Appendix B

Modeling with Verilog Primitives and UDPs

cadence

This module focuses on how to model component functionality with the built-in Verilog primitives.

Appendix Objective

In this appendix, you will:

- Use built-in primitives and continuous assignments to model logic and define combinational and sequential primitives (UDPs)

Topics

- Modeling drive and charge strength
- Modeling gate and net delays
- Choosing primitive types
- Built-in primitives
- User-defined primitives
- Example UDPs

cadence

This appendix explores the use of built-in primitives and continuous assignments to model detailed combinational behavior down to the switch level. It examines the built-in primitives, how to model primitive drive strength and capacitive net charge strength, and how to model propagation delays across primitives and nets.

Specifying Drive and Charge Strength

For most gates you can specify a drive strength:

- Drive
  - **highz**, **weak**, **pull**, **strong**, **supply**
- Charge
  - **small**, **medium**, **large**
  - Apply only to **trireg** nets

Example

```
(weak1, large0)
```

Level	Name	Mnemonic
7	supply1	Su1
6	strong1	St1
5	pull1	Pu1
4	large1	La1
3	weak1	We1
2	medium1	Me1
1	small1	Sm1
0	highz1	Hi1
0	highz0	Hi0
1	small0	Sm0
2	medium0	Me0
3	weak0	We0
4	large0	La0
5	pull0	Pu0
6	strong0	St0
7	supply0	Su0

To facilitate your modeling of gate-level and switch-level circuits in the digital realm, you can specify delays for nets and primitives, drive strength for continuous assignments and primitives, and a charge strength for trireg nets.

Primitives and continuous assignments drive values with one of the supply, strong, pull, weak, or high-impedance strengths. They can, and often do, drive the 0 and 1 values with different strengths. In fact, it is illegal to drive both the 0 and the 1 values with the same high-impedance strength.

The trireg net can have a charge strength. A trireg charge strength together with its decay time models net capacitance.

Although the table lists discrete strength levels, for understanding the resolution of net values it is helpful to think of the driving value as occupying a range of strengths. This range can, in the case of an unknown value, overlap both the zero strengths and the one strengths.

Specifying Gate and Net Delays

Number of Values	#(delay)	#(rise, fall)	#(rise, fall, off)
Transition to 0	delay	fall	fall
Transition to 1	delay	rise	rise
Transition to Z	delay	min (rise, fall)	off
Transition to X	delay	min (rise, fall)	min (rise, fall, off)

```
and #(1,2) (out1, in1, in2, in3); // rise, fall
bufif0 #(3,4,5) (out2, in, ctrl); // rise, fall, turn-off
or #(3.2:4.0:6.3) (out, in1, in2); // min:typ:max
not #(1:2:3, 2:3:5) (o,in); // min:typ:max for rise, fall
```


cadence

The propagation delay of a gate or net is by default 0.

You can specify rise, fall, and turn-off delays:

- Rise delays apply to transitions ending in 1.
- Fall delays apply to transitions ending in 0.
- Turn-off delays apply to transitions ending in the high-impedance state.

Turn-off delay does not apply to primitive outputs that do not “turn off”.

You can specify single values, or a triplet of minimum, typical, and maximum values for each delay.

Any change in the inputs is reflected at the output only after the specified delay.

If you do not provide unique delays for transitions to the high-impedance or unknown states, then the worst case delays are used.

Choosing Primitive Types

n-input	n-output	3-state	Pull	MOS Switches	Bidirectional Switches
and nand	buf not	bufif0 notif0	pullup pulldown	nmos rnmos	tranif0 rtranif0
or nor		bufif1 notif1		pmos rpmos	tranif1 rtranif1
xor xnor				cmos rcmos	tran rtran

- Most primitives can have an optional strength specification (more later).
- Most primitives can have an optional delay specification (more later).
- Individual primitive instances do not need an instance name.
- Primitives do not have named ports – upon instantiation, connections to output terminals precede connections to input terminals.

This table lists all the Verilog built-in primitives. Following pages describe each of these categories in more detail.

You can optionally specify a drive strength for an instance of any of these primitives except the switches. For gate-level modeling, specifying the gate drive strength more accurately models the value of a net driven by multiple devices having differently sized output transistors.

For the switch primitives, you instead select between a non-resistive switch and its resistive counterpart. The non-resistive switches generally pass the strength of the input value through to the output. The resistive switches generally reduce the strength one level from input to output. Exceptions to this general statement exist for both the non-resistive switches and the resistive switches.

You can optionally specify a delay for an instance of any of these primitives except the pullup or pulldown primitives and the uncontrolled bidirectional pass switches. The output of the pullup and pulldown primitives does not transition so cannot have a delay.

For individual primitive instances, you can omit the instance name. This is because primitives have no named internal items for you to access, so do not need a scope name.

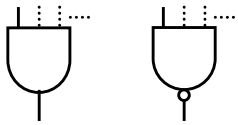
Primitives do not have named ports. Upon instantiation, you make ordered connections to output terminals followed by ordered connections to input terminals.

n-input Primitives

```
type [strength] [delay] [name] ( output, inputs… );
and (pull1,weak0) #(1,2) a1 (out, in1, in2);
```

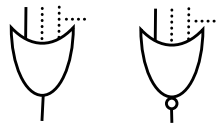
and / nand

```
and (o, i1, i2);
```



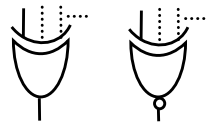
or / nor

```
or (o, i1, i2);
```



xor / xnor

```
xor (o, i1, i2);
```



The n-input gates have one output and one or more inputs and perform the expected logical function.

These gates by default strongly drive their outputs. You can optionally specify the supply, strong, pull, weak or high-impedance strengths. You specify the strengths in pairs, so if you need a non-default strength for one value then you need to also specify a strength for the other value. You can specify the high-impedance strength for only one of the values, as it makes no sense to instantiate a gate that does not drive either value.

The gates by default have no propagation delay. You can optionally specify either one or two propagation delays for the rise and fall transitions. If you specify one delay, then it applies to all transitions. If you specify two delays, then the minimum of the two delays applies to transitions to the unknown state.

n-output Primitives

```
type [strength] [delay] [name] ( outputs…, input );
buf (pull1,weak0) #(1,2) b1 (out1, out2, in);
```

buf

```
buf (o, i);
```

not

```
not (o, i);
```

The n-output gates have one or more outputs and one input and perform the expected logical function.

These gates by default strongly drive their outputs. You can optionally specify the supply, strong, pull, weak or high-impedance strengths. You specify the strengths in pairs, so if you need a non-default strength for one value then you need to also specify a strength for the other value. You can specify the high-impedance strength for only one of the values, as it makes no sense to instantiate a gate that does not drive either value.

The gates by default have no propagation delay. You can optionally specify either one or two propagation delays for the rise and fall transitions. If you specify one delay, then it applies to all transitions. If you specify two delays, then the minimum of the two delays applies to transitions to the unknown state.

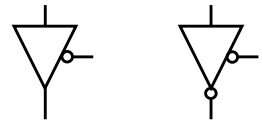
Having multiple outputs permits you to individually model the characteristics of the interconnect between each output and its load.

3-state Primitives

```
type [strength] [delay] [name] ( output, input, control );
bufif0 (pull1,strong0) #(1,2,3) b1 (out, in, en_);
```

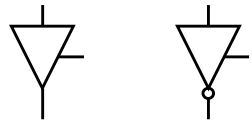
bufif0 / notif0

```
bufif0 (o, i, e_);
```



bufif1 / notif1

```
bufif1 (o, i, e);
```



bufif0		CONTROL			
		0	1	Z	X
D	0	0	Z	L	
A	1	1	Z	H	
T	Z	X	Z	X	
A	X	X	Z	X	X

0 or Z but definitely not 1

1 or Z but definitely not 0

The 3-state gates have an extra enable input. When not enabled, the output of these gates float to the high-impedance state.

These gates by default strongly drive their outputs. You can optionally specify the supply, strong, pull, weak or high-impedance strengths. You specify the strengths in pairs, so if you need a non-default strength for one value then you need to also specify a strength for the other value. You can specify the high-impedance strength for only one of the values, as it makes no sense to instantiate a gate that does not drive either value.

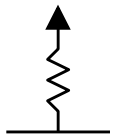
The gates by default have no propagation delay. You can optionally specify either one, two or three propagation delays for the rise, fall and turn-off transitions. If you specify one delay, then it applies to all transitions. If you specify two delays, then the minimum of the two delays applies to transitions to the high-impedance or unknown states.

Pullup and Pulldown Primitives

```
type [strength] [name] ( output );
pullup (weak1) p1 (out);
```

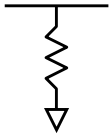
pullup

```
pullup (o);
```



pulldown

```
pulldown (o);
```



no delay!

cadence

The pull gates have one output and no inputs.

These gates by default drive their outputs with a pull strength. You can optionally specify the supply, strong, or weak strengths.

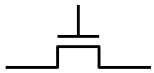
The gates have no propagation delay and you cannot specify any.

Unidirectional Switches

```
type [delay] [name] ( output, input, control );
type [delay] [name] ( output, input, ncontrol, pcontrol );
```

nmos / rnmos

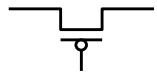
```
nmos (o, i, e);
```



nmos		CONTROL			
		0	1	Z	X
D	0	Z	0	L	
A	1	Z	1	H	H
T	Z	Z	Z	Z	Z
A	X	Z	X	X	X

pmos / rpmos

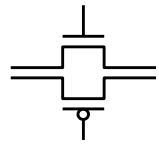
```
pmos (o, i, e_);
```



pmos		CONTROL			
		0	1	Z	X
D	0	0	Z	L	L
A	1	1	Z	H	H
T	Z	Z	Z	Z	Z
A	X	X	Z	X	X

cmos / rcmos

```
cmos (o, i, e, e_);
```



- No strength specification!
- Non-resistive reduce *supply* to *strong*
- Resistive versions reduce strengths 2-6 by 1

cadence

The truth tables for the nmos and pmos switches are similar to those for the bufif1 and bufif0 with the exception that they also pass high impedance inputs through to the output.

You cannot specify a drive strength for the switch primitives. You can instantiate a non-resistive switch that passes value strengths from input to output except that it reduces supply strength to strong strength, or a resistive switch that reduces value strengths from input to output by one level except that it reduces supply strengths by two levels and does not reduce the weak capacitance or high-impedance strengths.

The gates by default have no propagation delay. You can optionally specify either one, two or three propagation delays for the rise, fall and turn-off transitions. If you specify one delay then it applies to all transitions. If you specify two delays then the minimum of the two delays applies to transitions to the high-impedance or unknown states. If you specify three delays then the minimum of the three delays applies to transitions to the unknown state.

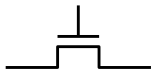
A cmos gate acts exactly as if you connect a nmos gate and pmos gate in parallel.

Bidirectional Switches

```
type [delay] [name] ( inout, inout, control );
type [name] ( inout, inout );
```

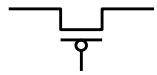
tranif1 / rtranif1

```
tranif1 (w1, w2, e);
```



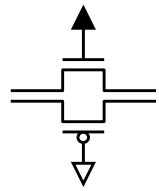
tranif0 / rtranif0

```
tranif0 (w1, w2, e_);
```



tran / rtran

```
tran (w1, w2);
```



Delay applies only to *control* input
no strength specification!

- Non-resistive reduce *supply* to *strong*
- Resistive versions reduce strengths 2-6 by 1

cadence

The bidirectional switches are also known as “pass” gates, as they pass values from either inout to either inout. Both inout terminals must connect to nets.

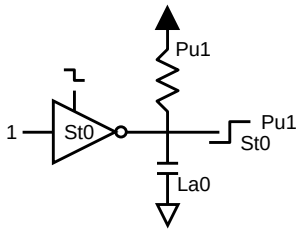
You cannot specify a drive strength for the switch primitives. You can instantiate a non-resistive switch that passes value strengths from input to output except that it reduces supply strength to strong strength, or a resistive switch that reduces value strengths from input to output by one level except that it reduces supply strengths by two levels and does not reduce the weak capacitance or high-impedance strengths.

The gates by default have no delay. You can optionally specify either one or two delays for the turn-on and turn-off transitions, but you cannot specify a delay for the data. If you specify one delay then it applies to all control-to-output transitions. If you specify two delays then the first is the turn-on delay and the second is the turn-off delay and the minimum of the two delays applies to control-to-output transitions to the high-impedance or unknown states.

Resolving Values of Unambiguous Strengths

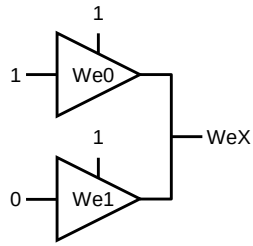
The one driver that is stronger than all the others "wins".

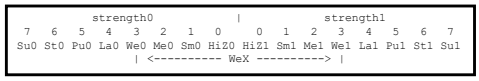
```
trireg (large) n;
pullup (n);
notif1 (n,1,en);
```



Resolution of equal strengths of opposite values produces a range of strengths.

```
bufif1 (weak0,weak1) (n,1,1);
bufif1 (weak0,weak1) (n,0,1);
```





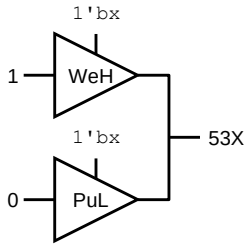
Where multiple sources each have a known value and a single strength, and one source is stronger than the others, that stronger source dominates the others and the result is the value and strength of the dominant source.

Two equal strongest drivers of opposite values produce a value that is unknown and having the range between the two strengths. For value resolution purposes a strength must always be either just one strength or just one continuous range of strengths.

Resolving Values of Ambiguous Strengths

Resolution of multiple ambiguous strengths produces a range from the strongest strength0 to the strongest strength1.

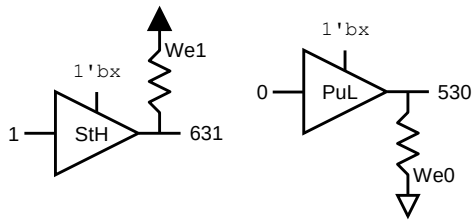
```
bufif1 (weak0,weak1) (n,1,1'bx);
bufif1 (pull0,pull1) (n,0,1'bx);
```




```
          strength0           |           strength1
  7   6   5   4   3   2   1   0     0   1   2   3   4   5   6   7
Su0 St0 Pu0 La0 We0 Me0 Sm0 HiZ0 HiZ1 Sm1 Me1 We1 La1 Pu1 St1 Su1
                                | <--- WeH --> |
        | <------- PuL ------> |
        | <--------------- 53X -------------> |
```

Resolution of ambiguous strengths with unambiguous strengths removes the weaker ambiguous strength components.

```
bufif1                  (n,1,1'bx);
pullup   (weak1)        (n);
bufif1   (pull0,pull1)  (m,0,1'bx);
pulldown (weak0)        (m)
```




```
          strength0           |           strength1
  7   6   5   4   3   2   1   0     0   1   2   3   4   5   6   7
Su0 St0 Pu0 La0 We0 Me0 Sm0 HiZ0 HiZ1 Sm1 Me1 We1 La1 Pu1 St1 Su1
                                | <-------- StH ---------> |
                                              | <- 631 -> |
        | <------- PuL ------> |
        |<-530->|
```

529 © Cadence Design Systems, Inc. All rights reserved.

cadence

Resolution of multiple ambiguous strengths produces a range from the strongest strength0 to the strongest strength1. Here we have a wire driven by a first buffer that because its control is unknown outputs a weak strength high and a second buffer that because its control is unknown outputs a pull strength low. The result is an unknown value between a pull strength0 and a weak strength1. The short notation for this is 53X – the strength0 followed by the strength1 followed by the value.

Resolution of ambiguous strengths with unambiguous strengths removes the weaker ambiguous strength components. Here we have a wire driven by a first buffer that because its control is unknown outputs a strong strength high, but because the net is also weakly pulled up, the net has no strength components below the weak strength. We also have a wire driven by a second buffer that because its control is unknown outputs a pull strength low, but because the net is also weakly pulled down, the net has no strength components below the weak strength.

If the strength is ambiguous but the value is not ambiguous, then the short notation is the strong strength followed by the weak strength followed by the value.

Reference: Drive Strengths and Charge Strengths

Type(s)	Optional Strength Spec	Comments
n-input, n-output, 3-state	**(*strength0*,*strength1*)** – or – **(*strength1*,*strength0*)**	strength0: **supply0 strong0 pull0 weak0 highz0** strength1: **supply1 strong1 pull1 weak1 highz1** Default is strong. One but not both can be highz.
pull	**(*strength0*,*strength1*)** – or – **(*strength1*,*strength0*)**	For pullup strength0 is ignored and can be omitted. For pulldown strength1 is ignored and can be omitted. Default is pull.
switch	–	Switches are not drivers.
continuous assign to scalar net	**(*strength0*,*strength1*)** – or – **(*strength1*,*strength0*)**	strength0: **supply0 strong0 pull0 weak0 highz0** strength1: **supply1 strong1 pull1 weak1 highz1** Default is strong. One but not both can be highz.
supply0, supply1	–	Strength is supply and cannot be changed.
tri0, tri1	–	Strength is pull and cannot be changed.
trireg net	**(*charge-strength*)**	charge strength: **large medium small**. Default is medium.

- It is the continuous assignment and not the net that gets the drive strength.
- A strength that appears in a declaration assignment is a drive strength.

cadence

You can specify a drive strength for a primitive instance or a continuous assignment to a scalar net or a scalar net declaration assignment:

- For the pullup and pulldown primitives, the default drive strength is the pull strength and you can specify the alternative supply, strong, weak or high-impedance strength. A strength you specify for a logic value it cannot drive is ignored.
- For the other nonswitch primitives and continuous assignments and net declaration assignments, the default drive strength is the strong strength and you can specify the alternative supply, pull, weak or high-impedance strength. You can specify a high-impedance strength for the 0 value or the 1 value but not for both the 0 and 1 values.
- You cannot specify a drive strength for the switch primitives.

You can specify a charge strength for a trireg net. The charge strength is the strength of the charge remaining on the net after all of its drivers turn off and before the charge decays. The default charge strength is the medium strength and you can specify the alternative large or small strength.

Reference: Gate and Net Delay

Type(s)	Optional Delay Spec	Comments
n-input, n-output	`#(delay)` `#(rise,fall)`	Can be just *dly* or *min*:*typ*:*max* Smallest delay to x

3-state,
CMOS,
MOS
	`#(delay)`
	`#(rise,fall)`
	`#(rise,fall,turn-off)`
		
		If 2 then smallest delay to z, x.
		If 3 then smallest delay to x.

pull	–	No delay specification
tran	–	No delay specification
tranif0, tranif1	`#(delay)` `#(on,off)`	Does not delay *data*. If 2 then smallest delay to x *from control*

trireg	`#(delay)`
	`#(rise,fall)`
	`#(rise,fall,decay)`
		If 1 then all transitions.
		If 2 then smallest delay to x.
		If 3 then 3rd is *decay* time to x.

other nets	`#(delay)`
	`#(rise,fall)`
	`#(rise,fall,turn-off)`
		If 1 then all transitions.
		If 2 then smallest delay to z, x.
		If 3 then smallest delay to x.

```
bufif0 (pull1,strong0) #(1,2,3) b1 (out, in, en_);
```

531

cadence

You can specify a delay value as a single expression or as a colon-separated list of three expressions for the minimum, typical and maximum delays. Simulators provide a way to select among these three delays:

- For the n-input and n-output primitives you can provide one delay or separate rise and fall delays. If you provide separate rise and fall delays then transitions to the unknown state use the smallest of the two delays.
- For the 3-state, CMOS and MOS primitives and most nets you can provide one delay or separate rise and fall delays or separate rise, fall and turn-off delays. If you provide separate rise and fall delays then transitions to the high-impedance state and unknown state use the smallest of the two delays. If you provide separate rise, fall and turn-off delays then transitions to unknown state use the smallest of the three delays.
- The pullup, pulldown and uncontrolled bidirectional pass switches do not have delays.
- The controlled bidirectional pass switches do not have delays from the data input but do have turn-on and turn-off delays from the control input. If you provide separate turn-on and turn-off delays then transitions to unknown state from the control input use the smallest of the two delays.
- For the trireg net, the third delay, if provided, is the decay time from the point where the net is not driven to point at which its value becomes unknown. You cannot drive a trireg net to the high-impedance value. If you drive a high-impedance or unknown value onto a trireg net then an unknown state appears on the output after the smallest of the first two delays.

Built-In Primitives

The Verilog language has 26 built-in primitives.

Category	Primitive	Connections
n-input	and nand or nor xor xnor	(output, inputs…)
n-output	buf not	(outputs…, input)
3-state	bufif0 notif0 bufif1 notif1	(output, input, control)
Pull	pullup pulldown	(output)
Unidirectional Switches	pmos nmos rpmos rnmos cmos rcmos	(output, input, control) (output, input, n-control, p-control)
Bidirectional Switches	tran rtran tranif0 tranif1 rtranif0 rtranif1	(inout1, inout2) (inout1, inout2, control)

cadence

The and, nand, nor, or, xnor, and xor primitives can have multiple inputs. The output is always the first terminal in the list of connections.

The buf and not primitives can have multiple outputs. The input is always the last terminal in the list of connections.

The bufif0, bufif1, notif0, and notif1 primitives can drive a high-impedance value.

The pullup primitive drives a pull-strength logic 1 and the pulldown primitive drives a pull-strength logic 0.

The nmos, pmos, rnmos, and rpmos primitives are unidirectional switches that can pass a high-impedance value from input to output as well as drive a high-impedance output.

The cmos and rcmos primitives are unidirectional switches that can pass a high-impedance value from input to output as well as drive a high-impedance output. The cmos primitive is like paired nmos and pmos devices. The rcmos device is like paired rnmos and rpmos devices.

The tran and rtran primitives are bidirectional switches that can pass a high-impedance value from either inout to the other inout.

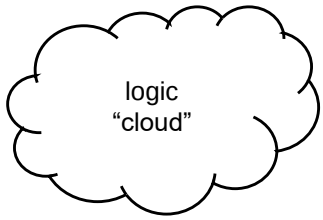
The tranif0, tranif1, rtranif0, and rtranif1 primitives are bidirectional switches that can pass a high-impedance value from either inout to the other inout, and also drive a high-impedance value.

The “resistive” versions of these primitives generally reduce the strength one level from input to output. They reduce supply strength by two levels and do not reduce the weak capacitance or high-impedance strengths.

User-Defined Primitives (UDPs)

Define your own primitive to take advantage of UDP features:

- You can enhance simulation performance by replacing a “glitchy” circuit of several built-in primitives with a single UDP.
- You can more accurately model the functionality of hardware because you have exact control over the output response to input changes.



versus

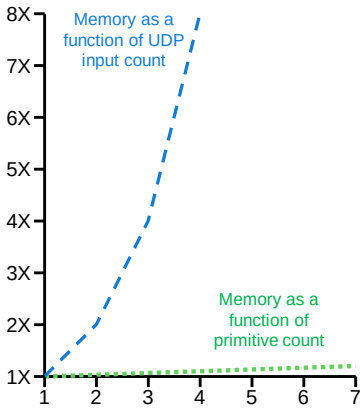
```
1010101010101010
0101010101010101
0101 lookup 1010
0101 table  0101
1010101010101010
0101010101010101
```


A combinational UDP can replace the logic of many primitives. This can reduce simulation time and memory requirements, which becomes more important if you instantiate the same logic numerous times.

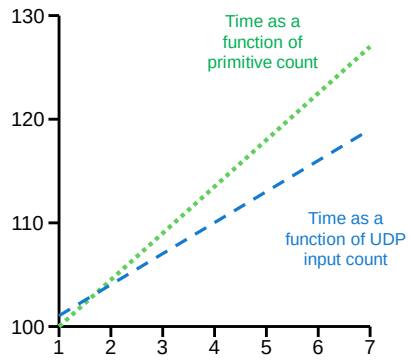
ASIC vendor simulation libraries typically utilize UDPs. This is okay because the simulation library itself will not be synthesized, and UDPs provide more accurate modeling of macro functionality than does behavioral RTL. Also, most of these libraries were at least initially prepared for use with the Verilog -XL simulator, which requires module and interconnect path timing to terminate at an acceleratable primitive.

UDPs and Platform Resources

Platform Memory Usage



Platform Time Usage



534

cadence

The UDP table size is exponentially proportional to the UDP input count.

The UDP table look-up time is constant regardless of the input count. The time to evaluate the inputs prior to look-up is linearly proportional to the input count. but increases more slowly than that of an equivalent circuit implemented with built-in primitives. This is because the simulator does not need to push otherwise wasted events through a cone of logic primitives.

Using a UDP is a trade of table memory for accuracy and performance. The memory disparity becomes less significant and the performance disparity becomes more significant when you instantiate very many UDPs of a few basic types that have perhaps up to five inputs. This is exactly what occurs in a postsynthesis netlist.

Defining a Primitive

This is what a user-defined primitive looks like:

- Define the UDP outside of any module description.
- Place the output first in the port list.
- Follow by up to ten inputs.
- For a sequential UDP:
  - Limited to 9 inputs
  - Declare the output type **reg**
  - Optionally initialize to 0, 1, or 1'bX

```
primitive name (output, inputs);
 output output;
 input input …;
 reg output;
 initial output = value;
 table
  . . .
 endtable
endprimitive
```


cadence

You declare a UDP in the definitions name space, that is, outside of and at the same level as other UDP declarations and module declarations.

Similarly to modules, you can use either the Verilog-1995 list of ports syntax or the Verilog-2001 list of port declarations syntax. Differently from modules, your one output port must appear first in the list, the list must not have inout ports, and you are limited to a maximum of ten input ports.

For a sequential UDP, that is, one that stores a value, the current value is considered one of the inputs, so you are limited to a maximum of nine input ports. You declare a register (reg) for the output of a sequential UDP, and you can optionally initialize the register to the 0, 1, or unknown value. You cannot initialize it to the high-impedance value and in fact a UDP cannot drive a high-impedance value.

Defining the Truth Table

This is what truth table entries for a sequential device may look like:

```
table
// d c : q : qnext
   0 r : ? : 0 ;
   1 r : ? : 1 ;
   ? n : ? : - ;
   * ? : ? : - ;
endtable
```

- Enter input states or transitions in the same order as the input port list.
- Then enter a colon, current state, colon, output state, and semicolon.
- The current state counts as one of the maximum of 10 inputs.

This is what truth table entries for a combination device may look like:

```
table
// s a b : y
   0 0 ? : 0 ;
   0 1 ? : 1 ;
   1 ? 0 : 0 ;
   1 ? 1 : 1 ;
endtable
```

- Enter input states in the same order as the input port list.
- Then enter a colon (:), the output state, and a semicolon (;).

cadence

Table row entries for a combinational device consist of input state values of 0, 1, X, or hook (“?”) and output state values of 0, 1, or X. The hook character input symbol indicates an iteration over all possible values, which is essentially a don’t-care situation.

The left table partially describes the logic of a multiplexor. For example, if the select input is 0 then the “a” input appears at the output. You will later see how to complete this table.

Table row entries for a sequential device may also contain input transitions from any of the input state values to any other of the input state values, and the output state value hyphen (“-”). The hyphen character output symbol indicates that this input transition does not change the output state.

The simulator does a separate table look-up for each transition of each changing input. For each table row you need to consider the effect of only a single input transition at a time.

The simulator acts upon the edge-sensitive entries first, then the level-sensitive entries, thus giving the level-sensitive entries higher priority.

Truth Table Symbols

Symbol	Values or Edges	Explanation
-		no change
?	0 or 1 or x	any value
b	0 or 1	any binary value
r	(01)	0->1 edge
f	(10)	1->0 edge
p	(01) or (0x) or (x1)	positive edge
n	(10) or (x0) or (1x)	negative edge
*	(??)	any edge

As the simulator does a separate table look-up for each transition of each changing input, you can think of the binary input states 0, 1, X as symbols themselves, for example the “0” input state really means either the 1-to-0 (10) or unknown-to-0 (X0) transition. For any input transition that the simulator does not find an output specification, the simulator places the unknown value on the output.

The hyphen (“-”) symbol applies only to the output and indicates that the input transition in this truth table row does not change the output.

The other symbols are each a short-cut for a set of input transitions. You can use the symbol or you can enter just exactly one input transition.

Example UDPs

You learn best by example, so here they are:

- Combinational example: 2-1 multiplexor
- Combinational example: full adder
- Level-sensitive sequential example: latch
  - Latch with low-active reset
- Edge-sensitive sequential example: D flip-flop
  - D flip-flop with low-active reset
  - D flip-flop using a notifier

cadence

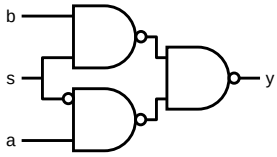
This section presents several examples of representative UDPs.

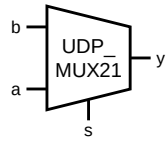
Combinational Example: 2-1 Multiplexor

- Define a UDP outside of any module.
- Place the output first in the port list.
- Make table entries in input port list order.
- Specify all combinations producing a known output.

Char	Iteration	Explanation
?	0 or 1 or x	any value

```
primitive UDP_MUX21 (y, s, a, b);
output y;
input s, a, b;
  table
  // s a b : O
     0 0 ? : 0; // select a
     0 1 ? : 1;
     1 ? 0 : 0; // select b
     1 ? 1 : 1;
     ? 0 0 : 0; // pessimism
     ? 1 1 : 1; // pessimism
  endtable
endprimitive
```





If the select input is 0, the output follows the a input regardless of the b input value.

If the select input is 1, the output follows the b input regardless of the a input value.

For every input combination for which you do not specify an output, the simulator makes the output unknown. You can almost always include table entries to reduce such pessimism. For this multiplexor, if the two data inputs have the same value, the output must assume that value regardless of the value of the select input. You cannot model this behavior with the built-in Verilog primitives.

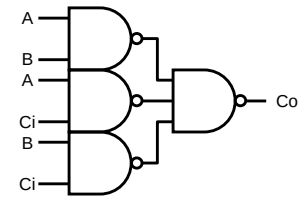
The table entries to reduce pessimism somewhat overlap previous table entries. Determine for example which two table rows match a situation where all inputs are low and which two table rows match a situation where all inputs are high. Such overlap is common when reducing pessimism and is permitted if all the overlapping rows specify the same output value.

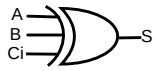
Combinational Example: Full Adder

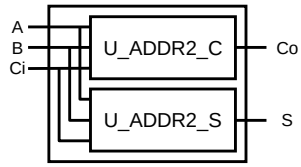
You can implement a full adder with only two combinational UDPs.

```
// FULL ADDER CARRY-OUT TERM
primitive U_ADDR2_C (Co,A,B,Ci);
output Co;
input A, B, Ci;
  table
  // A B Ci : Co
     1 1 ?  : 1;
     1 ? 1  : 1;
     ? 1 1  : 1;
     0 0 ?  : 0;
     0 ? 0  : 0;
     ? 0 0  : 0;
  endtable
endprimitive
```

```
// FULL ADDER SUM TERM
primitive U_ADDR2_S (S,A,B,Ci);
output S;
input A, B, Ci;
  table
  // A B Ci : S
     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
```







The carry output is high when the sum of the inputs is 2 or 3. The sum output is high when the sum of the inputs is 1 or 3.

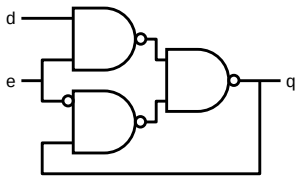
You can implement the full adder with two combinational UDPs to replace five built-in Verilog primitives. This removes the internal nodes, thus reducing the simulation database size and platform memory requirements. It also reduces the work the simulator must do to propagate events, thus the time to simulate the design.

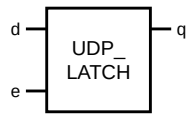
Level-Sensitive Sequential Example: Latch

- Declare the output type as **reg** if the UDP stores a value.
- You can use an **initial** statement to initialize the stored value.
- The current stored value is another table input.

Char	Iteration	Explanation
-		No change
?	0 or 1 or x	Any value

```
primitive UDP_LATCH (q, d, e);
output q; reg q;
input d, e;
  initial q = 1'b0;
  table
  // d e : q : qnext
     0 1 : ? : 0 ; // enabled 0
     1 1 : ? : 1 ; // enabled 1
     ? 0 : ? : - ; // disabled
     0 ? : 0 : - ; // pessimism
     1 ? : 1 : - ; // pessimism
  endtable
endprimitive
```





541 © Cadence Design Systems, Inc. All rights reserved.

cadence

A latch is logically a multiplexor with the output fed back to one of the inputs. If you model a latch with the built-in primitives then you waste computer and memory resources and your model is less than truly accurate and you many need to time propagation of the enable signal so that you do not lose data as you disable the latch.

For this example:

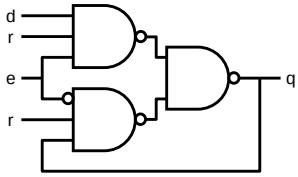
- If the device you are to model has power-up reset, then you can model the reset with an initial statement.
- If the e input is 1, then the q output follows the d input regardless of the current state.
- If the e input is 0, then the q output maintains its current state regardless of the d input.
- If the e input is unknown, then the output still does not change if the d input is the same as the current state. Including such scenarios reduces pessimism. If you omit such scenarios, then the simulator must set the output to the unknown state.

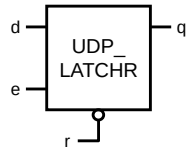
Latch with Low-Active Reset

Easiest to consider reset as just another level-sensitive input.

Char	Iteration	Explanation
-		No change
?	0 or 1 or x	Any value

```
primitive UDP_LATCHR (q, d, e, r);
output q; reg q;
input d, e, r;
  initial q = 1'b0;
  table
  // d e r : q : qnext
     ? ? 0 : ? : 0 ; // reset
     0 1 1 : ? : 0 ; // enabled 0
     1 1 1 : ? : 1 ; // enabled 1
     ? 0 1 : ? : - ; // disabled
     0 ? ? : 0 : - ; // pessimism
     1 ? 1 : 1 : - ; // pessimism
  endtable
endprimitive
```





542 © Cadence Design Systems, Inc. All rights reserved.

cadence

To add reset to the primitive latch we effectively gate off both multiplexor inputs. This is a situation where you must carefully read the vendor's documentation, as for some devices the reset has no effect while the latch is enabled.

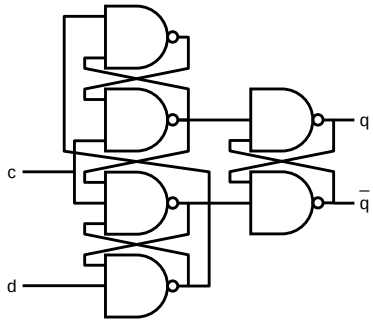
The truth table for this latch has an added column and row for the reset, and also has a slight modification to reduce pessimism when the reset state is unknown.

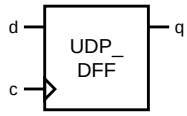
Edge-Sensitive Sequential Example: D Flip-Flop

Specify all transitions producing a known output – one transition per row.

Char	Edges	Explanation
r	(01)	0->1 edge
p	(01), (0x), (x1)	positive edge
n	(10), (x0), (1x)	negative edge
*	(??)	any edge

```
primitive UDP_DFF (q, d, c);
output q; reg q;
input d, c;
  table
  // d c : q : qnext
     0 r : ? : 0 ; // clk rise
     1 r : ? : 1 ; // clk rise
     0 p : 0 : - ; // clk posedge
     1 p : 1 : - ; // clk posedge
     ? n : ? : - ; // clk negedge
     * ? : ? : - ; // data edge
  endtable
endprimitive
```





543 © Cadence Design Systems, Inc. All rights reserved.

cadence

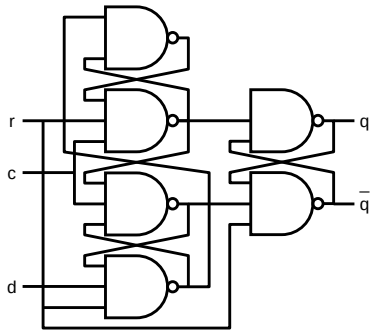
The simulator does a separate table look-up for each transition of each input. For each table row you need to consider the effect of only a single input transition:

- In a rising clock edge the output follows the data input.
- On any positive clock edge, if the data input matches the existing state, then the output does not change.
- For any negative clock edge the output does not change.
- For any data edge the output does not change.

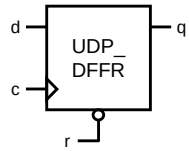
D Flip-Flop with Low-Active Reset

Easiest to consider as transitions rather than levels.

Char	Edges	Explanation
r	(01)	0->1 edge
p	(01), (0x), (x1)	positive edge
n	(10), (x0), (1x)	negative edge
*	(??)	any edge




```
primitive UDP_DFFR (q,d,c,r);
output q; reg q;
input d, c, r;
  table
  // d c r : q : qnext
     ? ? 0 : ? : 0; // reset
     ? ? n : 0 : -; // rst negedge
     ? ? p : ? : -; // rst posedge
     0 r ? : ? : 0; // clk rise
     1 r 1 : ? : 1; // clk rise
     0 p ? : 0 : -; // clk posedge
     1 p 1 : 1 : -; // clk posedge
     ? n ? : ? : -; // clk negedge
     * ? ? : ? : -; // data edge
  endtable
endprimitive
```



cadence

To add reset to the primitive D flip-flop we effectively gate off all cross-coupled nands.

The truth table for this D flip-flop has an added column and three added rows for the reset, and also has a slight modification to reduce pessimism when the reset state is unknown but we are clocking in low data anyway.

D Flip-Flop Using a Notifier

The following example is of a positive edge-triggered D flip-flop with an asynchronous reset, complete with timing checks and path delays. The model instantiates a UDP which has an input for the value of a notifier reg.

```
`timescale 1 ns / 1 ns
module dffr_m (q, d, clk, rst);
input d, clk, rst;
output q;
  reg nt;
  UDP_DFFR u1 (q, d, clk, rst, nt);
  specify
    specparam tS = 2;
    (clk => q) = (2:3:4);
    $setup (d, posedge clk, tS, nt);
  endspecify
endmodule
```

```
primitive UDP_DFFR (q, d, c, r, n);
output q; reg q;
input d, c, r, n;
  table
  // d c r n : q : qnext
     ? ? 0 ? : ? : 0; // reset
     ? ? n ? : 0 : -; // rst negedge
     ? ? p ? : ? : -; // rst posedge
     0 r ? ? : ? : 0; // clk rise
     1 r 1 ? : ? : 1; // clk rise
     0 p ? ? : 0 : -; // clk posedge
     1 p 1 ? : 1 : -; // clk posedge
     ? n ? ? : ? : -; // clk negedge
     * ? ? ? : ? : -; // data edge
     ? ? ? * : ? : x; // notifier
  endtable
endprimitive
```


A specify block defines input-to-output module timing paths and assigns delays to them and checks the relationship between edges of input ports. A specify block can appear only within a module definition and applies to all instances of the module. You can later annotate the timing on a per-instance basis. This training elsewhere describes specify blocks and annotation in more detail.

This specify block:

- Declares a specify parameter and assigns a value to it.
- Declares a module timing path between the clock input and the data output and assigns to it a timing triple.
- Enables a setup timing check between the data input and the clock rising edge.

All timing checks examine the relationship between edges of one or two signals and report a failure to meet the specified limit. All timing checks also accept an optional argument that is the name of a register (reg) variable to toggle upon timing check failure. Model developers typically connect this variable to a UDP input to make the UDP output go to the unknown state upon failure of an associated timing check.

Appendix Summary

You can use built-in primitives and continuous assignments to precisely model logic:

- Built-in primitives include logic gates and pullup/pulldown and unidirectional switches and bidirectional switches.

You can specify drive strengths for primitives and continuous assignments, and charge strengths for capacitive nets:

- Verilog resolves the value and strength of a signal on a net.

You can specify propagation delays across primitives and nets, and decay time for capacitive nets:

- Generally up to 3 values for rise, fall, and turn-off, and these values can be minimum:typical:maximum triples.

cadence

This module explored the use of built-in primitives and continuous assignments to model detailed combinational behavior down to the switch level. It examined the built-in primitives, how to model primitive drive strength and capacitive net charge strength, and how to model propagation delays across primitives and nets.

Appendix Summary (continued)

You can extend the set of built-in primitives by defining your own:

- You can define combinational and sequential primitives.
- You define their behavior with a table.
- You instantiate them like a built-in primitive.

UDPs accurately, compactly and efficiently describe logic function:

- You can replace a circuit of built-in primitives by a single UDP.

UDPs cannot do some things that a built-in primitive can do:

- A UDP has a single output and at most 10 inputs.
- A UDP port cannot be bi-directional and cannot drive the high-impedance (“z”) value.
- The logic synthesis standard does not accept UDPs.

cadence

This module explored user-defined primitives. It compared UDPs with the built-in Verilog primitives and explained why you would want to define your own UDP and how to define it. The module ended with several complete examples of representative UDPs.

Appendix Review

1. What Verilog element(s) can model capacitance?
2. What Verilog element(s) can model resistance?
3. Why do primitive instances not need instance names?
4. Why do primitive definitions not have port names?
5. What delay does the simulator use for transitions to X if you specify only rise, fall, and turnoff delays?

cadence

This page does not contain notes.

Appendix Review (continued)

6. Suggest some reasons to define your own primitives.
7. If multiple UDP inputs transition in the same simulation cycle, how many times does the simulator consult the look-up table?
8. What does the simulator do if some combination of input transition and states matches no table row?
9. What does the simulator do if some combination of input transition and states matches both a level-sensitive table construct and an edge-sensitive table construct?
10. What does the simulator do, if due to your use of wildcard symbols, some combination of input transition and states matches multiple level-sensitive table constructs, or multiple edge-sensitive table constructs, that specify different outputs?

cadence

This page does not contain notes.

Appendix Review Solutions

1. What Verilog element(s) can model capacitance?
   - Only a trireg net can have charge strength and decay time.
2. What Verilog element(s) can model resistance?
   - Only the rtran primitive models bidirectional uncontrolled resistance.
3. Why do primitive instances not need instance names?
   - Instance names provide a scope for accessing internal elements that primitives do not have.
4. Why do primitive definitions not have port names?
   - The number of primitive terminals is few and their order is fixed, thus accommodating named connections would provide little benefit.
5. What delay does the simulator use for transitions to X if you specify only rise, fall, and turnoff delays?
   - The simulator uses the worst-case (minimum) of the specified delays.
6. Suggest some reasons to define your own primitives.
   - The reasons are:
     - Precisely control output transitions in response to input transitions.
     - Replace a “glitchy” cloud of primitive logic with a few primitives having “clean” output transitions.
     - More efficiently utilize platform resources (cycles and memory).

cadence

This page does not contain notes.

Appendix Review Solutions (continued)

7. If multiple UDP inputs transition in the same simulation cycle, how many times does the simulator consult the look-up table?
   - The simulator consults the look-up table once for each transitioning input.
8. What does the simulator do if some combination of input transition and states matches no table row?
   - The simulator places the UDP output in the unknown (“X”) state.
9. What does the simulator do if some combination of input transition and states matches both a level-sensitive table construct and an edge-sensitive table construct?
   - The simulator processes first the edge-sensitive construct, and then the level-sensitive construct, which overrides.
10. What does the simulator do, if due to your use of wildcard symbols, some combination of input transition and states matches multiple level-sensitive table constructs, or multiple edge-sensitive table constructs, that specify different outputs?
    - This situation is an error that that should be detected during elaboration.

cadence

This page does not contain notes.

Appendix Exercise

- Given this behavioral model of one stage of a binary counter:
  - Convert the model to a structural representation that uses the Verilog built-in primitives. Assume that set and reset cannot both be simultaneously asserted.
  - Second convert the model to a user-defined primitive. Assume that set and reset cannot both be simultaneously asserted.

```
module countBit (
output reg q;
input clk, set_, rst_ );
 always @( negedge clk
        or negedge set_
        or negedge rst_ )
  if (~rst) q <= 0; else
  if (~set) q <= 1; else
            q <= ~q;
endmodule
```


cadence

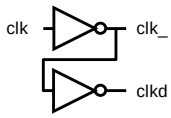
This page does not contain notes.

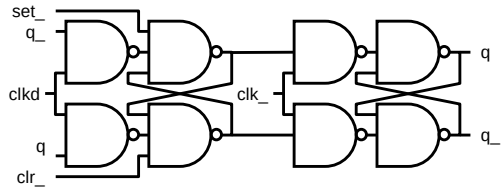
Appendix Exercise Solution

- Given this behavioral model of one stage of a binary counter:
  - Convert the model to a structural representation that uses the Verilog built-in primitives. Assume that set and reset cannot both be simultaneously asserted.
  - Second convert the model to a user-defined primitive. Assume that set and reset cannot both be simultaneously asserted.

```
module countBit (
output reg q;
input clk, set_, rst_ );
 always @( negedge clk
        or negedge set_
        or negedge rst_ )
  if (~rst) q <= 0; else
  if (~set) q <= 1; else
            q <= ~q;
endmodule
```

```
module countBit (
output wire q,
input clk, set_, rst_ );
 not  (clk_, clk), (clkd, clk_);
 nand (n1a,q_,clkd), (n1b,q,clkd);
 nand (n2a,set_,n1a,n2b), (n2b,rst_,n1b,n2a);
 nand (n3a,n2a,clk_), (n3b,n2b,clk_);
 nand (q,set_,n3a,q_), (q_,rst_,n3b,q);
endmodule
```





cadence

This page does not contain notes.

Labs

Lab B-1 Using Built-In Verilog Primitives with a Macro Library

- For this lab, you modify a provided macro library file to replace RTL descriptions with descriptions based upon the Verilog built-in primitives.

Lab B-2 Using User-Defined Verilog Primitives with a Macro Library

- For this lab, you further modify the macro library file to define a sequential flip-flop UDP and replace the primitive-based flip-flops with UDP-based flip-flops.

cadence

Your objective for the first lab is to use built-in primitives and continuous assignments to model logic.

For this lab, you use built-in primitives to define the functionality of a small set of ASIC macrocells.

Your objective for the second lab is to define combinational and sequential primitives.

For this lab, you use user-defined primitives to define the functionality of a small set of ASIC macrocells.