Verilog HDL:
User Defined Primitives

Pravin Zode

Outline

- UDP Introduction

- Types of UDP

- Rules for defining UDP

- Combinational UDP

- Sequential UDP

Introduction

- Verilog includes built-in primitives like and, or, not, nand, nor, etc.

- Designers can also create custom logic blocks called User-Defined Primitives (UDPs)

- Instantiated just like gate-level primitives in the design

- Useful for abstracting small logic blocks without full modules

- **Combinational UDPs:**

  - Output depends only on current inputs.

  - Example: 4-to-1 Multiplexer

- **Sequential UDPs:**

  - Output depends on current inputs + current output (state).

  - Output acts as the internal state.

  - Examples: Latches, Flip-Flops.

UDP Definition

```
//UDP name and terminal list
primitive <udp_name> (
<output_terminal_name> (only one allowed)
<input_terminal_names> );
```

```
//Terminal declarations
output <output_terminal_name>;
input <input_terminal_names>;
reg  <output_terminal_name>; (optional; only for sequential
                                                    UDP)
```

```
// UDP initialization (optional; only for sequential UDPs)
initial <output_terminal_name> = <value>;
```

```
//UDP state table
table
    <table entries>
endtable
```

```
//End of UDP definition
endprimitive
```

Rules for defining UDP

- Scalar Inputs Only

  - All input terminals must be 1-bit (scalar)

  - Multiple inputs are allowed

- Single Scalar Output

  - Only one 1-bit output terminal is permitted

  - Output must appear first in the terminal list

- Output Declaration

  - Declared using the keyword **output**

  - In sequential UDPs, it must also be declared as **reg.**

- Input Declaration

  - Inputs are declared using the **input** keyword

Rules for defining UDP

- initial statement can be used to initialize state (output)

- This is optional and assigns a 1-bit value to the output

- Valid State Table Values : Allowed entries: 0, 1, X

- z values are not supported and treated as X

- UDPs must be defined at the same level as modules

- Cannot be defined inside modules, but can be instantiated inside them.

- Do not support inout ports

- These rules apply to combinational and sequential UDPs a

Example

```verilog
1    primitive udp_example (out, in1, in2);
2        output out;
3        input in1, in2;
4        table
5            0 0 : 0;
6            0 1 : 1;
7            1 0 : 1;
8            1 1 : 0;
9        endtable
10   endprimitive
```

Example : Adder

```
1   // ------------------------------------
2   // Full Adder Sum Generation using UDP
3   // sum = a ^ b ^ c
4   //------------------------------------
5   primitive udp_sum (sum, a, b, c);
6     input a, b, c;
7     output sum;
8
9     table
10    // a b c : sum
11       0 0 0 : 0;
12       0 0 1 : 1;
13       0 1 0 : 1;
14       0 1 1 : 0;
15       1 0 0 : 1;
16       1 0 1 : 0;
17       1 1 0 : 0;
18       1 1 1 : 1;
19    endtable
20
21  endprimitive
```
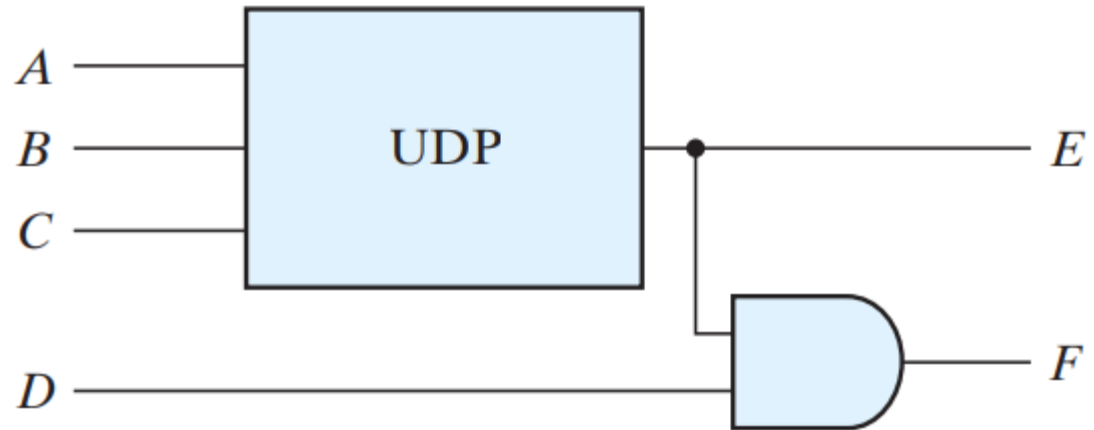
```
1   // ------------------------------------
2   // Full Adder Carry Generation UDP
3   // cout = (a & b) | (b & c) | (a & c)
4   // ------------------------------------
5   primitive udp_cy (cout, a, b, c);
6     input a, b, c;
7     output cout;
8
9     table
10    // a b c : cout
11       0 0 0 : 0;
12       0 0 1 : 0;
13       0 1 0 : 0;
14       0 1 1 : 1;
15       1 0 0 : 0;
16       1 0 1 : 1;
17       1 1 0 : 1;
18       1 1 1 : 1;
19    endtable
20
21  endprimitive
```

Example : Adder (Instantiation)

```verilog
1    // ---------------------------------------------------
2    // Full Adder Implementation by Instantiating UDPs
3    // SUM = a ^ b ^ c
4    // COUT = (a & b) | (b & c) | (a & c)
5    // ---------------------------------------------------
6
7    module full_adder (sum, cout, a, b, c);
8      input a, b, c;
9      output sum, cout;
10
11     // Instantiate UDP for sum
12     udp_sum   SUM   (sum, a, b, c);
13
14     // Instantiate UDP for carry
15     udp_cy    CARRY (cout, a, b, c);
16   endmodule
```

Example : Expression

```
1    // Verilog model: User-defined Primitive
2    primitive UDP_1 (D, A, B, C);
3    output D;
4    input A, B, C;
5    //Truth table f (A, B, C) =SOP(0, 2, 4, 6, 7);
6    table
7    // A B C : D // Column header comment
8    0 0 0 : 1;
9    0 0 1 : 0;
10   0 1 0 : 1;
11   0 1 1 : 0;
12   1 0 0 : 1;
13   1 0 1 : 0;
14   1 1 0 : 1;
15   1 1 1 : 1;
16   endtable
17   endprimitive
18   // Instantiate primitive
19   module Circuit_with_UDP_1 (e, f, a, b, c, d);
20   output e, f;
21   input a, b, c, d
22   UDP_1 x1 (e, a, b, c);
23   and a1 (f, e, d); // Option gate instance name omitted
24   endmodule
```

Rules for State Table Entries

- Input Order is Crucial : Inputs in the state table must appear in the same order as listed in the terminal definition

- Inputs and output in the state table are separated by a colon (:).

- Each state table entry must end with a semicolon (;) All valid input combinations that produce a known output must be explicitly written.

- Define all possible combinations, especially those involving X values, to avoid ambiguous outputs.

UDP with Don't care

```verilog
1   primitive udp_or (out, a, b);
2     output out;
3     input a, b;
4
5     table
6     // a   b : out
7        0   0 : 0;
8        0   1 : 1;
9        1   0 : 1;
10       1   1 : 1;
11
12       0   x : x;   // optional: or define 0 x : x;
13       x   0 : x;
14       1   x : 1;   // logic holds if one input is 1
15       x   1 : 1;
16       x   x : x;
17    endtable
18  endprimitive
```

UDP with Don't care shortcut

```verilog
1   primitive udp_or (out, a, b);
2     output out;
3     input a, b;
4
5     table
6     // a  b : out
7        0  0 : 0;
8        ?  1 : 1;
9        1  ? : 1;
10       x  0 : x;
11       0  x : x;
12       x  x : x;
13    endtable
14  endprimitive
```

- ? means 0, 1, or x → "don't care" for matching purposes.
- So:
- ? 1 : 1; covers {0 1}, {1 1}, {x 1}
- 1 ? : 1; covers {1 0}, {1 1}, {1 x}

Example: Combinational UDPs

```verilog
1  primitive comb_udp (out, a, b);
2    output out;
3    input a, b;
4    table
5      // a  b  : out
6         0  0  : 0;
7         0  1  : 1;
8         1  0  : 1;
9         1  1  : 0;
10   endtable
11 endprimitive
12
13 module tb_comb_udp;
14   reg a, b;
15   wire out;
16   comb_udp uut(out, a, b);
17   initial begin
18     $monitor("At time %0t: a=%b, b=%b, out=%b", $time, a, b, out);
19     a = 0; b = 0; #10;
20     a = 0; b = 1; #10;
21     a = 1; b = 0; #10;
22     a = 1; b = 1; #10;
23   end
24 endmodule
```

Expected Output:

```
At time 0: a=0, b=0, out=0
At time 10: a=0, b=1, out=1
At time 20: a=1, b=0, out=1
At time 30: a=1, b=1, out=0
```

Example: UDP Instance

```verilog
1   module udp_instance;
2     reg a, b;
3     wire y;
4     comb_udp my_udp (y, a, b);
5   endmodule
6
7   module tb_udp_instance;
8     reg a, b;
9     wire y;
10    comb_udp uut (y, a, b);
11    initial begin
12      $monitor("At time %0t: a=%b, b=%b, y=%b", $time, a, b, y);
13      a = 0; b = 0; #10;
14      a = 0; b = 1; #10;
15      a = 1; b = 0; #10;
16      a = 1; b = 1; #10;
17      $finish;
18    end
19  endmodule
```
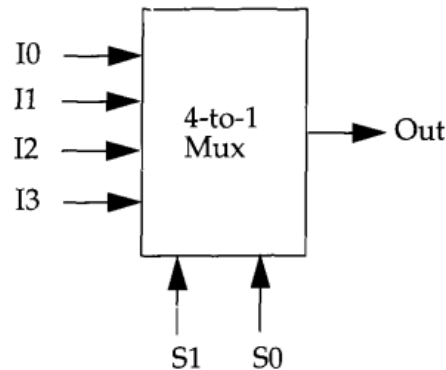
```verilog
primitive comb_udp (out, a, b);
  output out;
  input a, b;
  table
    // a  b  : out
       0  0  : 0;
       0  1  : 1;
       1  0  : 1;
       1  1  : 0;
  endtable
endprimitive
```

Expected Output:

```
At time 0: a=0, b=0, y=0
At time 10: a=0, b=1, y=1
At time 20: a=1, b=0, y=1
At time 30: a=1, b=1, y=0
```

S1 and S0 diagram with 4-to-1 Mux, inputs I0, I1, I2, I3 and output Out.

S1	S0	Out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

```
// 4-to-1 multiplexer. Define it as a primitive
primitive 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;

table
   //  i0   i1   i2   i3,  s1   s0   : out
       1    ?    ?    ?    0    0    : 1 ;
       0    ?    ?    ?    0    0    : 0 ;
       ?    1    ?    ?    0    1    : 1 ;
       ?    0    ?    ?    0    1    : 0 ;
       ?    ?    1    ?    1    0    : 1 ;
       ?    ?    0    ?    1    0    : 0 ;
       ?    ?    ?    1    1    1    : 1 ;
       ?    ?    ?    0    1    1    : 0 ;
       ?    ?    ?    ?    x    ?    : x ;
       ?    ?    ?    ?    ?    x    : x ;
endtable

endprimitive
```

- Sequential UDP must be declared as reg, since it stores state Uses internal state (reg) to store values
- A sequential UDP may use an initial block to initialize the output value before simulation starts
- State Table Format:
- Three fields separated by colons (:)
  - Inputs
  - Current state (value of the output reg)
  - Next state
- Example format:
  1 0 : 0 : 1; (Input = 1 0, Current State = 0, Next State = 1)

Input Specification Types:

- Level-sensitive: Responds to input values (like 0 or 1)
- Edge-sensitive: Responds to input transitions ((01), (10), etc.)

Next State Logic:

- Determined based on inputs + current state
- The next state becomes the output for the next evaluation

Complete Specification Required:

- Must define all valid combinations of input + current state to avoid unknown (X) results.

- Incomplete tables result in unpredictable outputs during simulation

Level-sensitive vs Edge-sensitive:

- Level-sensitive UDP: Behaves like latches

- Edge-sensitive UDP: Behaves like flip-flops (e.g., D flip-flop triggered on clock edge).

UDP Table Symbols

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 value	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

```
table
    //  d clock clear : q : q+  ;

        ?    ?      1   : ? : 0 ; //output = 0 if clear = 1
        ?    ?      f   : ? : - ; //ignore negative transition of clear

        1    f      0   : ? : 1 ; //latch data on negative transition of
        0    f      0   : ? : 0 ; //clock

        ?   (1x)    0   : ? : - ; //hold q if clock transitions to unknown
                                                //state

        ?    p      0   : ? : - ; //ignore positive transitions of clock

        *    ?0    : ? : - ; //ignore any change in d when
                                    //clock is steady
endtable
```

Example : Level-Sensitive Sequential UDPs

```verilog
1  primitive level_seq_udp (out, clk, d);
2    output out;
3    reg out;
4    input clk, d;
5    table
6      // clk  d  : state : new_state
7         0    ?  :   ?   :   - ; // No change when clk = 0
8         1    0  :   ?   :   0 ;
9         1    1  :   ?   :   1 ;
10   endtable
11 endprimitive
12
13 module tb_level_seq_udp;
14   reg clk, d;
15   wire out;
16   level_seq_udp uut(out, clk, d);
17   initial begin
18     clk = 0; d = 0;
19     #5 clk = 1; d = 1;
20     #5 clk = 0;
21     #5 clk = 1; d = 0;
22     #5 clk = 0;
23   end
24   initial $monitor("At time %0t: clk=%b, d=%b, out=%b", $time, clk, d, out);
25 endmodule
```

Expected Output:

```
At time 5: clk=1, d=1, out=1
At time 15: clk=1, d=0, out=0
```

Example : Level-Sensitive Sequential UDPs

```verilog
1    // Define level-sensitive latch using a sequential UDP
2    primitive latch(q, d, clock, clear);
3    // Output declaration
4    output q;
5    reg q; // 'q' must be reg for sequential UDP
6    // Input declaration
7    input d, clock, clear;
8    // Initialize output
9    initial
10       q = 0; // Set initial value of q
11   // State table definition
12   // Format: d clock clear : current_state : next_state ;
13   table
14       // Clear active (asynchronous clear)
15       ?  ?   1 :   ?   : 0 ; // Clear overrides everything
16       // Latch behavior when clock is high and clear is low
17       1  1   0 :   ?   : 1 ; // Latch data = 1
18       0  1   0 :   ?   : 0 ; // Latch data = 0
19       // Retain value when clock is low and clear is inactive
20       ?  0   0 :   ?   : - ; // Hold previous state
21   endtable
22   endprimitive
```

Example : Edge-Sensitive Sequential UDPs

```verilog
1   primitive edge_seq_udp (out, clk, d);
2     output out;
3     reg out;
4     input clk, d;
5     table
6       // clk  d  : state : new_state
7          (01) 0  :  ?    :  0;
8          (01) 1  :  ?    :  1;
9     endtable
10  endprimitive
11
12  module tb_edge_seq_udp;
13    reg clk, d;
14    wire out;
15    edge_seq_udp uut(out, clk, d);
16    initial begin
17      clk = 0; d = 0;
18      #5 clk = 1; d = 1;
19      #5 clk = 0;
20      #5 clk = 1; d = 0;
21      #5 clk = 0;
22    end
23    initial $monitor("At time %0t: clk=%b, d=%b, out=%b", $time, clk, d, out);
24  endmodule
```

Expected Output:

```
At time 5: clk=1, d=1, out=1
At time 15: clk=1, d=0, out=0
```

Example : Edge-Sensitive Sequential UDPs

```verilog
1    // Define an edge-sensitive sequential UDP (D Flip-Flop with asynchronous clear)
2    primitive edge_dff(q, d, clock, clear);
3    // Declarations
4    output q;
5    reg q; // Must be reg to store state
6    input d, clock, clear;
7    // Initialization
8    initial
9        q = 0;
10   // State table
11   // Format: d clock clear : q : q+ ;
12   table
13       ?  ?     1 : ? : 0 ;   // Asynchronous clear
14       ?  ?  (10): ? : - ;   // Ignore negative edge of clear
15       1 (10)  0 : ? : 1 ;   // Capture data=1 on rising edge of clock
16       0 (10)  0 : ? : 0 ;   // Capture data=0 on rising edge of clock
17       ? (1x)  0 : ? : - ;   // Hold output if clock becomes unknown after 1
18       ? (0?)  0 : ? : - ;   // Ignore positive transitions
19       ? (x1)  0 : ? : - ;   // Ignore positive transitions
20       (??)    ? : ? : - ;   // Ignore changes in 'd' when clock is steady
21
22   endtable
23   endprimitive
```

```
1    // Edge-triggered T-flipflop using UDP
2    primitive T_FF(q, clk, clear);
3
4    // Output and input declarations
5    output q;
6    reg q;
7    input clk, clear;
8
9    // No initialization needed; TFF is initialized using clear signal
10
11   table
12   // clk    clear : q : q+ ;
13      ?         1    : ? : 0 ;       // Asynchronous clear
14    (10)       0    : 1 : 0 ;       // Toggle q on rising edge if q is 1 -> becomes 0
15    (10)       0    : 0 : 1 ;       // Toggle q on rising edge if q is 0 -> becomes 1
16    (0?)       0    : ? : - ;       // Ignore positive transitions
17   endtable
18
19   endprimitive
```

```verilog
1   module counter(Q, clock, clear);
2     output [3:0] Q;
3     input clock, clear;
4
5     // Instantiate 4 T flip-flops
6     T_FF tff0(Q[0], clock, clear);
7     T_FF tff1(Q[1], Q[0], clear);
8     T_FF tff2(Q[2], Q[1], clear);
9     T_FF tff3(Q[3], Q[2], clear);
10  endmodule
```

```verilog
1   `timescale 1ns / 1ps
2   module tb_counter;
3     reg clk, clr;
4     wire [3:0] Q;
5
6     // Instantiate the counter
7     counter uut(Q, clk, clr);
8     // Clock generation
9     initial begin
10      clk = 0;
11      forever #5 clk = ~clk;
12      // 10ns clock period
13    end
14    // Stimulus
15    initial begin
16      $dumpfile("counter.vcd");
17      $dumpvars(0, tb_counter);
18
19      clr = 1; #10;
20      clr = 0; #200;
21
22      $finish;
23    end
24  endmodule
```

Summary

- User-Defined Primitives (UDP) allow creation of custom Verilog primitives using lookup tables
- UDPs are useful for defining simple, functional blocks in a concise format
- A UDP can have only one output terminal and must be defined at the same level as modules
- UDPs are instantiated like gate-level primitives (e.g., and, or)
- The state table is the core part of a UDP definition.

Thank you !

Happy Learning