'h0
Now                          90 ns
```

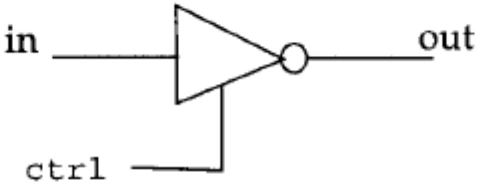
Tri-state gate level modeling

Tri-state Buffer: Gates with an additional control signal on buf and not gates are called tri-state gates. This type of device has two logic state inputs, “0” or a “1” but can produce three different output states, “0”, “1” or “ Hi-Z ” which is why it is called a “3-state” device. The 4 types of tri-state gates and described using verilog keywords: **bufif0 bufif1 notif1 notif0**

Active High:

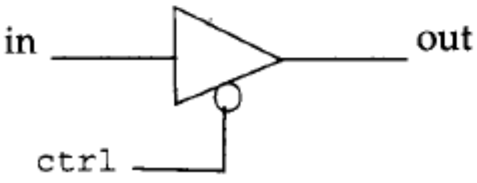


bufif1

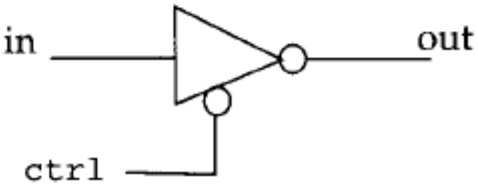


notif1

Active Low:



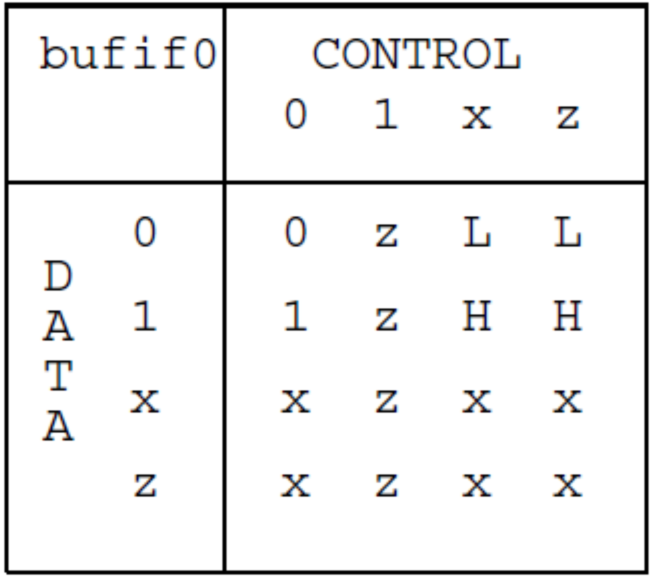
bufif0



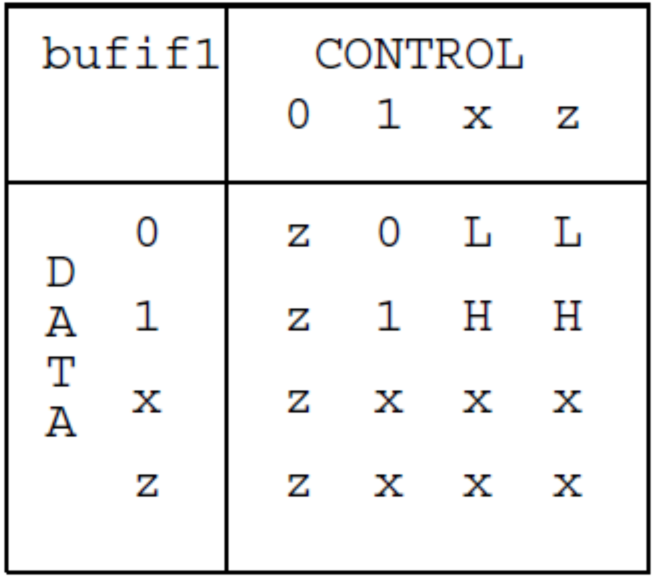
notif0

Tri-state gate level modeling

The truth table for tri-state gates are shown below:



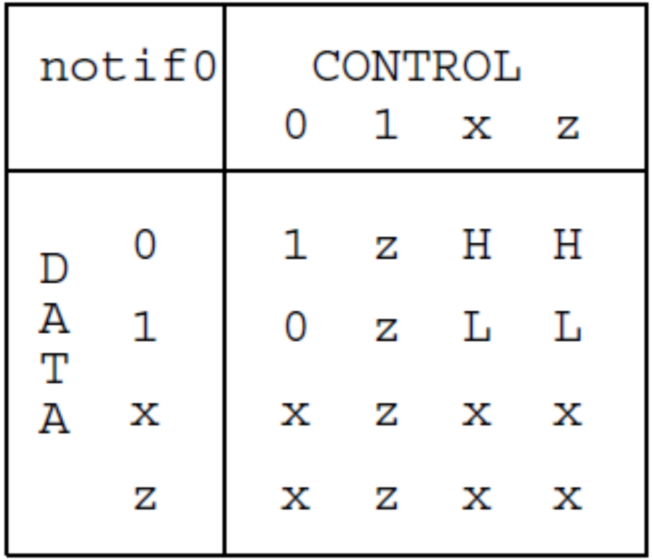
bufif0		CONTROL			
		0	1	x	z
DATA	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x



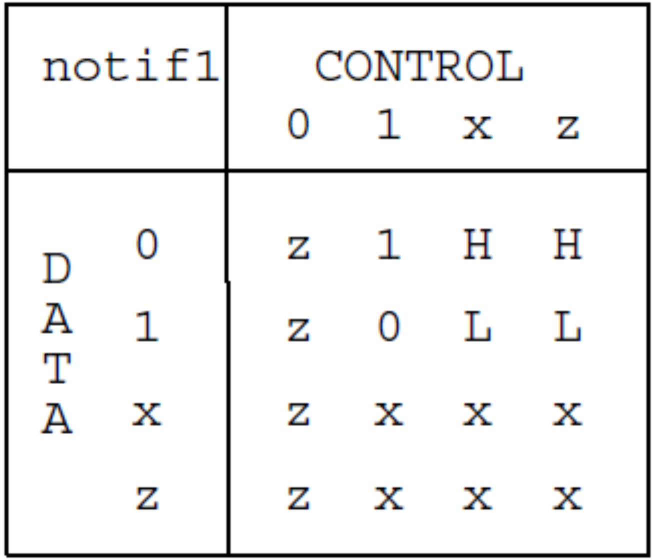
bufif1		CONTROL			
		0	1	x	z
DATA	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

Tri-state gate level modeling

The truth table for tri-state gates are shown below:



notif0		CONTROL			
		0	1	x	z
DATA	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x



notif1		CONTROL			
		0	1	x	z
DATA	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

Note: Some combinations of data input values and control input values cause these gates to output either of two values, without a preference for either value. These gates' logic tables include two symbols representing such unknown results. The symbol L represents a result which has a value of 0 or Z. The symbol H represents a result which has a value of 1 or Z.

Tri-state gate level modeling

Tri-state gate example: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 drivers are designed to drive the signal on mutually exclusive control signals.

```
//Instantiation of bufif gates.

bufif1 b1 (out, in, ctrl) ;

bufif0 b0 (out, in, ctrl) ;

//Instantiation of notif gates

notif1 n1 (out, in, ctrl) ;

notif0 n0 (out, in, ctrl) ;
```

Pullup and Pulldown sources

Declarations of these sources begin with one of the following keywords: **pullup pulldown**

A strength specification follows the keyword and an optional identifier follows the strength specification. A terminal list completes the declaration. A pullup source places a logic value of 1 on the nets listed in its terminal list. A pulldown source places a logic value of 0 on the nets listed in its terminal list. The signals that these sources place on nets have pull strength in the absence of a strength specification. There are no delay specifications for these sources because the signals they place on nets continue throughout simulation without variation.

Example: The following example declares two pullup instances:

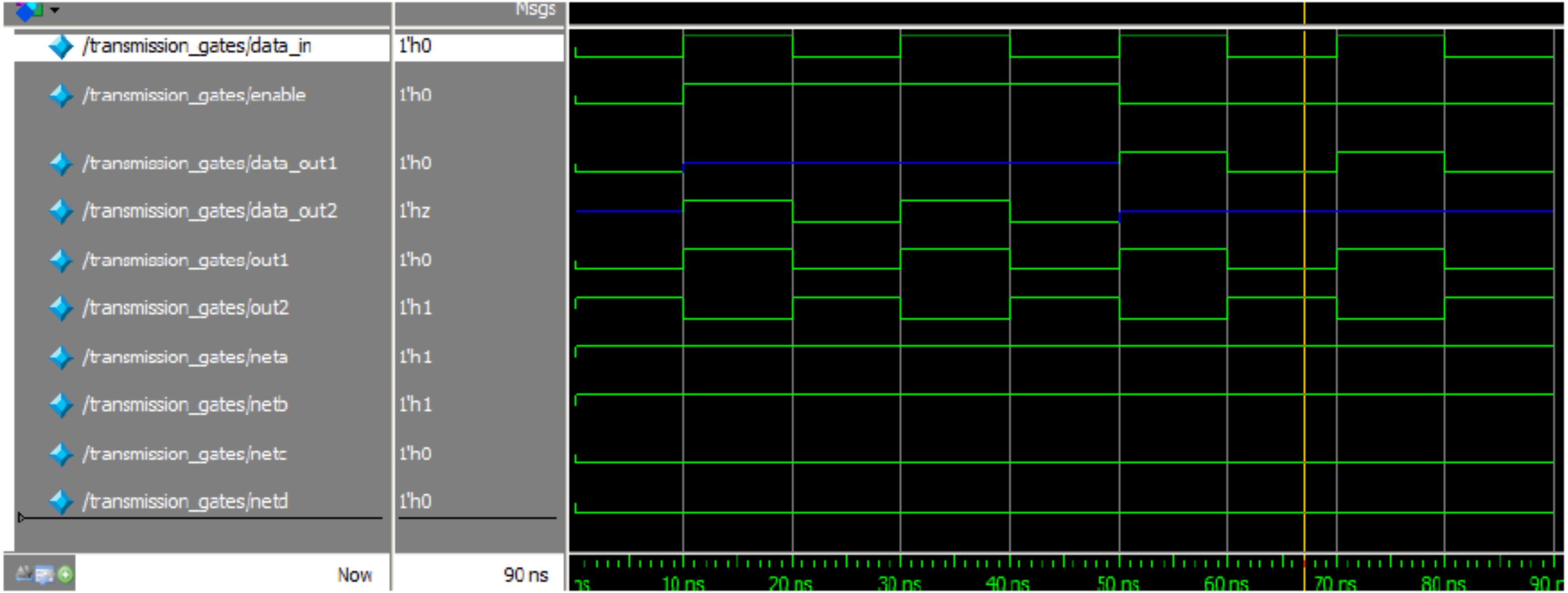
```
pullup (strong1, strong0) (neta), (netb);
```

In this example, one gate instance drives neta, the other drives netb.

tri-state gate example

```
module tristate_gates();
reg enable, data_in; //inputs defined as reg
wire data_out1,data_out2, out1, out2; //outputs defined as net data type
wire neta, netb, netc, netd; //internal nets
bufif0 U1(data_out1,data_in, enable); // active low tri state buffer instantiation
bufif1 U2(data_out2,data_in, enable); // active high tri state buffer instantiation
buf U3(out1,data_in); // buffer gate instantiation
not U4(out2,data_in); // not gate instantiation
pullup (strong0, strong1) pup (neta), (netb); // pullup will place 1 on neta and netb
pulldown (strong0, strong1) pud (netc), (netd); // pulldown will place 0 on netc and netd
Initial
        begin
                $monitor($time,“ data_in=%b enable=%b out1=%b out2= b data_out1=%b,data_out2=%b",
                data_in, enable, out1, out2, data_out1, data_out2);
                enable = 0; data_in = 0;
                #10 enable = 1;
                #40 enable = 0;
                #40 $stop;
        end
always #10 data_in = ~data_in;
endmodule
```

transmission gate simulation output



	Msgs
/transmission_gates/data_in	1'h0
/transmission_gates/enable	1'h0
/transmission_gates/data_out1	1'h0
/transmission_gates/data_out2	1'hz
/transmission_gates/out1	1'h0
/transmission_gates/out2	1'h1
/transmission_gates/neta	1'h1
/transmission_gates/netb	1'h1
/transmission_gates/netc	1'h0
/transmission_gates/netd	1'h0
Now	90 ns

transmission gate $monitor display

```
0 data_in=0 enable=0 out1=0 out2= b data_out1=1,data_out2=0z
# 10 data_in=1 enable=1 out1=1 out2= b data_out1=0,data_out2=z1
# 20 data_in=0 enable=1 out1=0 out2= b data_out1=1,data_out2=z0
# 30 data_in=1 enable=1 out1=1 out2= b data_out1=0,data_out2=z1
# 40 data_in=0 enable=1 out1=0 out2= b data_out1=1,data_out2=z0
# 50 data_in=1 enable=0 out1=1 out2= b data_out1=0,data_out2=1z
# 60 data_in=0 enable=0 out1=0 out2= b data_out1=1,data_out2=0z
# 70 data_in=1 enable=0 out1=1 out2= b data_out1=0,data_out2=1z
# 80 data_in=0 enable=0 out1=0 out2= b data_out1=1,data_out2=0z
```

UDPs

Verilog provides a standard set of built-in primitives, such as and, nand, or, nor, and not, as a part of the language. Verilog also provides the ability to define User-Defined Primitives (UDP). UDPs are instantiated exactly like gate-level primitives.

There are two types of UDPs:

- combinational
- sequential.

Combinational UDPs are defined where the output is solely determined by a logical combination of the inputs. A good example is a 8-to-1 multiplexer, full adder, decoder/encoder etc.

Sequential UDPs take the value of the current inputs and the current output to determine the value of the next output. The value of the output is also the internal state of the UDP. Good examples of sequential UDPs are latches and flip-flops.

UDPs Syntax

Syntax for defining User Defined Primitives are:

```
primitive primitive_name (output, input, input, ... ); /*UDP name and terminal list, only one output allowed */
 output terminal_declaration;
  input terminal_declaration;
  reg output_terminal; //Optional and only used for sequential UDP
  initial output_terminal = logic_value; // UDP initialization, used only for sequential UDPs

  table //UDP state table
        table_entry;

        table_entry;
  endtable
endprimitive
```

UDP Shorthand Symbol

Shorthand symbols	Meaning	Explanation
?	0, 1, x	Cannot be specified in an output field
b	0, 1	Cannot be specified in an output field
-	No change in state	Can be specified only in output field of a sequential UDP
r	(01)	rising edge of signal
f	(10)	falling edge of signal
p	(01), (0x) or (x1)	Potential rising edge of signal
n	(10), (1x) or (x0)	Potential falling edge of signal
*	(??)	Any value change in signal

UDP Rules

- *All terminals (input and output) must be scalar(1-bit)*
- *Only one scalar output terminal allowed and must appear first in terminal list.*
- *Output terminal is declared with the keyword **output** and declared as **reg** for sequential UDP because sequential UDPs store state. Inputs are declared with keyword **input**.*
- *UDP state table entries can contain values 0, 1, or x. UDPs do not support z values. z values passed to a UDP are treated as x values.*
- *The state in a sequential UDP can be initialized with an initial statement(Optional). Only 1, 0, and x values are used to define initial state. Default state is x.*
- *UDPs are defined at the same level as modules. UDPs cannot be defined inside modules. They can only be instantiated inside modules. UDPs are instantiated exactly like gate primitives.*
- *UDPs do not support **inout** ports.*

Combinational UDPs

Combinational UDPs take the inputs and produces output value by looking up the corresponding entry in the state table.The Syntax of state table entry of a combinational UDP looks like this

<input1> <input2> <inputN> : <output>

- The <input#> values in a state table entry must appear in the same order as they appear in the input terminal list.
- Inputs and output are separated by a " **:** ".
- A state table entry ends with a " **;** ".
- All possible combinations of inputs, where the output produces a known value, must be explicitly specified. Otherwise, if a certain combination occurs and the corresponding entry is not in the table, the output is x.

Combinational UDP example

```
//UDP for a 2-input AND gate

primitive udp_and2 (z1, x1, x2); //output z1 is listed first

input x1, x2; //input declaration
output z1;  //output declaration
//define state table
table //inputs are the same order as the input list
// x1 x2 : z1; comment is for readability
    0  0   :  0;
    0  1   :  0;
    1  0   :  0;
    1  1   :  1;
endtable
endprimitive
```

2x1 Combinational UDP Example

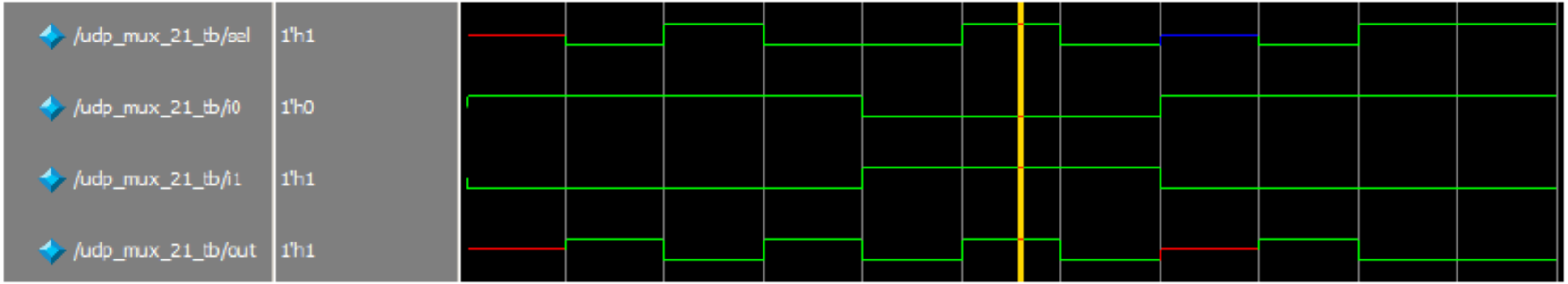
```
//2:1 mux combinational udp
primitive mux_21_udp(out, sel, i0, i1);
output out;
input sel, i0, i1;
table
// sel i0 i1 out
    0  0 ? : 0 ;
    0  1 ? : 1 ;
    1  ? 0 : 0 ;
    1  ? 1 : 1 ;
    ?  0 0 : 0 ;
    ?  1 1 : 1 ;
endtable
endprimitive
```

2x1 Combinational UDP Example test bench

```
module udp_mux_21_tb; // 2:1 mux test bench
reg sel, i0, i1;
wire out;
initial $monitor ("sel=%b, i0=%b, i1=%b, out=%b", sel, i0, i1, out);
initial
    begin
        i0 =1'b1; i1 =1'b0;    #10 sel =1'b0;
        #10 sel =1'b1;         #10 sel =1'b0;
        #10 i0 =1'b0; i1 =1'b1;
        #10 sel =1'b1;
        #10 sel =1'b0;
        #10 sel =1'bz;
        i0 =1'b1; i1 =1'b0;
        #10 sel =1'b0; #10 sel =1'b1;
        #20 $stop;
    end
mux_21_udp mux1(out, sel, i0, i1);//Udp instantiation
endmodule
```

2x1 mux simulation output

```
# sel=x, i0=1, i1=0, out=x
# sel=0, i0=1, i1=0, out=1
# sel=1, i0=1, i1=0, out=0
# sel=0, i0=1, i1=0, out=1
# sel=0, i0=0, i1=1, out=0
# sel=1, i0=0, i1=1, out=1
# sel=0, i0=0, i1=1, out=0
# sel=z, i0=1, i1=0, out=x
# sel=0, i0=1, i1=0, out=1
# sel=1, i0=1, i1=0, out=0
```



Sequential UDPs

Sequential UDPs differ from combinational UDPs in their definition and behavior. Sequential UDPs have the following differences: Syntax to write sequential UDP is shown below

<input1> <input2> <inputN> : <current state> : <next state>

- The output of a sequential UDP is always declared as a reg.
- An initial statement can be used to initialize output of sequential UDPs.
- There are three sections in a state table entry: inputs, current state, and next state. The three sections are separated by a colon (:) symbol.
- The input specification of state table entries can be in terms of **input levels** or **edge** transitions.
- The current state is the current value of the output register. The next state is computed based on inputs and the current state. The next state becomes the new value of the output register.
- All possible combinations of inputs must be specified to avoid unknown output values.

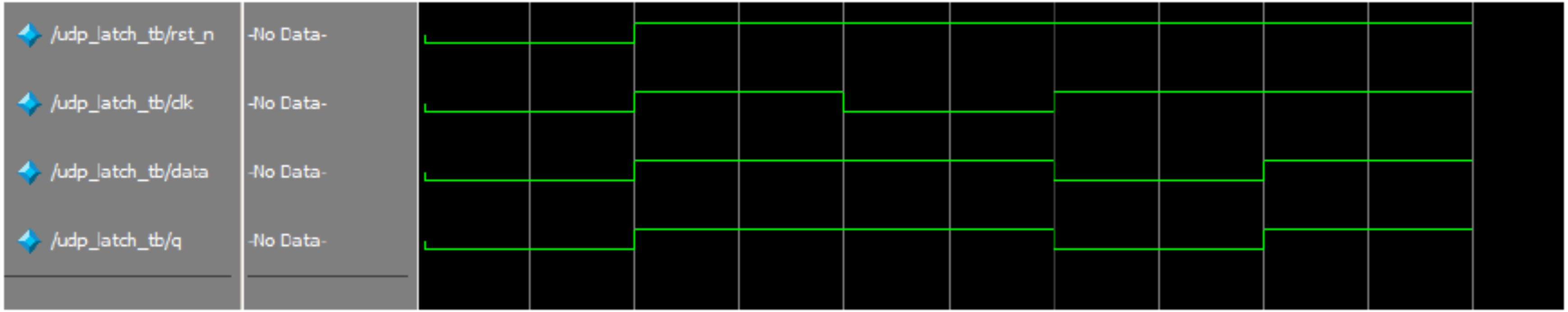
Sequential Udps Example

```
//A gated latch as a level-sensitive udp
primitive udp_latch (q, data, clk, rst_n);
input data, clk, rst_n;
output q;
reg q;
initial
q = 0; //initialize output q to 0
table //define state table
//inputs should be in the same order as the input list
//data clk rst_n : q : q+; q+ is next state
    ?     ?     0     : ? : 0; //latch is reset
    0     0     1     : ? : -; //- means no change in state
    0     1     1     : ? : 0; //data=0; clk=1; q+=0
    1     0     1     : ? : -;
    1     1     1     : ? : 1; //data=1; clk=1; q+=1
    ?     0     1     : ? : -;
endtable
endprimitive
```

Gated latch simulation output

$monitor display output:

```
# rst_n=0, data=0, clk=0, q=0
# rst_n=1, data=1, clk=1, q=1
# rst_n=1, data=1, clk=0, q=1
# rst_n=1, data=0, clk=1, q=0
# rst_n=1, data=1, clk=1, q=1
```



positive edge triggered d-flip flop udp example

```
//A positive-edge-sensitive D flip-flop
primitive udp_dff_pos_edge1 (q, d, clk, rst_n);
input d, clk, rst_n;
output q;
reg q;
initial q = 0; //only used for sequential udp to initialise output q to 0
q = 0;
table //define state table
//inputs are in the same order as the input list
// d  clk rst_n : q : q+; q+ is the next state
   ? (??)  0     : ? : 0; //rst_n = 0 means reset q+ =0
   0 (01)  1     : ? : 0; //(01) is rising edge
   1 (01)  1     : ? : 1; //rst_n = 1 means no rst
   1 (0x)  1     : 1 : 1; //(0x) is no change
   0 (0x)  1     : 0 : 0;
   ? (?0)  1     : ? : -; //ignore negative edge of clk
 (??) ?    1     : ? : -; //// ignore data changes on steady clock
endtable
endprimitive
```

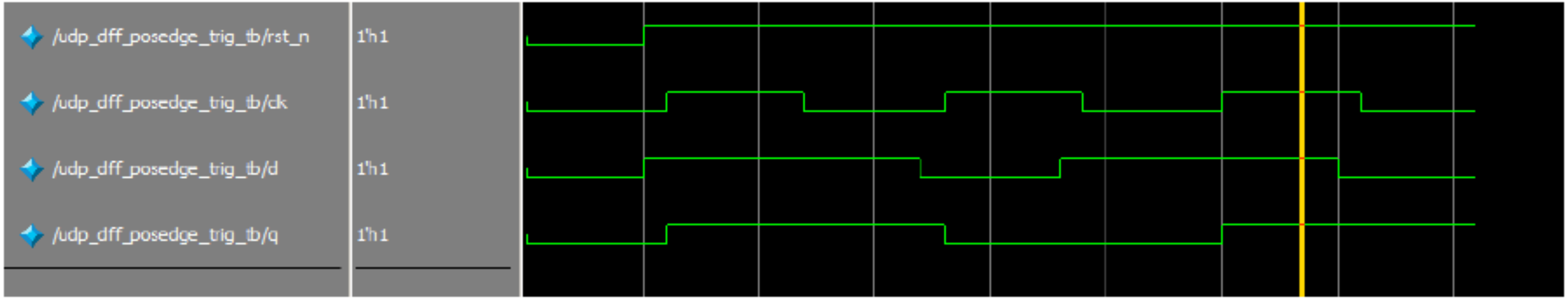
positive edge triggered d-flip flop udp test bench

```
//test bench for the positive-edge-triggered D flip-flop
module udp_dff_posedge_trig_tb;
reg d, clk, rst_n;
wire q;
initial
$monitor ("rst_n=%b, d=%b, clk=%b, q=%b", rst_n, d, clk, q); //display variables
initial // provide input vectors
        begin
                #0 rst_n=1'b0; d=1'b0; clk=1'b0;
                #10 rst_n=1'b1; d=1'b1; #2 clk=1'b1;
                #10 rst_n=1'b1; d=1'b1; #2 clk=1'b0;
                #10 rst_n=1'b1; d=1'b0; #2 clk=1'b1;
                #10 rst_n=1'b1; d=1'b1; #2 clk=1'b0;
                #10 rst_n=1'b1; d=1'b1; #2 clk=1'b1;
                #10 rst_n=1'b1; d=1'b0; #2 clk=1'b0;
                #10 $stop;
        end
udp_dff_pos_edge1 inst1 (q, d, clk, rst_n);  //instantiation of dff module
endmodule
```

positive edge triggered d-flip flop udp simulation output

$monitor display output:

```
# rst_n=0, d=0, clk=0, q=0
# rst_n=1, d=1, clk=0, q=0
# rst_n=1, d=1, clk=1, q=1
# rst_n=1, d=1, clk=0, q=1
# rst_n=1, d=0, clk=0, q=1
# rst_n=1, d=0, clk=1, q=0
# rst_n=1, d=1, clk=1, q=0
# rst_n=1, d=1, clk=0, q=0
# rst_n=1, d=1, clk=1, q=1
# rst_n=1, d=0, clk=1, q=1
# rst_n=1, d=0, clk=0, q=1
```


/udp_dff_posedge_trig_tb/rst_n
1'h1
/udp_dff_posedge_trig_tb/clk
1'h1
/udp_dff_posedge_trig_tb/d
1'h1
/udp_dff_posedge_trig_tb/q
1'h1

Guidelines for Design

- UDPs model functionality only. They do not model timing or process technology (such as CMOS, TTL, ECL).
- The limit on the maximum number of inputs of a UDP is specific to the Verilog simulator being used. However, Verilog simulators are required to allow a minimum of 9 inputs for sequential UDPs and 10 for combinational UDPs.
- A UDP is typically implemented as a lookup table in memory. As the number of inputs increases, the number of table entries grows exponentially. Thus, the memory requirement for a UDP grows exponentially in relation to the number of inputs. It is not advisable to design a block with a large number of inputs as a UDP.
- Level-sensitive entries take precedence over edge sensitive entries. If an edge-sensitive and level-sensitive entry clash on the same inputs, the output is determined by the level-sensitive entry because it has precedence over the edge-sensitive entry.
- More than one edge specification in a single table entry, as shown below, is illegal in Verilog. (01) (10) 0 : ? : 1 ; //illegal;two edge transitions in an entry