


EE461 Verilog-HDL

Week 4 UDP Design

Alex Yang, Engineering School, NPU

Week 4 outlines

- User Defined Primitives **(Non Synthesizable)**
  - Introduction
  - Syntax
  - UDP ports rules
  - Body
  - Symbols
  - Combinational UDPs
  - Level Sensitive Sequential UDP
  - Edge-Sensitive UDPs
  - UDP or Module Decision

Syntax

- UDP begins with reserve word primitive and ends with endprimitive.

```
primitive udpSyn (
                        a_o,            // Port a
                        b_i,            // Port b
                        c_i,            // Port c
                        d-i             // Port d
);
            output  a_0;
            input   b_i,    c_i,    d_i;

            // UDP function code here

endprimitive
```

Introduction

- If built-in primitives in Verilog like gates, transmission gates and switches are not enough, Verilog provides UDP, or simply User Defined Primitives to generate more.
- The following models are included
  - Combinational Logic
  - Sequential Logic
- Timing information along with these UDPs could model complete ASIC library models.

UDP ports rules

- An UDP can contain only one output and up to 10 inputs.
- Output port should be the first port followed by one or more input ports.
- All UDP ports are scalar, i.e. Vector ports are not allowed.
- UDPs can NOT have bidirectional ports.
- The output terminal of a sequential UDP requires an additional declaration as type reg.
- It is illegal to declare a reg for the output terminal of a combinational UDP

UDP ports rules

- The state in a sequential UDP can be initialized with an ‘initial’ statement. This statement is optional. A 1-bit value is assigned to the output, which is declared as reg.
- The state table entries can contain values 0,1,or x . UDPs do not handle z values. Z values passed to a UDP are treated x values.
- "inout" port can't be used in UDP.
- UDPs are defined at the same level as modules. UDPs can not be defined inside modules. They can be instantiated only inside modules. UDPs are instantiated exactly like gate primitives.
- UDP primitive can't instantiate any other primitives or modules
- If more outputs are needed in UDP, instantiate more UPDs in top module
- Nonsynthesizable

Body

- Table
  - Functionality of Primitives: "table" and "endtable"

```
primitive udpBody (
                        a_o,            // Port a
                        b_i,            // Port b
                        c_i             // Port c
);
            output  a_o;
            input   b_i,    c_i;

            table                                   // A = B | C;
                // B  C    : A//Input ports must be the same as port list
                   ?  1    : 1;            //? = 0, 1 or x
                   1  ?    : 1;
                   0  0    : 0;
            endtable

endprimitive
```

Body

```
`include "udpBody.v"                    //This file and testbench must be under the same directory
module udpBodyTB();
        reg     b_r,    c_r;
        wire    a_w;
        udpBody uUdpBody (
                        a_w,
                        b_r,
                        c_r
        );
        initial begin
                $monitor(" B_i = %b C_i = %b  A_o = %b", b_r, c_r, a_w);
                b_r = 0;
                c_r = 0;
                #1 b_r = 1;
                #1 b_r = 0;
                #1 c_r = 1;
                #1 b_r = 1'bx;
                #1 c_r = 0;
                #1 b_r = 1;
                #1 c_r = 1'bx;
                #1 b_r = 0;
                #1 $finish;
        end
endmodule
```

Results:

B_i = 0 C_i = 0 A_o = 0

B_i = 1 C_i = 0 A_o = 1

B_i = 0 C_i = 0 A_o = 0

B_i = 0 C_i = 1 A_o = 1

B_i = x C_i = 1 A_o = 1

B_i = x C_i = 0 A_o = x

B_i = 1 C_i = 0 A_o = 1

B_i = 1 C_i = x A_o = 1

B_i = 0 C_i = x A_o = x

Body

- Initial
  - Initial statement is used for initialization of sequential logic UDPs.

```
primitive udpInit (
                        a_o,
                        b_i,
                        c_i
);
                output  a_o;
                input   b_i,        c_i;
                reg     a_o;

                initial a_o = 1'b1;         // Variable in LHS must be "reg"
                                            // Variables in RHS could be "reg" or "wire"
                table
                        // udp_initial behaviour
                endtable

endprimitive
```

Combinational UDPs

- Example:

```
primitive udpAnd (
                    out_o,
                    a_i,        // Port list
                    b_i
);
            output    out_o;    // Don't care order of port list in declaration
            input     a_i;      // Output must be wire data type
            input     b_i;
            table   // a b : out;   // Order of inputs must be the same as port list
                      0 0 : 0;
                      0 1 : 0;
                      1 0 : 0;
                      1 1 : 1;
                    // x  x : x;
            endtable
endprimitive
```

If all inputs are specified as x, then the output must be specified as x. All combinations that are not explicitly specified result in a default output state of x.

Combinational UDPs

- State table entries
  - <input1><input2>……<inputn>:<output>;
    The <input#> values must be in the same order as in the input list
  - It is not necessary to explicitly specify every possible input combination. Output is x if the combinations are not explicitly.
  - Create truth table first w/o considering x & z, and modify to shorthand format with “?”.
  - It is illegal to have the same combination of inputs, specified for different outputs.

```
primitive udpAnd(f_o, a_i, b_i);
    output    f_o;
    input     a_i,
    input     b_i;
    table
            // a b : f;
               1 1 : 1;  // If adding one more line, like a = 1 and b = 1, f=0, it is illegal.
               0 ? : 0;
               ? 0 : 0;     // ? expanded to 0, 1, x
    endtable
endprimitive
```

Combinational UDPs

- Instantiation of UDPs

```
`include "udpXOR.v"
`include "udpAND.v"
`include "udpOR.v"
module udpFA(sum, cout, x, y, cin);
        output sum, cout;
        input x, y, cin;
        wire s1, c1, c2;
        // instantiate udp primitives
        udpXOR   (s1, x, y);          //Unit name is optional
        udpAND   (c1, x, y);
        udpXOR   (sum, s1, cin);
        udpAND   (c2, s1, cin);
        udpOR    (cout, c1, c2);
endmodule
```

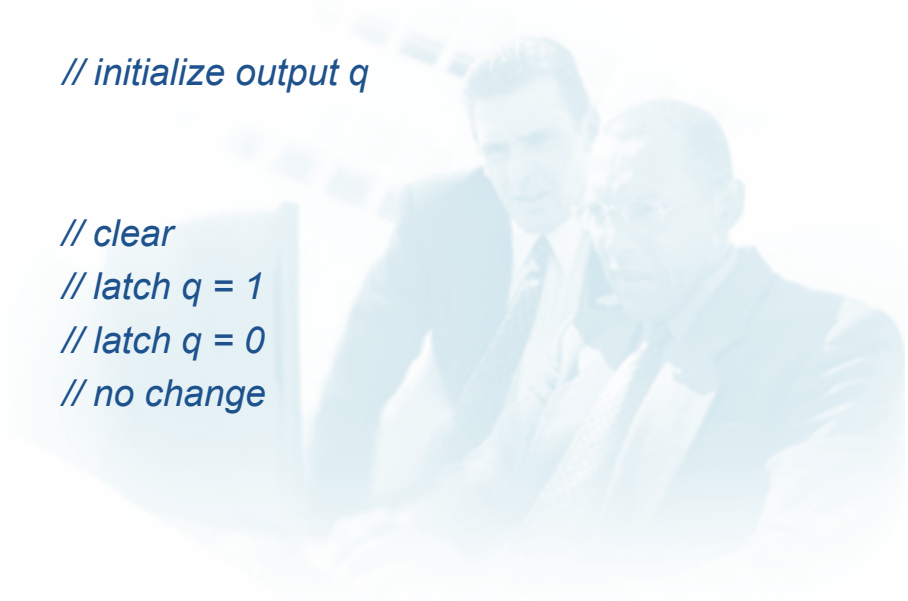
Level Sensitive Sequential UDP

- State table entries

  <input1><input2>……<inputn>:<current_state>:<next_state>;

  - The output is always declared as a reg
  - An initial statement can be used to initialize output

```
primitive dLatch(q, d, gate, clear);
        output      q;
        input       d, gate, clear;
        reg         q;
        initial     q = 0;                  // initialize output q
        table
                // d gate clear : q : q+;
                   ?   ?     1 :  ? : 0 ;   // clear
                   1   1     0 :  ? : 1 ;   // latch q = 1
                   0   1     0 :  ? : 0 ;   // latch q = 0
                   ?   0     0 :  ? : - ;   // no change
        endtable
endprimitive
```



Edge-Sensitive UDPs

- State table entries
  - All unspecified transitions default to the output value x
  - All transitions that should not affect the output must be explicitly specified. Otherwise, they will cause the value of the output to change to x.
  - If the UDP is sensitive to edges of any input, the desired output state must be specified for all edges of all inputs.

Edge-Sensitive UDPs

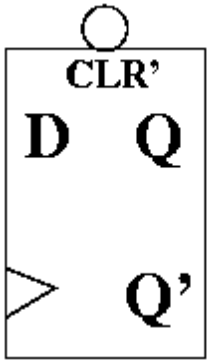
- State table entries
  - It is important to completely specify the UDP by covering all possible combinations of edge transitions and levels in the state table for which the outputs have a known value. Otherwise, some combinations may results in an unknown value.
  - Only one edge specification is allowed per table entry. More than one edge specification in a single table entry is illegal in Verilog.

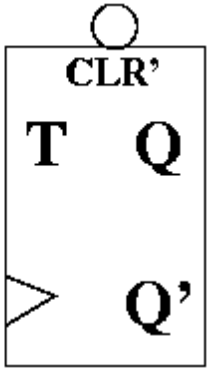
Symbols

Symbol	Interpretation	Explanation
?	0 or 1 or X	? means the variable can be 0 or 1 or x
b	0 or 1	Same as ?, but x is not included
f	(10)	Falling edge on an input
r	(01)	Rising edge on an input
p	(01) or (0x) or (x1) or **(1z)** or (z1)	Rising edge including x and z
n	(10) or (1x) or (x0) or **(0z)** or (z0)	Falling edge including x and z
*	(??)	All transitions
-	no change	No Change

Edge-Sensitive UDPs

- D Flip Flop & T Flip Flop





D	Q	Q^+	Operation
0	0	0	Reset
0	1	0	Reset
1	0	1	Set
1	1	1	Set

T	Q	Q^+	Operation
0	0	0	Hold
0	1	1	Hold
1	0	1	Toggle
1	1	0	Toggle

Edge-Sensitive UDPs

- Example:

```
primitive TFF(q, clk, clear);
        output   q;
        input    clk,      clear;
        reg      q;
        // define the behavior of edge-triggered T_FF
        table
            // clk clear : q : q+;
               ?      1 : ? : 0 ;        // asynchronous clear
               ?   (10) : ? : - ;        // ignore negative edge of clear
               (01)   0 : 1 : 0 ;        // toggle at positive edge
               (01)   0 : 0 : 1 ;        // of clk
               (1?)   0 : ? : - ;        // ignore negative edge of clk
        endtable
endprimitive
```

Edge-Sensitive UDPs

- Example:

```
primitive DFF(q, d, clk, reset);
        output   q;
        input    d, clk, reset;
        reg      q;

        table
            // d   clk reset : q : q+;
               ?    ?      1 : ? : 0 ;      // output = 1, if clear = 1
               ?    ?   (10) : ? : - ;      // ignore negative edge of clear

               1  (10)    0  : ?: 1 ;       // Get data at falling edge of
               0  (10)    0  : ?: 0 ;       // clock

               ?  (1x)    0  :  ? : - ;     // Hold q if clock goes to x state
```

Edge-Sensitive UDPs

- Example:

```
// d   clk reset : q : q+;
   ?  (0?)    0 : ? : - ;      //Ignore negative edge of clear
   ?  (x1)    0 : ? : - ;      // Ignore negative edge of clear

  (??)  ?     0 :  ? : - ;     //Ignore any changes of d w/o
                               // clock edge trigger
    endtable
endprimitive
```

Notice that the following statement is illegal

```
    table
    ... ...
        (01)   (10)   0:  ? : -;      // Illegal; 2 transitions in one entry
    ... ...
    endtable
```

Edge-Sensitive UDPs

- Instantiation of UDPs

```
`include "TFF.v"
module rippleCounter(clock, clear, q);
        input           clock;
        input           clear;
        output [3:0]  q;
        // instantiate the T_FFs.
        TFF   uTFF0                 (q[0], clock, clear);
        TFF   uTFF1                 (q[1], q[0], clear);
        TFF   uTFF2                 (q[2], q[1], clear);
        TFF   uTFF3                 (q[3], q[2], clear);
endmodule
```

UDP or Module Decision

- UDPs model functionality only without timing or process technology information(such as CMOS, TTL, ECL). The primary purpose of a UDP is to define the functional portion of a block in a simple and concise form.
- UDP is only for one output terminal. For more than one output, module should be used.
- Verilog simulators are required to allow a max of 9 inputs for sequential UDPs and 10 for combinational UDPs.
- It is not advisable to design a block with a large number of inputs as a UDP due to large memory spaces demanded
- UDPs are not always the appropriate method to design a block. Sometimes it is easier to design blocks as a module, such as an 8-to-1 multiplexer design. RTL level design would be much simpler, instead of UDP with large number of table entries. Indeed, It is important to consider complexity trade-offs for UDP selection