7

Figure 7-0
Example 7-0
Syntax 7-0
Table 7-0

User-Defined Primitives (UDPs)

This chapter describes a modeling technique whereby the user can effectively augment the set of predefined gate primitives by designing and specifying new primitive elements called user-defined primitives (UDPs). Instances of these new UDPs can then be used in exactly the same manner as the gate primitives to represent the circuit being modeled. This technique can reduce the amount of memory that a description needs and can improve simulation performance. Evaluation of these UDPs is accelerated by the Verilog-XL algorithm.

The following two types of behavior can be represented in a user-defined primitive:

- *combinational*—modeled by a combinational UDP
- *sequential*—modeled by a sequential UDP

A sequential UDP uses the value of its inputs and the current value of its output to determine the next value of its output. Sequential UDPs provide an easy and efficient way to model sequential circuits such as flip-flops and latches. A sequential UDP can model both level-sensitive and edge-sensitive behavior.

The maximum number of inputs to a combinational UDP is ten. The maximum number of inputs to a sequential UDP is limited to nine because the internal state counts as an input. Each UDP has exactly one output, which can be in one of three states: `0`, `1`, or `x`. The tri-state value `z` is not supported. In sequential UDPs, the output always has the same value as the internal state.

7.1 Memory Usage and Performance Considerations

The user should be aware of the amount of memory required for the internal tables created for evaluation of these UDPs during simulation. Although only one such table is required per UDP definition, and not for each instance, the UDPs with 8, 9, and 10 inputs do consume a large amount of memory. The trade-off here is one of speed versus memory. If many instances of a large UDP are needed, then it is easily possible to gain back the memory used by the definition, because each UDP instance can take less memory than that required for the group of gates it replaces.

The memory required for a UDP definition is given below:

Number of variables	Memory required (K bytes)
1-5	<1
6	5
7	17
8	56
9	187
10	623

Table 7-1: UDP memory requirements

Note that the number of variables is the number of inputs for combinational UDPs and the number of inputs plus one for sequential UDPs.

7.2 Syntax

The formal syntax of the UDP definition is as follows:

```
<UDP>
        ::= primitive <name_of_UDP> ( <output_terminal_name>,
                  <input_terminal_name> <,<input_terminal_name>>* ) ;
            <UDP_declaration>+
            <UDP_initial_statement>?
            <table_definition>
            endprimitive
<name_of_UDP>
        ::= <IDENTIFIER>
<UDP_declaration>
        ::= <UDP_output_declaration>
        ||= <reg_declaration>
        ||= <UDP_input_declaration>
<UDP_output_declaration>
        ::= output <output_terminal _name>;
<reg_declaration>
        reg <output_terminal_name> ;
<UDP_input_declaration>
        ::= input <input_terminal _name>
                  <,<input_terminal_name>>* ) ;
<UDP_initial_statement>
        ::= initial <output_terminal_name> = <init_val> ;
<init_val>
        ::= 1'b0
        ||= 1'b1
        ||= 1'bx
        ||= 1
        ||= 0
<table_definition>
        ::= table
              <table_entries>
              endtable
<table_entries>
        ::= <combinational_entry>+
        ||= <sequential_entry>+

                                        —continued
```

Syntax 7-1: Syntax for user-defined primitives

```
<combinational_entry>
        ::= <level_input_list> : <OUTPUT_SYMBOL> ;
<sequential_entry>
        ::= <input_list> : <state> : <next_state> ;
<input_list>
        ::= <level_input_list>
        ||= <edge_input_list>
<level_input_list>
        ::= <LEVEL_SYMBOL>+
<edge_input_list>
        ::= <LEVEL_SYMBOL>* <edge> <LEVEL_SYMBOL>*
<edge>
        ::= ( <LEVEL_SYMBOL> <LEVEL_SYMBOL> )
        ||= <EDGE_SYMBOL>
<state>
        ::= <LEVEL_SYMBOL>
<next_state>
        ::= <OUTPUT_SYMBOL>
        ||= -   (This is a literal hyphen—
                    see Section 7.15 for more details.)
Lexical tokens:
    <OUTPUT_SYMBOL> is one of the following:
        0  1  x  X
    <LEVEL_SYMBOL> is one of the following:
        0  1  x  X  ?  b  B
    <EDGE_SYMBOL> is one of the following:
        r  R  f  F  p  P  n  N  *
```

Syntax 7-1 continued: Syntax for user-defined primitives

7.3 UDP Definition

UDP definitions are independent of modules; they are at the same level as module definitions in the syntax hierarchy. They can appear anywhere in the source text, either before or after they are used inside a module. They **may not** appear between the keywords `module` and `endmodule`.

A UDP definition begins with the keyword `primitive`. This is followed by an identifier, which is the name of the UDP. This in turn is followed by a comma separated list of terminals enclosed in parentheses, which is followed by a semicolon.

The UDP definition header described previously is followed by terminal declarations and a state table. The UDP definition is terminated by the keyword `endprimitive`.

7.3.1 UDP Terminals

UDPs have multiple input terminals and exactly one output terminal; they cannot have bidirectional inout terminals.

The output terminal MUST be the first terminal in the terminal list.

All UDP terminals are scalar. No vector terminals are allowed.

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.

7.3.2 UDP Declarations

UDPs must contain input and output terminal declarations. The output terminal declaration begins with the keyword `output`, followed by one output terminal name. The input terminal declaration begins with the keyword `input`, followed by one or more input terminal names.

Sequential UDPs must contain a `reg` declaration for the output terminal. Combinational UDPs cannot contain a `reg` declaration. The initial value of the output terminal `reg` can be specified in an `initial` statement in a sequential UDP.

7.3.3 Sequential UDP `initial` Statement

The sequential UDP `initial` statement specifies the value of the output terminal when simulation begins. This statement begins with the keyword `initial`. The statement that follows must be an assignment statement that assigns a single bit literal value to the output terminal `reg`.

7.3.4 UDP State Table

The state table which defines the behavior of a UDP begins with the keyword `table` and is terminated with the keyword `endtable`.

Each row of the table is created using a variety of characters that indicate input and output states. Three states—`0`, `1`, and `x`—are supported. The `z` state is explicitly excluded from consideration in

user-defined primitives. A number of special characters are defined to represent certain combinations of state possibilities. These are detailed in this chapter, in Section 7.10, *Symbols to Enhance Readability*.

The order of the input state fields of each row of the state table is taken directly from the terminal list in the UDP definition header. It is NOT related to the order of the input declarations.

Combinational UDPs have one field per input and one field for the output. The input fields are separated from the output field by a colon.

Sequential UDPs have an additional field inserted between the input fields and the output field. This additional field represents the current state of the UDP and is considered equivalent to the current output value. It is delimited by colons.

Each row defines the output for a particular combination of input states. 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. Each row of the table is terminated by a semicolon.

Consider the following entry from a UDP state table:

```
0     1  : ? :   1    ;
```

In this entry the ? represents a don't-care condition—it is replaced by cases of the entry when the ? is replaced by 1, 0, and x. This specifies that when the inputs are 0 and 1, no matter what the value of the current state, the output is 1.

It is not necessary to explicitly specify every possible input combination. All combinations that are not explicitly specified result in a default output state of x.

It is illegal to have the same combination of inputs, including edges, specified for different outputs.

7.4 Combinational UDPs

In combinational UDPs, the output state is determined solely as a function of the current input states. Whenever an input changes state, the UDP is evaluated and one of the state table rows is matched. The output state is set to the value indicated by that row.

Consider the following example, which defines a multiplexer with two data inputs, a control input. Remember, there can only be a single output.

```
primitive multiplexer(mux, control, dataA, dataB ) ;
    output mux ;
    input control, dataA, dataB ;
    table
    // control dataA dataB  mux
           0     1     0  :  1  ;
           0     1     1  :  1  ;
           0     1     x  :  1  ;
           0     0     0  :  0  ;
           0     0     1  :  0  ;
           0     0     x  :  0  ;

           1     0     1  :  1  ;
           1     1     1  :  1  ;
           1     x     1  :  1  ;
           1     0     0  :  0  ;
           1     1     0  :  0  ;
           1     x     0  :  0  ;

           x     0     0  :  0  ;
           x     1     1  :  1  ;
    endtable
endprimitive
```

Example 7-1: Combinational form of user-defined primitive

The first entry in the table above can be explained as follows: when `control` equals `0` and `dataA` equals `1` and `dataB` equals `0`, then output `mux` equals `1`.

All combinations of the inputs that are not explicitly specified will drive the output to the unknown value `x`. For example, in the table for `multiplexer` above (Example 7-1), the input combination `0xx(control=0, dataA=x, dataB=x)` is not specified. If this combination occurs during simulation, the value of output `mux` will become `x`.

To improve the readability, and to ease writing of the table, several special symbols are provided. A `?` represents iteration of the table entry over the values `0`, `1`, and `x` — a `?` generates cases of that entry where the `?` is replaced by a `0`, `1`, or `x`. It represents a don't-care condition on that input. Using `?`, the description of a multiplexer given in Example 7-1 can be abbreviated as implemented in Example 7-2.

```
primitive multiplexer(mux,control,dataA,dataB ) ;
    output mux ;
    input control, dataA, dataB ;
    table
   // control dataA dataB mux
          0     1     ?  : 1  ;   // ? = 0,1,x
          0     0     ?  : 0  ;
          1     ?     1  : 1  ;
          1     ?     0  : 0  ;

          x     0     0  : 0  ;
          x     1     1  : 1  ;
    endtable
endprimitive
```

Example 7-2: Special symbols in user-defined primitive

7.5 Level-Sensitive Sequential UDPs

Level-sensitive sequential behavior is represented the same way as combinational behavior, except that the output is declared to be of type `reg`, and there is an additional field in each table entry. This new field represents the current state of the UDP.

The output field in a sequential UDP represents the next state.

Consider the example of a latch in Example 7-3.

```
primitive latch(q, clock, data) ;
    output q; reg q ;
    input clock, data;
    table
        // clock data   q     q+
             0     1  : ? :   1   ;
             0     0  : ? :   0   ;
             1     ?  : ? :   -   ; // - = no change
    endtable
endprimitive
```

Example 7-3: UDP for a latch

This description differs from a combinational UDP model in two ways. First, the output identifier `q` has an additional `reg` declaration to indicate that there is an internal state `q`. The output value of the UDP is always the same as the internal state. Second, a field for the current state, which is separated by colons from the inputs and the output, has been added.

7.6 Edge-Sensitive UDPs

In level-sensitive behavior, the values of the inputs and the current state are sufficient to determine the output value. Edge sensitive behavior differs in that changes in the output are triggered by specific transitions of the inputs. This makes the state table a transition table as illustrated in Example 7-4.

```
primitive d_edge_ff(q, clock, data);
output q; reg q;
input clock, data;
table
// obtain output on rising edge of clock
// clock   data   q     q+
   (01)     0   :  ?  :  0  ;
   (01)     1   :  ?  :  1  ;
   (0?)     1   :  1  :  1  ;
   (0?)     0   :  0  :  0  ;
// ignore negative edge of clock
   (?0)     ?   :  ?  :  -  ;
// ignore data changes on steady clock
   ?     (??)   :  ?  :  -  ;
    endtable
endprimitive
```

Example 7-4: UDP for an edge-sensitive D-type flip-flop

Example 7-4 has terms like (`01`) in the input fields. These terms represent transitions of the input values. Specifically, (`01`) represents a transition from `0` to `1`. The first line in the table of the previous UDP

definition (Example 7-4) can be interpreted as follows: when clock changes value from 0 to 1 and data equals 0, the output goes to 0 no matter what the current state.

Please note: Each table entry can have a transition specification on, at most, one input. Entries such as the one shown below are illegal:

```
(01)(01)0 : 0 : 1
```

As in the combinational and the level-sensitive entries, a ? implies iteration of the entry over the values 0, 1, and x. A dash (–) in the output column indicates no value change.

All unspecified transitions default to the output value x. Thus, in the previous example, transition of clock from 0 to x with data equal to 0 and current state equal to 1 will result in the output q going to 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.

7.7 Sequential UDP Initialization

The value on the output terminal of a sequential UDP can be specified with an initial statement that contains a procedural assignment statement. The initial statement is optional.

Like initial statements in modules, the initial statement in UDPs begin with the keyword initial. The valid contents of initial statements in UDPs and the valid left and right-hand sides of their procedural assignment statements differ from initial statements in modules. The difference between these two types of initial statements is described in Table 7-2.

initial statements in UDPs	initial statements in modules
contents limited to one procedural assignment statement	contents can be one procedural statement of any type or a block statement that contains more than one procedural statement
the procedural assignment statement must assign a value to a reg whose identifier matches the identifier of an output terminal	procedural assignment statements in initial statements can assign values to a reg whose identifier does not match the identifier of an output terminal
the procedural assignment statement must assign one of the following values: 1'b1 1'b0 1'bx 1 0	procedural assignment statements can assign values of any size, radix, and value

Table 7-2: `Initial` statements in UDPs and modules

Example 7-5 shows a sequential UDP that contains an `initial` statement.

```
primitive srff (q,s,r);
output q;
input s,r;
reg q;
initial q = 1'b1;
table
//   s   r   q   q+
     1   0 : ? : 1 ;
     f   0 : 1 : - ;
     0   r : ? : 0 ;
     0   f : 0 : - ;
     1   1 : ? : 0 ;
endtable
endprimitive
```

sequential UDP initial statement specifies that output terminal q has a value of 1 at the start of the simulation

Example 7-5: Sequential UDP `initial` statement

In Example 7-5, the output `q` has an initial value of 1 at the start of the simulation; a delay specification in the UDP instance does not delay the simulation time of the assignment of this initial value to the output. When simulation starts, this value is the current state in the state table.

Please note: Verilog-XL does not have an initialization or power-up phase. The initial value on the output to a sequential UDP does not propagate to the design output before simulation starts. All nets in the fanout of the output of a sequential UDP begin with a value of `x` even when that output has an initial value of `1` or `0`.

The following example and figure show how values are applied in a module that instantiates a sequential UDP with an `initial` statement. Example 7-6 shows the source description for the module and UDP.

```
primitive dff1 (q,clk,d);
input clk,d;
output q;
reg q;
initial
   q = 1'b1;

table
// clkd    q     q+
   p  0  :  ?  :  0  ;
   p  1  :  ?  :  1  ;
   n  ?  :  ?  :  -  ;
   ?  *  :  ?  :  -  ;
endtable
endprimitive

module dff (q,qb,clk,d);
input clk,d;
output q,qb;
   dff1   g1 (qi,clk,d);
   buf #3 g2 (q,qi);
   not #5 g3 (qb,qi);
endmodule
```

initial statement

UDP instance output is *qi*

q and *qb* are in the fanout of *qi*

Example 7-6: Instance of a sequential UDP with an `initial` *statement*

In Example 7-6, UDP `dff1` contains an `initial` statement that sets the initial value of its output to `1`. Module `dff` contains an instance of UDP `dff1`. In this instance, the UDP output is `qi`; the output's fanout includes nets `q` and `qb`.

Figure 7-1 shows the schematic of the module in Example 7-6 and the simulation times of the propagation of the initial value of the output of the UDP.

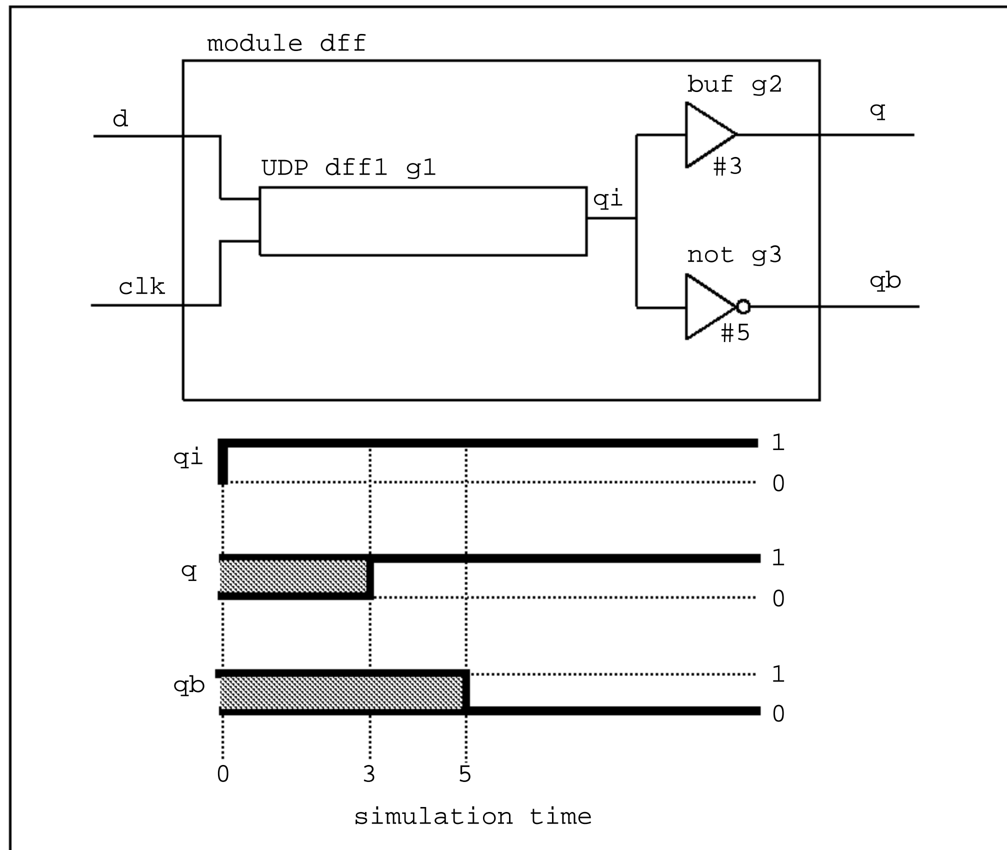

Figure 7-1: Module schematic and the simulation times of initial value propagation

In Figure 7-1, the fanout from the UDP output `qi` includes nets `q` and `qb`. At simulation time `0`, `qi` changes value to `1`. That initial value of `qi` does not propagate to net `q` until simulation time `3`, and does not propagate to net `qb` until simulation time `5`.

7.8 UDP Instances

Instances of user-defined primitives are specified inside modules in the same manner as for gates. The instance name is optional, just as for gates. The terminal order is as specified in the UDP definition. Only two delays can be specified, because z is not supported for UDPs.

The system can generate names for unnamed instances of UDPs. See Section 12.6 for more information on automatic naming.

Example 7-7 creates an instance of the D-type flip-flop d_edge_ff (defined in Example 7-4).

```
module flip;
   reg clock , data ;
   parameter p1 = 10 ;
   parameter p2 = 33;
   d_edge_ff #(5,7) d_inst( q, clock, data);
initial
begin
   data = 1;
   clock = 1;
end
always #p1 clock = ~clock;
always #p2 data = ~data;
endmodule
```

Example 7-7: UPD for a D-type flip-flop

7.9 Compilation

Several checks are applied to user-defined primitive definitions as they are compiled.

The table entries are checked for consistency. This means that if two entries specify different outputs for the same combination of inputs, including edges, an error will result. Special care should be taken when using the ?, b, *, p, and n symbols which are described in the next section.

The table entries are checked for redundancy. If two or more table entries specify the same output for the same combination of inputs, including edges, a warning will result. The message indicates the entry that duplicates what is specified in previous lines.

7.10 Symbols to Enhance Readability

Like ?, there are several symbols that can be used in UDP definitions to make the description more readable. The symbols described in Table 7-3 are used in Example 7-8.

Symbol	Interpretation	Explanation
b	0 or 1	like ?, except x is excluded
r	(01)	rising edge on an input
f	(10)	falling edge on an input
p	(01) or (0x) or (x1) or (1z) or (z1)	rising edges, including unknown
n	(10) or (1x) or (x0) or (0z) or (z0)	falling edges, including unknown
*	(??)	all transitions

Table 7-3: Symbols for readability

7.11 Mixing Level-Sensitive and Edge-Sensitive Descriptions

UDP definitions allow a mixing of the level-sensitive and the edge-sensitive constructs in the same description. An edge-triggered JK flip-flop with asynchronous preset and clear needs such a mixture. Example 7-8 illustrates this concept.

```
primitive jk_edge_ff(q, clock, j, k, preset, clear);
    output q; reg q;
    input clock, j, k, preset, clear;
    table
    //clock jk pc state output/next state
        ?   ?? 01 : ? :   1  ; //preset logic
        ?   ?? *1 : 1 :   1  ;
        ?   ?? 10 : ? :   0  ; //clear logic
        ?   ?? 1* : 0 :   0  ;
        r   00 00 : 0 :   1  ; //normal clocking cases
        r   00 11 : ? :   -  ;
        r   01 11 : ? :   0  ;
        r   10 11 : ? :   1  ;
        r   11 11 : 0 :   1  ;
        r   11 11 : 1 :   0  ;
        f   ?? ?? : ? :   -  ;
        b   *? ?? : ? :   -  ; //j and k transition cases
        b   ?* ?? : ? :   -  ;
    endtable
endprimitive
```

Example 7-8: Sequential UDP for level-sensitive and edge-sensitive behavior

In this example, the preset and clear logic is level-sensitive. Whenever the preset and clear combination is `01`, the output has value `1`. Similarly, whenever the preset and clear combination has value `10`, the output has value `0`.

The remaining logic is sensitive to edges of the clock. In the normal clocking cases, the flip-flop is sensitive to the rising clock edge as indicated by an `r` in the clock field in those entries. The insensitivity to the falling edge of clock is indicated by a hyphen (-) in the output field (see Section 7.15) for the entry with an `f` as the value of clock. Remember that the desired output for this input transition must be specified to

avoid unwanted x values at the output. The last two entries show that the transitions in j and k inputs do not change the output on a steady low or high clock.

7.12 Reducing Pessimism

Three-valued logic tends to make pessimistic estimates of the output when one or more inputs are unknown. User-defined primitives can be used to reduce this pessimism. The following is an extension of the previous latch example illustrating reduction of pessimism.

```
primitive latch(q, clock, data)
    output q; reg q ;
    input clock, data ;

    table
//          clock data state output/next state
             0     1  : ? :   1   ;
             0     0  : ? :   0   ;
             1     ?  : ? :   -   ; // - = no change
//         ignore x on clock when data equals state
             x     0  : 0 :   -   ;
             x     1  : 1 :   -   ;

    endtable
endprimitive
```

Example 7-9: Latch UDP illustrating pessimism

The last two entries specify what happens when the clock input has value x. If these are omitted, the output will go to x whenever the clock is x. This is a pessimistic model, as the latch should not change its output if it is already 0 and the data input is 0. Similar analysis is true for the situation when the data input is 1 and the current output is 1.

Consider the `jk` flip-flop with preset and clear in Example 7-10.

```
primitive jk_edge_ff(q, clock, j, k, preset, clear);
    output q; reg q;
    input clock, j, k, preset, clear;
    table
    //clock jk pc state output/next state
    //preset logic
        ?   ?? 01 : ? :   1 ;
        ?   ?? *1 : 1 :   1 ;
    //clear logic
        ?   ?? 10 : ? :   0 ;
        ?   ?? 1* : 0 :   0 ;
    //normal clocking cases
        r   00 00 : 0 :   1 ;
        r   00 11 : ? :   - ;
        r   01 11 : ? :   0 ;
        r   10 11 : ? :   1 ;
        r   11 11 : 0 :   1 ;
        r   11 11 : 1 :   0 ;
        f   ?? ?? : ? :   - ;
    //j and k cases
        b   *? ?? : ? :   - ;
        b   ?* ?? : ? :   - ;
    //cases reducing pessimism
        p   00 11 : ? :   - ;
        p   0? 1? : 0 :   - ;
        p   ?0 ?1 : 1 :   - ;
        (?0)?? ?? : ? :   - ;
        (1x)00 11 : ? :   - ;
        (1x)0? 1? : 0 :   - ;
        (1x)?0 ?1 : 1 :   - ;
        x   *0 ?1 : 1 :   - ;
        x   0* 1? : 0 :   - ;
    endtable
endprimitive
```

Example 7-10: UDP for a JK flip-flop with preset and clear

This example has additional entries for the positive clock (`p`) edges, the negative clock edges (`?0` and `1x`), and with the clock value `x`. In all of these situations, the output is deduced to remain unchanged rather than going to `x`. Thus, this model is less pessimistic than the previous example.

7.13 Level-Sensitive Dominance

In the Verilog HDL, edge-sensitive cases are processed first, followed by level-sensitive cases. When level-sensitive and edge-sensitive cases specify different output values, the result is specified by the level-sensitive case. The following table shows level-sensitive and edge-sensitive entries in Example 7-10, their level-sensitive or edge-sensitve behavior, and a case that each includes.

entry	included case	behavior
`?  ?? 01: ?: 1;`	`0  00 01: 0: 1;`	level-sensitive
`f  ?? ??: ?: -;`	`f  00 01: 0: 0;`	edge-sensitive

Table 7-4: The level-sensitive and edge-sensitive entries in Example 7-10

The included cases specify opposite next state values for the same input and current state combination.

The level-sensitive included case specifies that when the inputs `clock`, `jk` and `pc` values are `0  00 01`, and the current state is `0`, the output changes to `1`.

The edge-sensitive included case specifies that when `clock` falls from `1` to `0`, and the other inputs `jk` and `pc` are `00 01`, and the current state is `0`, the output changes to `0`.

When the edge-sensitive case is processed first, followed by the level-sensitive case, the output changes to `1`.

7.14 Processing of Simultaneous Input Changes

When multiple UDP inputs change at the same simulation time the UDP will be evaluated multiple times, once per input value change. This situation cannot be detected by any form of table entry. This fact has important implications for modeling sequential circuits where the order of input changes and subsequent UDP evaluations can have a profound effect on the results of the simulation.

Consider the D-type flip-flop in Example 7-11.

```
primitive d_edge_ff(q, clock, data);
output q; reg q;
input clock, data;
table
// obtain output on rising edge of clock
// clock   data   q     q+
   (01)     0   :  ?  :  0  ;
   (01)     1   :  ?  :  1  ;
   (0?)     1   :  1  :  1  ;
   (0?)     0   :  0  :  0  ;
// ignore negative edge of clock
   (?0)     ?   :  ?  :  -  ;
// ignore data changes on steady clock
   ?     (??)   :  ?  :  -  ;
    endtable
endprimitive
```

Example 7-11: D-type flip-flop

If the current state of the flip-flop is 0 and the clock and data inputs make transitions from 0 to 1 at the same simulation time, then the state of the output at the next simulation time is unpredictable because it cannot be predicted which of these transitions is processed first.

If the clock input transition is processed first and the data input transition is processed second, then the next state of the output will be 0. Likewise, if the data input transition is processed first and the clock transition is processed second, then the next state of the output will be 1.

This fact should be taken into consideration when constructing models. Keep in mind that gate-level models have the same sort of unpredictable behavior given particular input transition sequences; event-driven simulation is subject to idiosyncratic dependence on the order in which events are processed.

Timing checks can be used to detect simultaneous input transitions, provide a warning, and affect the simulation results; see Chapter13, *Specify Blocks*.

7.15 Summary of Symbols

The following table summarizes the meaning of all the value symbols that are valid in the table part of a UDP definition.

Symbol	Interpretation	Notes
0	logic 0	
1	logic 1	
x	unknown	
?	iteration of 0, 1, and x	cannot be given in output field
b	iteration of 0 and 1	cannot be given in output field
-	no change	can only be given in the output field of a sequential UDP
(vw)	value change from v to w	v and w can be any one of 0, 1, x, ? or b
*	same as (??)	any value change on input
r	same as (01)	rising edge on input
f	same as (10)	falling edge on input
p	iteration of (01), (0x), and (x1)	potential positive edge on the input
n	iteration of (10), (1x), and (x0)	potential Negative edge on the input

Table 7-5: UDP table symbols

7.16 Examples

The following examples show UDP modeling for an `and-or` gate, a majority function for carry, and a 2-channel multiplexor with storage.

```
// Description of an AND-OR gate.
// out = (a1 & a2 & a3) | (b1 & b2).
primitive and_or(out, a1,a2,a3, b1,b2);
    output out;
    input a1,a2,a3, b1,b2;
    table
    //   a   b  : out ;
        111 ?? :  1  ;
        ??? 11 :  1  ;
        0?? 0? :  0  ;
        0?? ?0 :  0  ;
        ?0? 0? :  0  ;
        ?0? ?0 :  0  ;
        ??0 0? :  0  ;
        ??0 ?0 :  0  ;
    endtable
endprimitive
```

Example 7-12: UDP for an `and-or` gate

```
// Majority function for carry
// carryout = (a & b) | (a & carryin) | (b & carryin)
primitive carry(carryout, carryin, a, b);
    output carryout;
    input carryin, a, b;
    table
        0 00 : 0;
        0 01 : 0;
        0 10 : 0;
        0 11 : 1;
        1 00 : 0;
        1 01 : 1;
        1 10 : 1;
        1 11 : 1;
        // the following cases reduce pessimism
        0 0x : 0;
        0 x0 : 0;
        x 00 : 0;
        1 1x : 1;
        1 x1 : 1;
        x 11 : 1;
    endtable
endprimitive
```

Example 7-13: UDP for a majority function for carry

```
// Description of a 2-channel multiplexer with storage.
// The storage is level sensitive.

primitive mux_with_storage(out,clk,control,dataA,dataB);
    output out;
    reg out;
    input clk, control, dataA, dataB;

    table
    //clk control dataA dataB : current-state : next state ;
       1     0       1     ?   :       ?       :     1      ;
       1     0       0     ?   :       ?       :     0      ;
       1     1       ?     1   :       ?       :     1      ;
       1     1       ?     0   :       ?       :     0      ;
       1     x       0     0   :       ?       :     0      ;
       1     x       1     1   :       ?       :     1      ;
       0     ?       ?     ?   :       ?       :     -      ;
       x     0       1     ?   :       1       :     -      ;
       x     0       0     ?   :       0       :     -      ;
       x     1       ?     1   :       1       :     -      ;
       x     1       ?     0   :       0       :     -      ;
    endtable

endprimitive
```

Example 7-14: UDP for a 2-channel multiplexor with storage