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

Types of UDP

- **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

```
primitive udp_example (out, in1, in2);
  output out;
  input in1, in2;
  table
    0 0 : 0;
    0 1 : 1;
    1 0 : 1;
    1 1 : 0;
  endtable
endprimitive
```

Example : Adder

```
// ----------------------------------
// Full Adder Sum Generation using UDP
// sum = a ^ b ^ c
//----------------------------------
primitive udp_sum (sum, a, b, c);
  input a, b, c;
  output sum;

  table
  // a b c : sum
     0 0 0 : 0;
     0 0 1 : 1;
     0 1 0 : 1;
     0 1 1 : 0;
     1 0 0 : 1;
     1 0 1 : 0;
     1 1 0 : 0;
     1 1 1 : 1;
  endtable

endprimitive
```

```
// ----------------------------------
// Full Adder Carry Generation UDP
// cout = (a & b) | (b & c) | (a & c)
// ----------------------------------
primitive udp_cy (cout, a, b, c);
  input a, b, c;
  output cout;

  table
  // a b c : cout
     0 0 0 : 0;
     0 0 1 : 0;
     0 1 0 : 0;
     0 1 1 : 1;
     1 0 0 : 0;
     1 0 1 : 1;
     1 1 0 : 1;
     1 1 1 : 1;
  endtable

endprimitive
```

Example : Adder (Instantiation)

```
// ----------------------------------------------------------
// Full Adder Implementation by Instantiating UDPs
// SUM = a ^ b ^ c
// COUT = (a & b) | (b & c) | (a & c)
// ----------------------------------------------------------

module full_adder (sum, cout, a, b, c);
  input a, b, c;
  output sum, cout;

  // Instantiate UDP for sum
  udp_sum  SUM   (sum, a, b, c);

  // Instantiate UDP for carry
  udp_cy   CARRY (cout, a, b, c);
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

```
primitive udp_or (out, a, b);
  output out;
  input a, b;

  table
  // a  b : out
     0  0 : 0;
     0  1 : 1;
     1  0 : 1;
     1  1 : 1;

     0  x : x;  // optional: or define 0 x : x;
     x  0 : x;
     1  x : 1;  // logic holds if one input is 1
     x  1 : 1;
     x  x : x;
  endtable
endprimitive
```

UDP with Don’t care shortcut

```
primitive udp_or (out, a, b);
  output out;
  input a, b;

  table
  // a  b : out
     0  0 : 0;
     ?  1 : 1;
     1  ? : 1;
     x  0 : x;
     0  x : x;
     x  x : x;
  endtable
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

```
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

module tb_comb_udp;
  reg a, b;
  wire out;
  comb_udp uut(out, a, b);
  initial begin
    $monitor("At time %0t: a=%b, b=%b, out=%b", $time, a, b, out);
    a = 0; b = 0; #10;
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
    a = 1; b = 1; #10;
  end
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

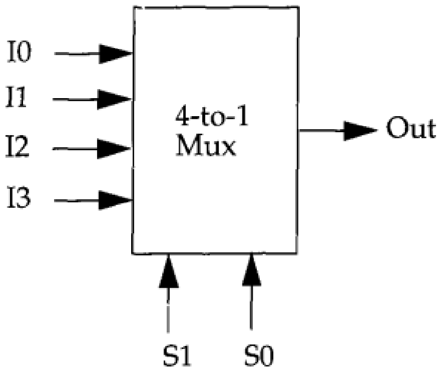
```
module udp_instance;
  reg a, b;
  wire y;
  comb_udp my_udp (y, a, b);
endmodule

module tb_udp_instance;
  reg a, b;
  wire y;
  comb_udp uut (y, a, b);
  initial begin
    $monitor("At time %0t: a=%b, b=%b, y=%b", $time, a, b, y);
    a = 0; b = 0; #10;
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
    a = 1; b = 1; #10;
    $finish;
  end
endmodule
```

```
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
```

Example: Combinational UDPs



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 UDPs

- 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)

Sequential UDPs

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

Sequential UDPs

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

UDP Table Symbols

```
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

```
primitive level_seq_udp (out, clk, d);
  output out;
  reg out;
  input clk, d;
  table
    // clk  d  : state : new_state
       0    ?  :   ?   :    - ; // No change when clk = 0
       1    0  :   ?   :    0 ;
       1    1  :   ?   :    1 ;
  endtable
endprimitive

module tb_level_seq_udp;
  reg clk, d;
  wire out;
  level_seq_udp uut(out, clk, d);
  initial begin
    clk = 0; d = 0;
    #5 clk = 1; d = 1;
    #5 clk = 0;
    #5 clk = 1; d = 0;
    #5 clk = 0;
  end
  initial $monitor("At time %0t: clk=%b, d=%b, out=%b", $time, clk, d, out);
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

```
// Define level-sensitive latch using a sequential UDP
primitive latch(q, d, clock, clear);
// Output declaration
output q;
reg q; // 'q' must be reg for sequential UDP
// Input declaration
input d, clock, clear;
// Initialize output
initial
    q = 0; // Set initial value of q
// State table definition
// Format: d clock clear : current_state : next_state ;
table
    // Clear active (asynchronous clear)
    ?  ?   1 :   ?   : 0 ; // Clear overrides everything
    // Latch behavior when clock is high and clear is low
    1  1   0 :   ?   : 1 ; // Latch data = 1
    0  1   0 :   ?   : 0 ; // Latch data = 0
    // Retain value when clock is low and clear is inactive
    ?  0   0 :   ?   : - ; // Hold previous state
endtable
endprimitive
```

Example : Edge-Sensitive Sequential UDPs

```
primitive edge_seq_udp (out, clk, d);
  output out;
  reg out;
  input clk, d;
  table
    // clk  d  : state : new_state
       (01) 0  :  ?   :  0;
       (01) 1  :  ?   :  1;
  endtable
endprimitive

module tb_edge_seq_udp;
  reg clk, d;
  wire out;
  edge_seq_udp uut(out, clk, d);
  initial begin
    clk = 0; d = 0;
    #5 clk = 1; d = 1;
    #5 clk = 0;
    #5 clk = 1; d = 0;
    #5 clk = 0;
  end
  initial $monitor("At time %0t: clk=%b, d=%b, out=%b", $time, clk, d, out);
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

```
// Define an edge-sensitive sequential UDP (D Flip-Flop with asynchronous clear)
primitive edge_dff(q, d, clock, clear);
// Declarations
output q;
reg q; // Must be reg to store state
input d, clock, clear;
// Initialization
initial
    q = 0;
// State table
// Format: d clock clear : q : q+ ;
table
    ?  ?     1 : ? : 0 ;   // Asynchronous clear
    ?  ?  (10): ? : - ;   // Ignore negative edge of clear
    1 (10)   0 : ? : 1 ;   // Capture data=1 on rising edge of clock
    0 (10)   0 : ? : 0 ;   // Capture data=0 on rising edge of clock
    ? (1x)   0 : ? : - ;   // Hold output if clock becomes unknown after 1
    ? (0?)   0 : ? : - ;   // Ignore positive transitions
    ? (x1)   0 : ? : - ;   // Ignore positive transitions
    (??)     ? : ? : - ;   // Ignore changes in 'd' when clock is steady

endtable
endprimitive
```

Example : Sequential UDPs (T-FF)

```
// Edge-triggered T-flipflop using UDP
primitive T_FF(q, clk, clear);

// Output and input declarations
output q;
reg q;
input clk, clear;

// No initialization needed; TFF is initialized using clear signal

table
// clk   clear : q : q+ ;
    ?       1   : ? : 0 ;     // Asynchronous clear
  (10)     0    : 1 : 0 ;     // Toggle q on rising edge if q is 1 -> becomes 0
  (10)     0    : 0 : 1 ;     // Toggle q on rising edge if q is 0 -> becomes 1
  (0?)     0    : ? : - ;     // Ignore positive transitions
endtable

endprimitive
```

Example : Sequential UDPs (Counter)

```verilog
module counter(Q, clock, clear);
  output [3:0] Q;
  input clock, clear;

  // Instantiate 4 T flip-flops
  T_FF tff0(Q[0], clock, clear);
  T_FF tff1(Q[1], Q[0], clear);
  T_FF tff2(Q[2], Q[1], clear);
  T_FF tff3(Q[3], Q[2], clear);
endmodule
```

```verilog
`timescale 1ns / 1ps
module tb_counter;
  reg clk, clr;
  wire [3:0] Q;

  // Instantiate the counter
  counter uut(Q, clk, clr);
  // Clock generation
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
    // 10ns clock period
  end
  // Stimulus
  initial begin
    $dumpfile("counter.vcd");
    $dumpvars(0, tb_counter);

    clr = 1; #10;
    clr = 0; #200;

    $finish;
  end
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