6

Register, Multibit, Multiplexer, and Three-State Inference

HDL Compiler can infer registers (latches and flip-flops), multiplexers, and three-state cells. This chapter explains inference behavior and results, in the following sections:

- Register Inference
- Multibit Inference
- Multiplexer Inference
- Three-State Inference

Register Inference

Register inference allows you to use sequential logic in your designs and keep your designs technology-independent. A register is a simple, 1-bit memory device, either a latch or a flip-flop. A latch is a level-sensitive memory device. A flip-flop is an edge-triggered memory device.

The register inference capability can support coding styles other than those described in this chapter. However, for best results,

- Restrict each `always` block to a single type of memory-element inferencing:
  - Latch
  - Latch with asynchronous set or reset
  - Flip-flop
  - Flip-flop with asynchronous reset
  - Flip-flop with synchronous reset
- Use the templates provided in “Inferring Latches” on page 6-10 and “Inferring Flip-Flops” on page 6-25.

Reporting Register Inference

HDL Compiler provides the following controls for reporting register inference:

- Configuring the inference report
- Selecting the latch inference warnings

The following sections describe these controls.

Configuring the Inference Report

HDL Compiler can generate an inference report that shows the information HDL Compiler passes on to Design Compiler about the inferred devices. Use the following variables to configure an inference report:

`hdlin_report_inferred_modules = true`

This variable controls the generation of the inference report. You can select from the following settings for this variable:

`false`
HDL Compiler does not generate an inference report.

`true`
HDL Compiler generates a general inference report when building a design. This is the default setting. Example 6-1 shows a general inference report for a JK flip-flop.

`verbose`
HDL Compiler generates a verbose inference report when building a design. It provides the asynchronous set or reset, synchronous set or reset, and synchronous toggle conditions of each latch or flip-flop, expressed as Boolean formulas. Example 6-2 shows a verbose inference report for a JK flip-flop.

`hdlin_reg_report_length = 60`

This variable indicates the length of the Boolean formulas reported in the verbose inference report. You must specify an integer value for this variable. The default setting is 60.

Example 6-1 General Inference Report for a JK Flip-Flop

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	Y	Y	N

Example 6-2 Verbose Inference Report for a JK Flip-Flop

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	Y	Y	N

```
Q_reg
Sync-reset: J’ K
Sync-set: J K’
Sync-toggle: J K
Sync-set and Sync-reset ==> Q: X
```

In the inference reports in Example 6-1 and Example 6-2,

- Y indicates that the flip-flop has a synchronous reset (SR) and a synchronous set (SS)
- N indicates that the flip-flop does not have an asynchronous reset (AR), an asynchronous set (AS), or a synchronous toggle (ST)

In the verbose inference report (Example 6-2), the last part of the report lists the objects that control the synchronous reset and set conditions. In this example, a synchronous reset occurs when J is low (logic 0) and K is high (logic 1). The last line of the report indicates the register output value when both set and reset are active:

zero(0)

Indicates that the reset has priority and that the output goes to logic 0.

one (1)

Indicates that the set has priority and that the output goes to logic 1.

X

Indicates that there is no priority and that the output is unstable.

“Inferring Latches” on page 6-10 and “Inferring Flip-Flops” on page 6-25 provide inference reports for each register template. After you input a Verilog description, check the inference report to verify that HDL Compiler passes the correct information to Design Compiler.

Selecting Latch Inference Warnings

Use the `hdlin_check_no_latch` variable to control whether HDL Compiler generates warning messages when inferring latches.

If `hdlin_check_no_latch` is set true, HDL Compiler generates a warning message when it infers a latch. This is useful for verifying that a combinational design does not contain memory components. The default setting of the `hdlin_check_no_latch` variable is false.

Controlling Register Inference

Use HDL Compiler directives or dc_shell variables to direct HDL Compiler to the type of sequential device you want inferred. HDL Compiler directives give you control over individual signals, and dc_shell variables apply to an entire design.

Attributes That Control Register Inference

HDL Compiler provides the following directives for controlling register inference:

`async_set_reset`

When a signal has this directive set to true, HDL Compiler searches for a branch that uses the signal as a condition. HDL Compiler then checks whether the branch contains an assignment to a constant value. If the branch does, the signal becomes an asynchronous reset or set.

Attach this directive to single-bit signals, using the following syntax:

```
// synopsys async_set_reset ”signal_name_list”
```

`async_set_reset_local`

HDL Compiler treats listed signals in the specified block as if they have the `async_set_reset` directive set to true.

Attach this directive to a block label, using the following syntax:

```
/* synopsys async_set_reset_local block_label
   ”signal_name_list” */
```

`async_set_reset_local_all`

HDL Compiler treats all signals in the specified blocks as if they have the `async_set_reset` directive set to true.

Attach this directive to block labels, using the following syntax:

```
/* synopsys async_set_reset_local_all
   ”block_label_list” */
```

sync_set_reset

When a signal has this directive set to true, HDL Compiler checks the signal to determine whether it synchronously sets or resets a register in the design.

Attach this directive to single-bit signals, using the following syntax:

```
//synopsys sync_set_reset ”signal_name_list”
```

sync_set_reset_local

HDL Compiler treats listed signals, in the specified block as if they have the sync_set_reset directive set to true.

Attach this directive to a block label, using the following syntax:

```
/* synopsys sync_set_reset_local block_label
   ”signal_name_list” */
```

sync_set_reset_local_all

HDL Compiler treats all signals in the specified blocks as if they have the sync_set_reset directive set to true.

Attach this directive to block labels, using the following syntax:

```
/* synopsys sync_set_reset_local_all
   ”block_label_list” */
```

one_cold

A one-cold implementation means that all signals in a group are active-low and that only one signal can be active at a given time. The one_cold directive prevents Design Compiler from implementing priority encoding logic for the set and reset signals.

Add a check to the Verilog code to ensure that the group of signals has a one-cold implementation. HDL Compiler does not produce any logic to check this assertion.

Attach this directive to set or reset signals on sequential devices, using the following syntax:

```
// synopsys one_cold "signal_name_list"
```

`one_hot`

A one-hot implementation means that all signals in a group are active-high and that only one signal can be active at a given time. The `one_hot` directive prevents Design Compiler from implementing priority encoding logic for the set and reset signals.

Add a check to the Verilog code to ensure that the group of signals has a one-hot implementation. HDL Compiler does not produce any logic to check this assertion.

Attach this directive to set or reset signals on sequential devices, using the following syntax:

```
// synopsys one_hot "signal_name_list"
```

Variables That Control Register Inference

You can use the following dc_shell variables to control register inference:

`hdlin_ff_always_async_set_reset = true`

When this variable is true, HDL Compiler automatically checks for asynchronous set and reset conditions of flip-flops.

`hdlin_ff_always_sync_set_reset = false`

When this variable is true, HDL Compiler automatically checks for synchronous set and reset conditions of flip-flops.

`hdlin_latch_always_async_set_reset = false`

When this variable is true, HDL Compiler automatically checks for asynchronous set and reset conditions of latches. When this variable is false, HDL Compiler interprets each control object of a latch as synchronous.

Setting the variable to true is equivalent to specifying every object in the design in the object list for the `async_set_reset` directive. When true for a design subsequently analyzed, every constant 0 loaded on a latch is used for asynchronous reset and every constant 1 loaded on a latch is used for asynchronous set. HDL Compiler does not limit checks for assignments to a constant 0 or constant 1 to a single block. That is, HDL Compiler performs checking across blocks.

`hdlin_keep_feedback = false`

When this variable is false, HDL Compiler removes all flip-flop feedback loops. For example, HDL Compiler removes feedback loops inferred from a statement such as `Q=Q`. Removing the state feedback from a simple D flip-flop creates a synchronous loaded flip-flop. Set this variable to true if you want to keep feedback loops.

`hdlin_keep_inv_feedback = true`

When this variable is false, HDL Compiler removes all inverted flip-flop feedback loops. For example, HDL Compiler removes feedback loops inferred from a statement such as `Q=Q`. Removing the inverted feedback from a simple D flip-flop creates a toggle flip-flop. Set this variable to true if you want to keep feedback loops.

Inferring Latches

In simulation, a signal or variable holds its value until that output is reassigned. In hardware, a latch implements this holding-of-state capability. HDL Compiler supports inference of the following types of latches:

- SR latch
- D latch
- Master-slave latch

The following sections provide details about each of these latch types.

Inferring SR Latches

Use SR latches with caution, because they are difficult to test. If you decide to use SR latches, verify that the inputs are hazard-free (that they do not glitch). During synthesis, Design Compiler does not ensure that the logic driving the inputs is hazard-free.

Example 6-3 shows the Verilog code that implements the inferred SR latch shown in Figure 6-1 on page 6-12 and described in Table 6-1 on page 6-11. Because the output y is unstable when both inputs have a logic 0 value, you might want to include a check in the Verilog code to detect this condition during simulation. Synthesis does not support such checks, so you must put the `translate_off` and `translate_on` directives around the check. See “translate_off and translate_on Directives” on page 9-6 for more information about special comments in the Verilog source code.

Example 6-4 shows the inference report HDL Compiler generates.

Table 6-1 SR Latch Truth Table (NAND Type)

set	reset	y
0	0	Not stable
0	1	1
1	0	0
1	1	y

Example 6-3 SR Latch

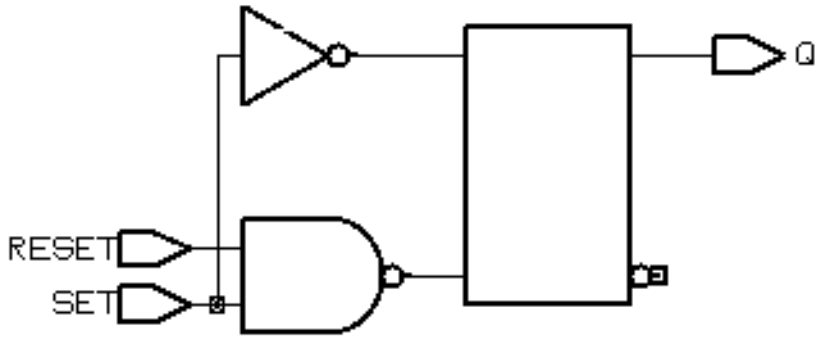
```
module sr_latch (SET, RESET, Q);
  input SET, RESET;
  output Q;
  reg Q;

//synopsys async_set_reset ”SET, RESET”
always @(RESET or SET)
  if (~RESET)
    Q = 0;
  else if (~SET)
    Q = 1;
endmodule
```

Example 6-4 Inference Report for an SR Latch

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	Y	Y	-	-	-

```
Q_reg
Async-reset: RESET’
Async-set: SET’
Async-set and Async-reset ==> Q: 1
```

Figure 6-1 SR Latch



Inferring D Latches

When you do not specify the resulting value for an output under all conditions, as in an incompletely specified if or case statement, HDL Compiler infers a D latch.

For example, the `if` statement in Example 6-5 infers a D latch because there is no `else` clause. The Verilog code specifies a value for output `Q` only when input enable has a logic 1 value. As a result, output `Q` becomes a latched value.

Example 6-5 Latch Inference Using an if Statement

```
always @ (DATA or GATE) begin
  if (GATE) begin
    Q = DATA;
  end
end
```

The `case` statement in Example 6-6 infers D latches, because the `case` statement does not provide assignments to decimal for values of I between 10 and 15.

Example 6-6 Latch Inference Using a case Statement

```
always @(I) begin
  case(I)
    4'h0: decimal= 10'b0000000001;
    4'h1: decimal= 10'b0000000010;
    4'h2: decimal= 10'b0000000100;
    4'h3: decimal= 10'b0000001000;
    4'h4: decimal= 10'b0000010000;
    4'h5: decimal= 10'b0000100000;
    4'h6: decimal= 10'b0001000000;
    4'h7: decimal= 10'b0010000000;
    4'h8: decimal= 10'b0100000000;
    4'h9: decimal= 10'b1000000000;
  endcase
end
```

To avoid latch inference, assign a value to the signal under all conditions. To avoid latch inference by the `if` statement in Example 6-5, modify the block as shown in Example 6-7 or Example 6-8. To avoid latch inference by the `case` statement in Example 6-6, add the following statement before the `endcase` statement:

```
default: decimal= 10'b0000000000;
```

Example 6-7 Avoiding Latch Inference

```
always @ (DATA, GATE) begin
  Q = 0;
  if (GATE)
    Q = DATA;
end
```

Example 6-8 Another Way to Avoid Latch Inference

```
always @ (DATA, GATE) begin
  if (GATE)
    Q = DATA;
  else
    Q = 0;
end
```

Variables declared locally within a subprogram do not hold their value over time, because every time a subprogram is called, its variables are reinitialized. Therefore, HDL Compiler does not infer latches for variables declared in subprograms. In Example 6-9, HDL Compiler does not infer a latch for output `Q`.

Example 6-9 Function: No Latch Inference

```
function MY_FUNC
  input DATA, GATE;
  reg STATE;

  begin
    if (GATE) begin
      STATE = DATA;
    end
    MY_FUNC = STATE;
  end
end function
. . .
Q = MY_FUNC(DATA, GATE);
```

The following sections provide truth tables, code examples, and figures for these types of D latches:

- Simple D Latch
- D Latch With Asynchronous Set or Reset
- D Latch With Asynchronous Set and Reset

Simple D Latch

When you infer a D latch, make sure you can control the gate and data signals from the top-level design ports or through combinational logic. Controllable gate and data signals ensure that simulation can initialize the design.

Example 6-10 provides the Verilog template for a D latch. HDL Compiler generates the verbose inference report shown in Example 6-11. Figure 6-2 shows the inferred latch.

Example 6-10 D Latch

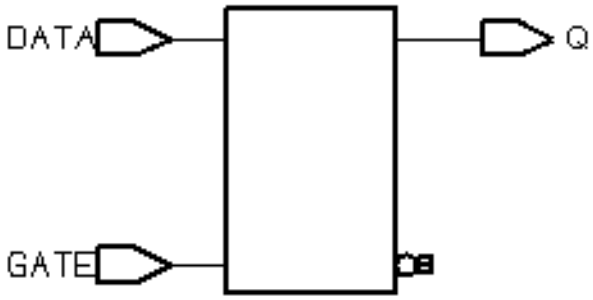
```
module d_latch (GATE, DATA, Q);
  input GATE, DATA;
  output Q;
  reg Q;

always @(GATE or DATA)
  if (GATE)
    Q = DATA;

endmodule
```

Example 6-11 Inference Report for a D Latch

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	N	N	-	-	-

```
Q_reg
reset/set: none
```

Figure 6-2 D Latch



D Latch With Asynchronous Set or Reset

The templates in this section use the `async_set_reset` directive to direct HDL Compiler to the asynchronous set or reset pins of the inferred latch.

Example 6-12 provides the Verilog template for a D latch with an asynchronous set. HDL Compiler generates the verbose inference report shown in Example 6-13. Figure 6-3 shows the inferred latch.

Example 6-12 D Latch With Asynchronous Set

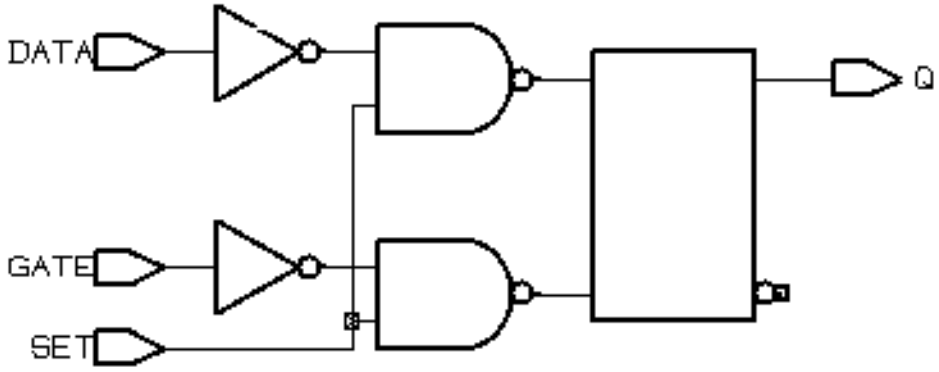
```
module d_latch_async_set (GATE, DATA, SET, Q);
  input GATE, DATA, SET;
  output Q;
  reg Q;

//synopsys async_set_reset ”SET”
always @(GATE or DATA or SET)
  if (~SET)
    Q = 1’b1;
  else if (GATE)
    Q = DATA;
endmodule
```

Example 6-13 Inference Report for D Latch With Asynchronous Set

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	N	Y	-	-	-

```
Q_reg
Async-set: SET'
```

Figure 6-3 D Latch With Asynchronous Set



Note:

Because the target technology library does not contain a latch with an asynchronous set, Design Compiler synthesizes the set logic by using combinational logic.

Example 6-14 provides the Verilog template for a D latch with an asynchronous reset. HDL Compiler generates the verbose inference report shown in Example 6-15. Figure 6-4 shows the inferred latch.

Example 6-14 D Latch With Asynchronous Reset

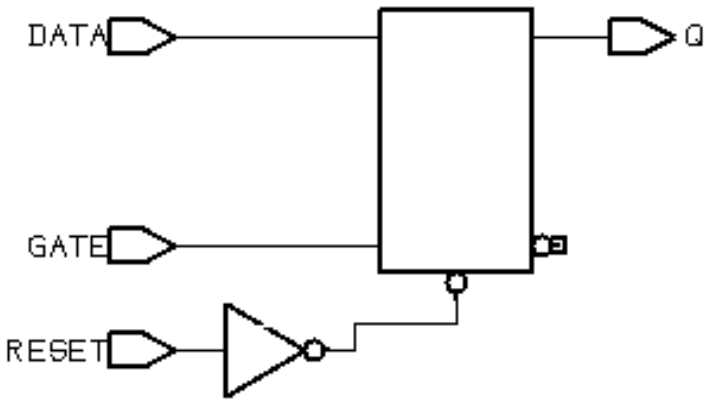
```
module d_latch_async_reset (RESET, GATE, DATA, Q);
  input RESET, GATE, DATA;
  output Q;
  reg Q;

//synopsys async_set_reset ”RESET”
always @ (RESET or GATE or DATA)
  if (~RESET)
    Q = 1’b0;
  else if (GATE)
    Q = DATA;
endmodule
```

Example 6-15 Inference Report for D Latch With Asynchronous Set

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	Y	N	-	-	-

```
Q_reg
Async-reset: RESET’
```

Figure 6-4 D Latch With Asynchronous Reset



D Latch With Asynchronous Set and Reset

Example 6-16 provides the Verilog template for a D latch with an active-low asynchronous set and reset. This template uses the `async_set_reset_local` directive to direct HDL Compiler to the asynchronous signals in block infer. This template uses the `one_cold` directive to prevent priority encoding of the set and reset signals. For this template, if you do not specify the `one_cold` directive, the set signal has priority, because it serves as the condition for the if clause. HDL Compiler generates the verbose inference report shown in Example 6-17. Figure 6-4 shows the inferred latch.

Example 6-16 D Latch With Asynchronous Set and Reset

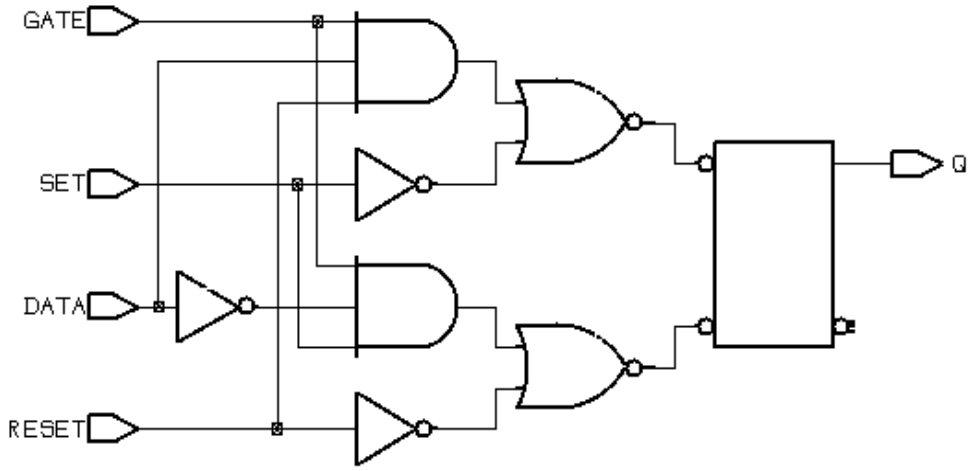
```
module d_latch_async (GATE, DATA, RESET, SET, Q);
  input GATE, DATA, RESET, SET;
  output Q;
  reg Q;

// synopsys async_set_reset_local infer ”RESET, SET”
// synopsys one_cold ”RESET, SET”
always @ (GATE or DATA or RESET or SET)
begin : infer
  if (!SET)
    Q = 1’b1;
  else if (!RESET)
   Q = 1’b0;
   else if (GATE)
    Q = DATA;
end

// synopsys translate_off
always @ (RESET or SET)
  if (RESET == 1’b0 & SET == 1’b0)
  $write (”ONE-COLD violation for RESET and SET.”);
// synopsys translate_on
endmodule
```

Example 6-17 Inference Report for D Latch With Asynchronous Set and Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Latch	1	-	-	Y	Y	-	-	-

```
Q_reg
Async-reset: RESET'
Async-set: SET'
Async-set and Async-reset ==> Q: X
```

Figure 6-5 D Latch With Asynchronous Set and Reset



Inferring Master-Slave Latches

HDL Compiler infers master-slave latches by using the `clocked_on_also` `signal_type` attribute.

In your Verilog description, describe the master-slave latch as a flip-flop by using only the slave clock. Specify the master clock as an input port, but do not connect it. In addition, attach the `clocked_on_also` attribute to the master clock port (called `MCK` in these examples).

This coding style requires that cells in the target technology library have slave clocks defined in the library with the `clocked_on_also` attribute in the cell's state declaration. (For more information, see the Synopsys Library Compiler documentation.)

If Design Compiler does not find any master-slave latches in the target technology library, the tool leaves the master-slave generic cell (`MSGEN`) unmapped. Design Compiler does not use D flip-flops to implement the equivalent functionality of the cell.

Note:

Although the vendor's component behaves as a master-slave latch, Library Compiler supports only the description of a master-slave flip-flop.

Master-Slave Latch With Single Master-Slave Clock Pair

Example 6-19 provides the Verilog template for a master-slave latch. The template uses the `dc_script_begin` and `dc_script_end` compiler directives. See "Embedding Constraints and Attributes" on page 9-22 for more information. HDL Compiler generates the verbose inference report shown in Example 6-20. Figure 6-6 shows the inferred latch.

Example 6-18 Master-Slave Latch

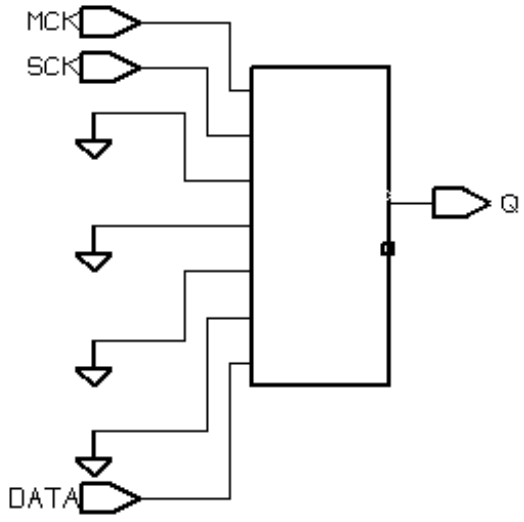
```
module mslatch (SCK, MCK, DATA, Q);
  input SCK, MCK, DATA;
  output Q;
  reg Q;

// synopsys dc_script_begin
// set_signal_type ”clocked_on_also” MCK
// synopsys dc_script_end

always @ (posedge SCK)
  Q <= DATA;
endmodule
```

Example 6-19 Inference Report for a Master-Slave Latch

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	N	N

```
Q_reg
set/reset/toggle: none
```

Figure 6-6 Master-Slave Latch



Master-Slave Latch With Multiple Master-Slave Clock Pairs

If the design requires more than one master-slave clock pair, you must specify the associated slave clock in addition to the `clocked_on_also` attribute. Example 6-21 illustrates the use of the `clocked_on_also` attribute with the `-associated_clock` option.

Example 6-20 Inferring Master-Slave Latches With Two Pairs of Clocks

```
module mslatch2 (SCK1, SCK2, MCK1, MCK2, D1, D2, Q1, Q2);
  input SCK1, SCK2, MCK1, MCK2, D1, D2;
  output Q1, Q2;
  reg Q1, Q2;

// synopsys dc_script_begin
// set_signal_type ”clocked_on_also” MCK1 -associated_clock SCK1
// set_signal_type ”clocked_on_also” MCK2 -associated_clock SCK2
// synopsys dc_script_end

always @ (posedge SCK1)
  Q1 <= D1;

always @ (posedge SCK2)
  Q2 <= D2;
endmodule
```

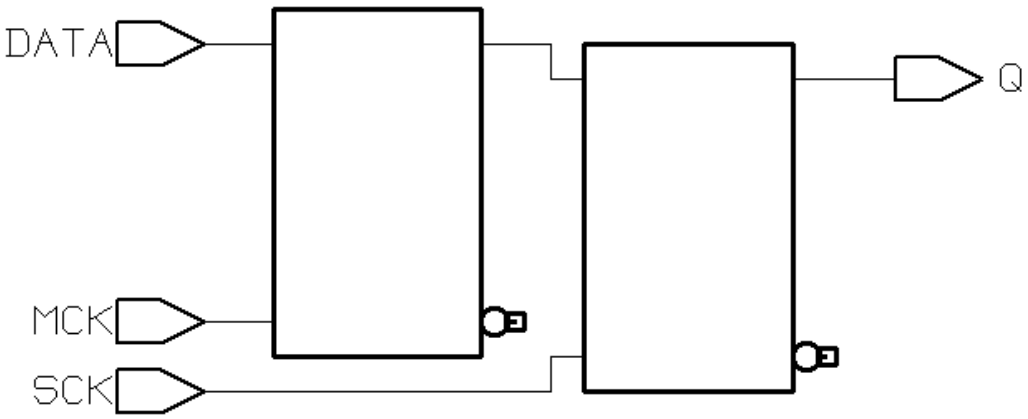
Master-Slave Latch With Discrete Components

If your target technology library does not contain master-slave latch components, you can infer two-phase systems by using D latches. Example 6-22 shows a simple two-phase system with clocks MCK and SCK. Figure 6-7 shows the inferred latch.

Example 6-21 Two-Phase Clocks

```
module latch_verilog (DATA, MCK, SCK, Q);
  input DATA, MCK, SCK;
  output Q;
  reg Q;

  reg TEMP;

always @(DATA or MCK)
  if (MCK)
    TEMP <= DATA;

always @(TEMP or SCK)
  if (SCK)
    Q <= TEMP;
endmodule
```

Figure 6-7 Two-Phase Clocks



Inferring Flip-Flops

HDL Compiler can infer D flip-flops, JK flip-flops, and toggle flip-flops. The following sections provide details about each of these flip-flop types.

Inferring D Flip-Flops

HDL Compiler infers a D flip-flop whenever the sensitivity list of an `always` block includes an `edge` expression (a test for the rising or falling edge of a signal). Use the following syntax to describe a rising edge:

```
posedge SIGNAL
```

Use the following syntax to describe a falling edge:

```
negedge SIGNAL
```

When the sensitivity list of an `always` block contains an `edge` expression, HDL Compiler creates flip-flops for all the variables that are assigned values in the block. Example 6-22 shows the most common use of an `always` block to infer a flip-flop.

Example 6-22 Using an always Block to Infer a Flip-Flop

```
always @(edge)
begin
  .
end
```

Simple D Flip-Flop

When you infer a D flip-flop, make sure you can control the clock and data signals from the top-level design ports or through combinational logic. Controllable clock and data signals ensure that simulation can initialize the design. If you cannot control the clock and data signals, infer a D flip-flop with an asynchronous reset or set or with a synchronous reset or set.

When you are inferring a simple D flip-flop, the `always` block can contain only one edge expression.

Example 6-23 provides the Verilog template for a positive-edge-triggered D flip-flop. HDL Compiler generates the verbose inference report shown in Example 6-24. Figure 6-8 shows the inferred flip-flop.

Example 6-23 Positive-Edge-Triggered D Flip-Flop

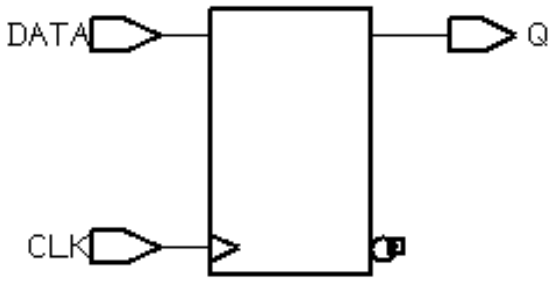
```
module dff_pos (DATA, CLK, Q);
  input DATA, CLK;
  output Q;
  reg Q;

always @(posedge CLK)
  Q <= DATA;
endmodule
```

Example 6-24 Inference Report for a Positive-Edge-Triggered D Flip-Flop

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	N	N

```
Q_reg
set/reset/toggle: none
```

Figure 6-8 Positive-Edge-Triggered D Flip-Flop



Example 6-25 provides the Verilog template for a negative-edge-triggered D flip-flop. HDL Compiler generates the verbose inference report shown in Example 6-26. Figure 6-9 shows the inferred flip-flop.

Example 6-25 Negative-Edge-Triggered D Flip-Flop

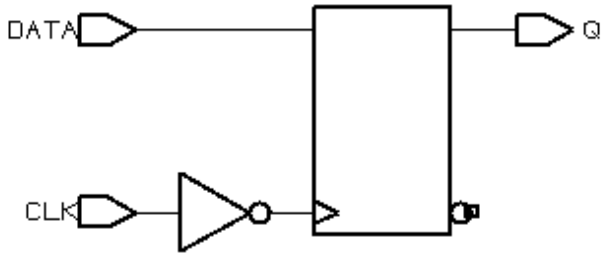
```
module dff_neg (DATA, CLK, Q);
  input DATA, CLK;
  output Q;
  reg Q;

always @(negedge CLK)
  Q <= DATA;
endmodule
```

Example 6-26 Inference Report for a Negative-Edge-Triggered D Flip-Flop

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	N	N

```
Q_reg
set/reset/toggle: none
```

Figure 6-9 Negative-Edge-Triggered D Flip-Flop



D Flip-Flop With Asynchronous Set or Reset

When inferring a D flip-flop with an asynchronous set or reset, include edge expressions for the clock and the asynchronous signals in the sensitivity list of the `always` block. Specify the asynchronous conditions by using `if` statements. Specify the branches for the asynchronous conditions before the branches for the synchronous conditions.

Example 6-27 provides the Verilog template for a D flip-flop with an asynchronous set. HDL Compiler generates the verbose inference report shown in Example 6-28. Figure 6-10 shows the inferred flip-flop.

Example 6-27 D Flip-Flop With Asynchronous Set

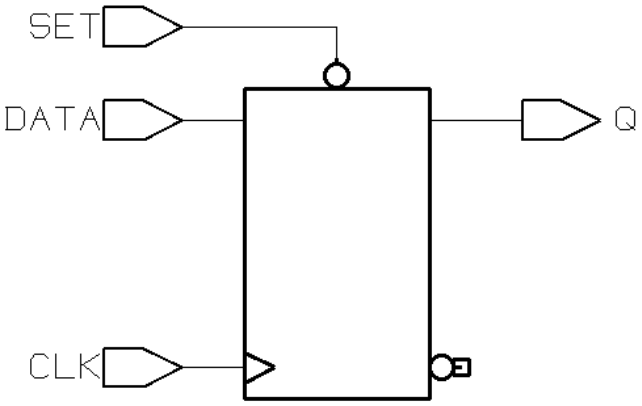
```
module dff_async_set (DATA, CLK, SET, Q);
  input DATA, CLK, SET;
  output Q;
  reg Q;

always @(posedge CLK or negedge SET)
  if (~SET)
    Q <= 1'b1;
  else
    Q <= DATA;
endmodule
```

Example 6-28 Inference Report for a D Flip-Flop With Asynchronous Set

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	Y	N	N	N

```
Q_reg
Async-set: SET'
```

Figure 6-10 D Flip-Flop With Asynchronous Set



Example 6-29 provides the Verilog template for a D flip-flop with an asynchronous reset. HDL Compiler generates the verbose inference report shown in Example 6-30. Figure 6-11 shows the inferred flip-flop.

Example 6-29 D Flip-Flop With Asynchronous Reset

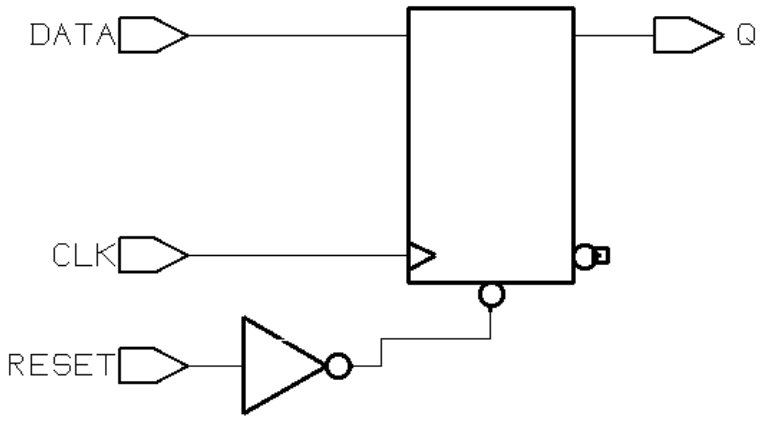
```
module dff_async_reset (DATA, CLK, RESET, Q);
  input DATA, CLK, RESET;
  output Q;
  reg Q;

always @(posedge CLK or posedge RESET)
  if (RESET)
    Q <= 1'b0;
  else
    Q <= DATA;
endmodule
```

Example 6-30 Inference Report for a D Flip-Flop With Asynchronous Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	Y	N	N	N	N

```
Q_reg
Async-reset: RESET
```

Figure 6-11 D Flip-Flop With Asynchronous Reset



D Flip-Flop With Asynchronous Set and Reset

Example 6-31 provides the Verilog template for a D flip-flop with active high asynchronous set and reset pins. The template uses the `one_hot` directive to prevent priority encoding of the set and reset signals. For this template, if you do not specify the `one_hot` directive, the reset signal has priority, because it is used as the condition for the if clause. HDL Compiler generates the verbose inference report shown in Example 6-32. Figure 6-12 shows the inferred flip-flop.

Example 6-31 D Flip-Flop With Asynchronous Set and Reset

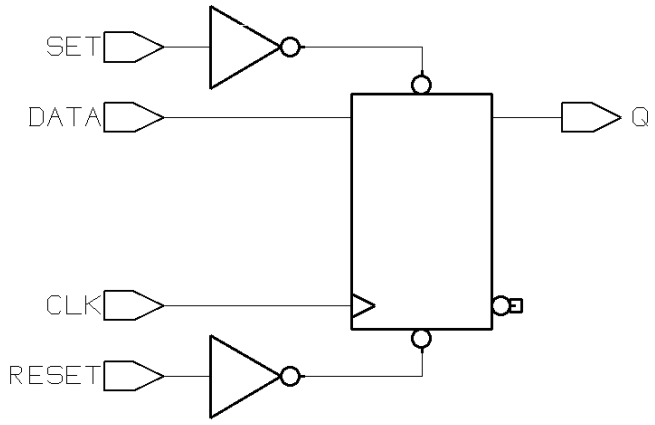
```
module dff_async (RESET, SET, DATA, Q, CLK);
  input CLK;
  input RESET, SET, DATA;
  output Q;
  reg Q;

// synopsys one_hot ”RESET, SET”
always @(posedge CLK or posedge RESET or
         posedge SET)
  if (RESET)
    Q <= 1’b0;
  else if (SET)
    Q <= 1’b1;
  else Q <= DATA;

// synopsys translate_off
always @ (RESET or SET)
  if (RESET + SET > 1)
  $write (”ONE-HOT violation for RESET and SET.”);
// synopsys translate_on
endmodule
```

Example 6-32 Inference Report for a D Flip-Flop With Asynchronous Set and Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	Y	Y	N	N	N

```
Q_reg
Async-reset: RESET
Async-set: SET
Async-set and Async-reset ==> Q: X
```

Figure 6-12 D Flip-Flop With Asynchronous Set and Reset



D Flip-Flop With Synchronous Set or Reset

The previous examples illustrate how to infer a D flip-flop with asynchronous controls—one way to initialize or control the state of a sequential device. You can also synchronously reset or set a flip-flop (see Example 6-33 and Example 6-35). The `sync_set_reset` directive directs HDL Compiler to the synchronous controls of the sequential device.

When the target technology library does not have a D flip-flop with synchronous reset, HDL Compiler infers a D flip-flop with synchronous reset logic as the input to the D pin of the flip-flop. If the reset (or set) logic is not directly in front of the D pin of the flip-flop, initialization problems can occur during gate-level simulation of the design.

Example 6-33 provides the Verilog template for a D flip-flop with synchronous set. HDL Compiler generates the verbose inference report shown in Example 6-34. Figure 6-13 shows the inferred flip-flop.

Example 6-33 D Flip-Flop With Synchronous Set

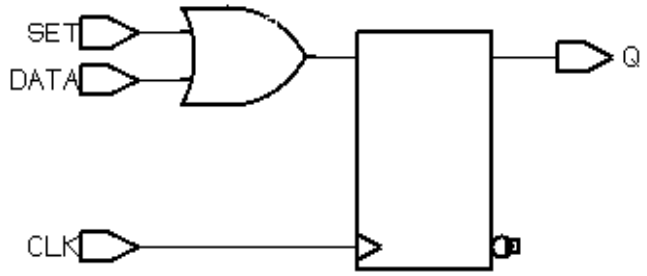
```
module dff_sync_set (DATA, CLK, SET, Q);
  input DATA, CLK, SET;
  output Q;
  reg Q;

//synopsys sync_set_reset ”SET”
always @(posedge CLK)
  if (SET)
    Q <= 1’b1;
  else
    Q <= DATA;
endmodule
```

Example 6-34 Inference Report for a D Flip-Flop With Synchronous Set

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	Y	N

```
Q_reg
Sync-set: SET
```

Figure 6-13 D Flip-Flop With Synchronous Set



Example 6-35 provides the Verilog template for a D flip-flop with synchronous reset. HDL Compiler generates the verbose inference report shown in Example 6-36. Figure 6-14 shows the inferred flip-flop.

Example 6-35 D Flip-Flop With Synchronous Reset

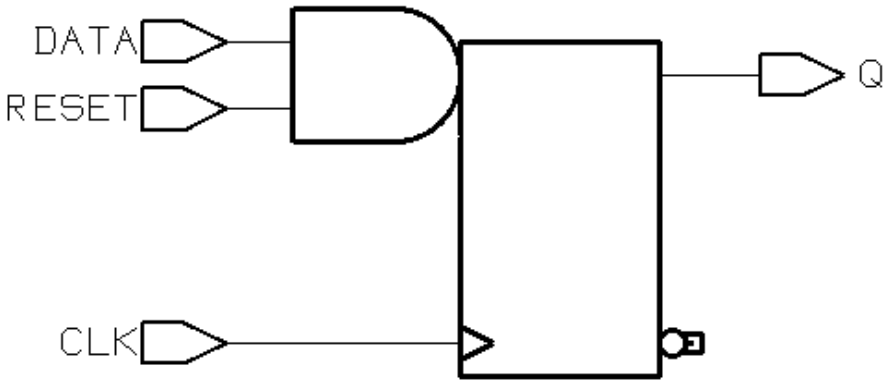
```
module dff_sync_reset (DATA, CLK, RESET, Q);
  input DATA, CLK, RESET;
  output Q;
  reg Q;

//synopsys sync_set_reset ”RESET”
always @(posedge CLK)
  if (~RESET)
    Q <= 1’b0;
  else
    Q <= DATA;
endmodule
```

Example 6-36 Inference Report for a D Flip-Flop With Synchronous Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	Y	N	N

```
Q_reg
Sync-reset: RESET'
```

Figure 6-14 D Flip-Flop With Synchronous Reset



D Flip-Flop With Synchronous and Asynchronous Load

D flip-flops can have asynchronous or synchronous controls. To infer a component with synchronous as well as asynchronous controls, you must check the asynchronous conditions before you check the synchronous conditions.

Example 6-37 provides the Verilog template for a D flip-flop with a synchronous load (called SLOAD) and an asynchronous load (called ALOAD). HDL Compiler generates the verbose inference report shown in Example 6-38. Figure 6-15 shows the inferred flip-flop.

Example 6-37 D Flip-Flop With Synchronous and Asynchronous Load

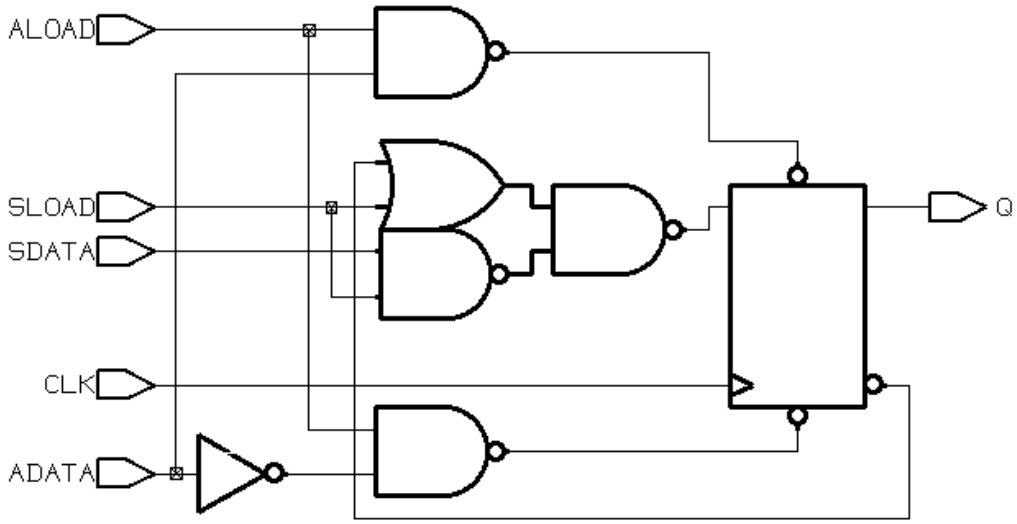
```
module dff_a_s_load (ALOAD, SLOAD, ADATA, SDATA, CLK, Q);
  input ALOAD, ADATA, SLOAD, SDATA, CLK;
  output Q;
  reg Q;

always @ (posedge CLK or posedge ALOAD)
  if (ALOAD)
    Q <= ADATA;
  else if (SLOAD)
    Q <= SDATA;
endmodule
```

Example 6-38 Inference Report for a D Flip-Flop With Synchronous and Asynchronous Load

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	N	N	N

```
Q_reg
set/reset/toggle: none
```

Figure 6-15 D Flip-Flop With Synchronous and Asynchronous Load



Multiple Flip-Flops With Asynchronous and Synchronous Controls

If a signal is synchronous in one block but asynchronous in another block, use the `sync_set_reset_local` and `async_set_reset_local` directives to direct HDL Compiler to the correct implementation.

In Example 6-39, block `infer_sync` uses the reset signal as a synchronous reset and block `infer_async` uses the reset signal as an asynchronous reset. HDL Compiler generates the verbose inference reports shown in Example 6-40. Figure 6-16 shows the resulting design.

Example 6-39 Multiple Flip-Flops With Asynchronous and Synchronous Controls

```
module multi_attr (DATA1, DATA2, CLK, RESET, SLOAD,
                   Q1, Q2);
  input DATA1, DATA2, CLK, RESET, SLOAD;
  output Q1, Q2;
  reg Q1, Q2;

//synopsys sync_set_reset_local infer_sync ”RESET”
always @(posedge CLK)
begin : infer_sync
  if (~RESET)
    Q1 <= 1’b0;
  else if (SLOAD)
    Q1 <= DATA1;   // note: else hold Q
end

//synopsys async_set_reset_local infer_async ”RESET”
always @(posedge CLK or negedge RESET)
begin: infer_async
  if (~RESET)
    Q2 <= 1’b0;
  else if (SLOAD)
    Q2 <= DATA2;
end
endmodule
```

Example 6-40 Inference Reports for Example 6-39

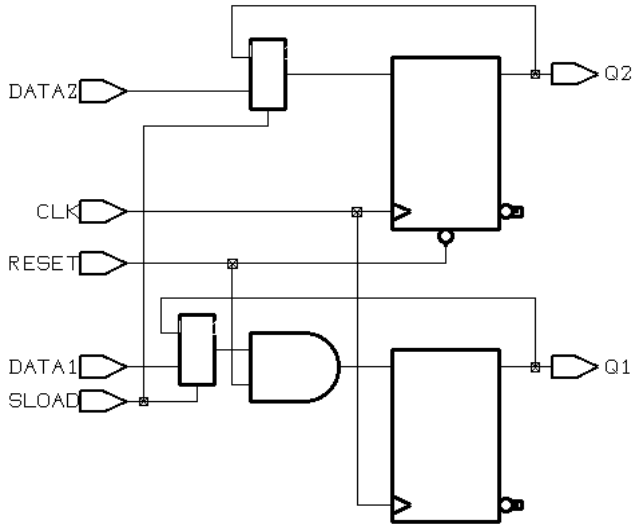
Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q1_reg	Flip-flop	1	-	-	N	N	Y	N	N

```
Q1_reg
Sync-reset: RESET'
```

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q2_reg	Flip-flop	1	-	-	Y	N	N	N	N

```
Q2_reg
Async-reset: RESET'
```

Figure 6-16 Multiple Flip-Flops With Asynchronous and Synchronous Controls



Understanding the Limitations of D Flip-Flop Inference

If you use an `if` statement to infer D flip-flops, your design must meet the following requirements:

- Set and reset conditions cannot use complex expressions.

  The following reset condition is invalid, because it uses a complex expression:

  ```
  always @(posedge clk and negedge reset)
    if (reset == (1-1))
    .
  end
  ```

  HDL Compiler generates the VE-92 message when you use a complex expression in a set or reset condition.

- An `if` statement must occur at the top level of the `always` block.

  The following example is invalid, because the `if` statement does not occur at the top level:

  ```
  always @(posedge clk or posedge reset) begin
    #1;
    if (reset)
    .
  end
  ```

  HDL Compiler generates the following message when the `if` statement does not occur at the top level:

  ```
  Error: The statements in this 'always' block are outside
  the scope of the synthesis policy (%s). Only an 'if'
  statement is allowed at the top level in this 'always'
  block. Please refer to the HDL Compiler reference manual
  for ways to infer flip-flops and latches from 'always'
  blocks. (VE-93)
  ```

Inferring JK Flip-Flops

Use the `case` statement to infer JK flip-flops. Before reading in the Verilog description, set the `hdlin_keep_inv_feedback` variable to false.

Note:

If your inference report does not show a synchronous toggle (ST), check that you have set this variable correctly.

This section describes JK flip-flops and JK flip-flops with asynchronous set and reset.

JK Flip-Flop

When you infer a JK flip-flop, make sure you can control the J, K, and clock signals from the top-level design ports to ensure that simulation can initialize the design.

Example 6-41 provides the Verilog code that implements the JK flip-flop described in Table 6-2. In the JK flip-flop, the J and K signals act as active-high synchronous set and reset. Use the `sync_set_reset` directive to indicate that the J and K signals are the synchronous set and reset for the design. Example 6-42 shows the verbose inference report generated by HDL Compiler. Figure 6-17 shows the inferred flip-flop.

Table 6-2 Truth Table for JK Flip-Flop

J	K	CLK	Qn+1
0	0	Rising	Qn
0	1	Rising	0
1	0	Rising	1
1	1	Rising	QnB
X	X	Falling	Qn

Example 6-41 JK Flip-Flop

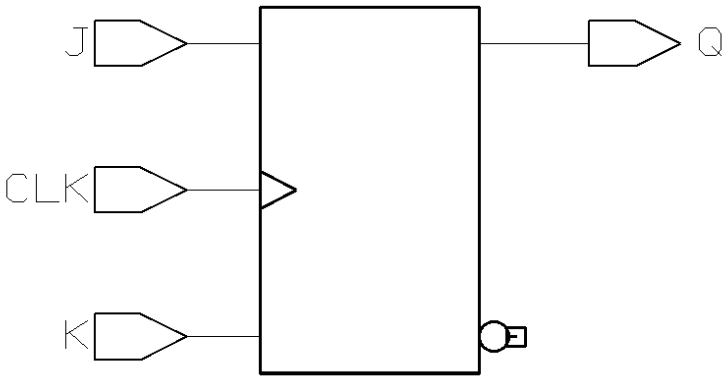
```
module JK(J, K, CLK, Q);
  input J, K;
  input CLK;
  output Q;
  reg Q;

// synopsys sync_set_reset ”J, K”
always @ (posedge CLK)
  case ({J, K})
    2’b01 : Q = 0;
    2’b10 : Q = 1;
    2’b11 : Q = ~Q;
  endcase
endmodule
```

Example 6-42 Inference Report for JK Flip-Flop

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	N	N	Y	Y	Y

```
Q_reg
Sync-reset: J' K
Sync-set: J K'
Sync-toggle: J K
Sync-set and Sync-reset ==> Q: X
```

Figure 6-17 JK Flip-Flop



JK Flip-Flop With Asynchronous Set and Reset

Example 6-43 provides the Verilog template for a JK flip-flop with asynchronous set and reset. Use the `sync_set_reset` directive to indicate the JK function. Use the `one_hot` directive to prevent priority encoding of the J and K signals. HDL Compiler generates the verbose inference report shown in Example 6-44. Figure 6-18 shows the inferred flip-flop.

Example 6-43 JK Flip-Flop With Asynchronous Set and Reset

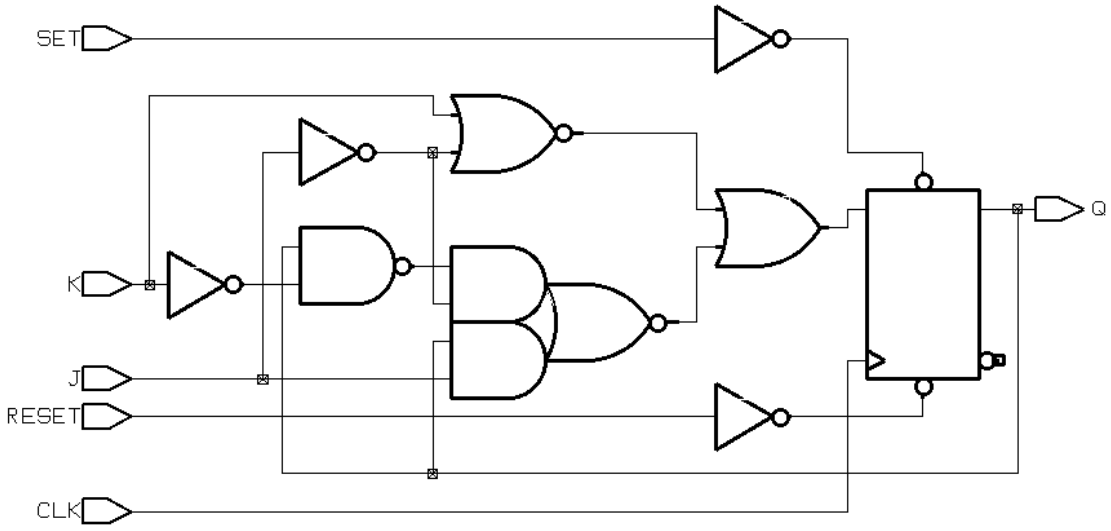
```
module jk_async_sr (RESET, SET, J, K, CLK, Q);
  input RESET, SET, J, K, CLK;
  output Q;
  reg Q;

// synopsys sync_set_reset ”J, K”
// synopsys one_hot ”RESET, SET”
always @ (posedge CLK or posedge RESET or
          posedge SET)
  if (RESET)
    Q <=1’b0;
  else if (SET)
    Q <=1’b1;
  else
    case ({J, K})
      2’b01 : Q = 0;
      2’b10 : Q = 1;
      2’b11 : Q = ~Q;
    endcase

//synopsys translate_off
always @(RESET or SET)
  if (RESET + SET > 1)
    $write (”ONE-HOT violation for RESET and SET.”);
// synopsys translate_on
endmodule
```

Example 6-44 Inference Report for JK Flip-Flop With Asynchronous Set and Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
Q_reg	Flip-flop	1	-	-	Y	Y	Y	Y	Y

```
Q_reg
Async-reset: RESET
Async-set: SET
Sync-reset: J' K
Sync-set: J K'
Sync-toggle: J K
Async-set and Async-reset ==> Q: X
Sync-set and Sync-reset ==> Q: X
```

Figure 6-18 JK Flip-Flop With Asynchronous Set and Reset



Inferring Toggle Flip-Flops

To infer toggle flip-flops, follow the coding style in the following examples and set the `hdlin_keep_inv_feedback` variable to false.

Note:

If your inference report does not show a synchronous toggle (ST), check that you have set this variable correctly.

You must include asynchronous controls in the toggle flip-flop description. Without them, you cannot initialize toggle flip-flops to a known state.

This section describes toggle flip-flops with an asynchronous set or reset and toggle flip-flops with an enable and an asynchronous reset.

Toggle Flip-Flop With Asynchronous Set or Reset

Example 6-45 shows the template for a toggle flip-flop with asynchronous set. HDL Compiler generates the verbose inference report shown in Example 6-46. Figure 6-19 shows the flip-flop.

Example 6-45 Toggle Flip-Flop With Asynchronous Set

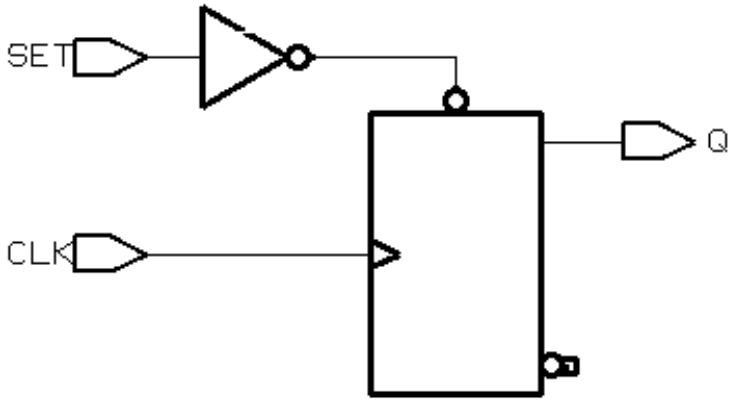
```
module t_async_set (SET, CLK, Q);
  input SET, CLK;
  output Q;
  reg Q;

always @ (posedge CLK or posedge SET)
  if (SET)
    Q <= 1;
   else
    Q <= ~Q;
endmodule
```

Example 6-46 Inference Report for a Toggle Flip-Flop With Asynchronous Set

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
TMP_Q_reg	Flip-flop	1	-	-	N	Y	N	N	Y

```
TMP_Q_reg
Async-set: SET
Sync-toggle: true
```

Figure 6-19 Toggle Flip-Flop With Asynchronous Set



Example 6-47 provides the Verilog template for a toggle flip-flop with asynchronous reset. Example 6-48 shows the verbose inference report. Figure 6-20 shows the inferred flip-flop.

Example 6-47 Toggle Flip-Flop With Asynchronous Reset

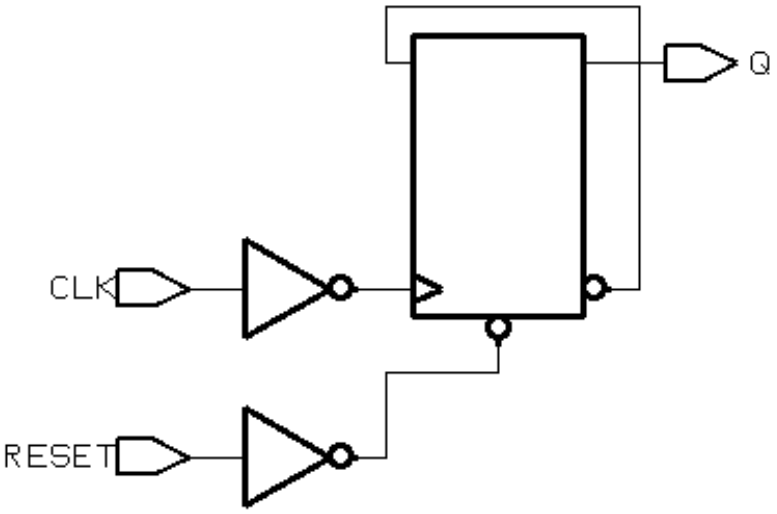
```
module t_async_reset (RESET, CLK, Q);
  input RESET, CLK;
  output Q;
  reg Q;

always @ (posedge CLK or posedge RESET)
  if (RESET)
    Q <= 0;
  else
    Q <= ~Q;
endmodule
```

Example 6-48 Inference Report: Toggle Flip-Flop With Asynchronous Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
TMP_Q_reg	Flip-flop	1	-	-	Y	N	N	N	Y

```
TMP_Q_reg
Async-reset: RESET
    Sync-toggle: true
```

Figure 6-20 Toggle Flip-Flop With Asynchronous Reset



Toggle Flip-Flops With Enable and Asynchronous Reset

Example 6-49 provides the Verilog template for a toggle flip-flop with an enable and an asynchronous reset. The flip-flop toggles only when the enable (TOGGLE signal) has a logic 1 value. HDL Compiler generates the verbose inference report shown in Example 6-50. Figure 6-21 shows the inferred flip-flop.

Example 6-49 Toggle Flip-Flop With Enable and Asynchronous Reset

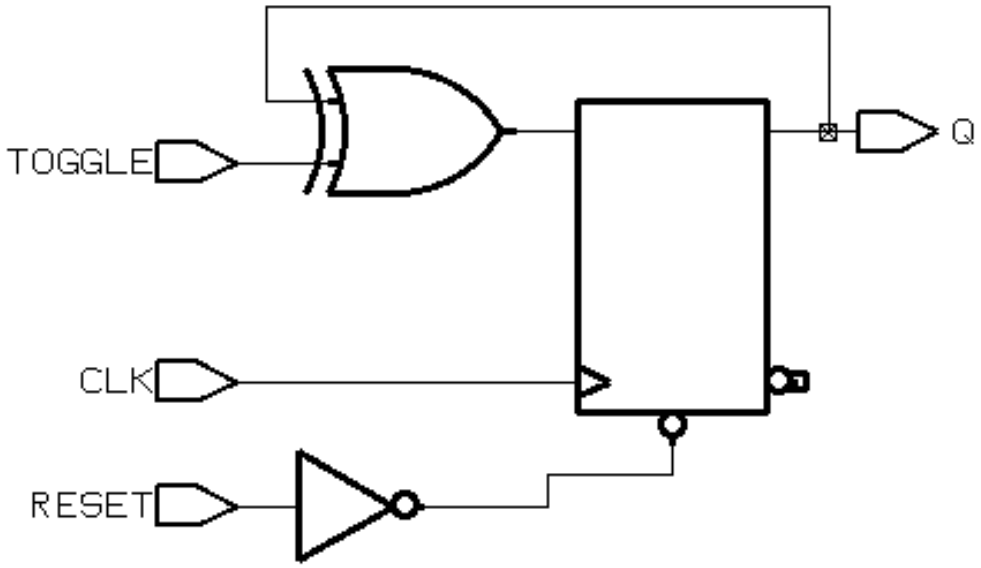
```
module t_async_en_r (RESET, TOGGLE, CLK, Q);
  input RESET, TOGGLE, CLK;
  output Q;
  reg Q;
always @ (posedge CLK or posedge RESET)
begin : infer
  if (RESET)
    Q <= 0;
  else if (TOGGLE)
    Q <= ~Q;
end
endmodule
```

Example 6-50 Inference Report: Toggle Flip-Flop With Enable and Asynchronous Reset

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
TMP_Q_reg	Flip-flop	1	-	-	Y	N	N	N	Y

```
TMP_Q_reg
Async-reset: RESET
Sync-toggle: TOGGLE
```

Figure 6-21 Toggle Flip-Flop With Enable and Asynchronous Reset



Getting the Best Results

This section provides tips for improving the results you achieve during flip-flop inference. Topics include

- Minimizing flip-flop count
- Correlating synthesis results with simulation results

Minimizing Flip-Flop Count

An `always` block that contains a clock edge in the sensitivity list causes HDL Compiler to infer a flip-flop for each variable assigned a value in that `always` block. It might not be necessary to infer as flip-flops all variables in the `always` block. Make sure your HDL description builds only as many flip-flops as the design requires.

The description in Example 6-51 builds six flip-flops, one for each variable assigned a value in the `always` block (`COUNT[2:0]`, `AND_BITS`, `OR_BITS`, and `XOR_BITS`).

Example 6-51 Circuit With Six Implied Registers

```
module count (CLK, RESET, AND_BITS, OR_BITS, XOR_BITS);
  input CLK, RESET;
  output AND_BITS, OR_BITS, XOR_BITS;
  reg AND_BITS, OR_BITS, XOR_BITS;

  reg [2:0] COUNT;

always @(posedge CLK) begin
  if (RESET)
    COUNT <= 0;
  else
    COUNT <= COUNT + 1;

  AND_BITS <= & COUNT;
  OR_BITS <= | COUNT;
  XOR_BITS <= ^ COUNT;
end
endmodule
```

In this design, the outputs—`AND_BITS`, `OR_BITS`, and `XOR_BITS`—depend solely on the value of the variable `COUNT`. If the variable `COUNT` is inferred as a register, these three outputs are unnecessary.

To compute values synchronously and store them in flip-flops, set up an `always` block with a signal edge trigger. Put the assignments you want clocked in the `always` block with the signal edge trigger, `posedge CLK` in the synchronous block in Example 6-52.

To let other values change asynchronously, put these assignments in a separate `always` block with no signal edge trigger, as shown in the asynchronous block in Example 6-52.

You avoid inferring extra flip-flops by using this style in your description.

Example 6-52 Circuit With Three Implied Registers

```
module count (CLK, RESET,
              AND_BITS, OR_BITS, XOR_BITS);
  input CLK, RESET;
  output AND_BITS, OR_BITS, XOR_BITS;
  reg AND_BITS, OR_BITS, XOR_BITS;

  reg [2:0] COUNT;

//synchronous block
always @(posedge CLK) begin
  if (RESET)
    COUNT <= 0;
  else
    COUNT <= COUNT + 1;
end
//asynchronous block
always @(COUNT) begin
  AND_BITS <= & COUNT;
  OR_BITS <= | COUNT;
  XOR_BITS <= ^ COUNT;
end
endmodule
```

The technique of separating combinational logic from registered or sequential logic is useful for describing state machines. See the following examples in Appendix A:

- “Count Zeros—Combinational Version” on page A-2
- “Count Zeros—Sequential Version” on page A-5
- “Drink Machine—State Machine Version” on page A-8
- “Drink Machine—Count Nickels Version” on page A-13
- “Carry-Lookahead Adder” on page A-15

Correlating With Simulation Results

Using delay specifications with registered values can cause the simulation to behave differently from the logic HDL Compiler synthesizes. For example, the description in Example 6-53 contains delay information that causes Design Compiler to synthesize a circuit that behaves unexpectedly (the post-synthesis simulation results do not match the pre-synthesis simulation results).

Example 6-53 Delays in Registers

```
module flip_flop (D, CLK, Q);
     input D, CLK;
     output Q;
     .
endmodule

module top (A, C, D, CLK);
     .
     reg B;

always @ (A or C or D or CLK)
begin
     B <= #100 A;
     flip_flop F1(A, CLK, C);
     flip_flop F2(B, CLK, D);
end
endmodule
```

In Example 6-53, B changes 100 nanoseconds after A changes. If the clock period is less than 100 nanoseconds, output D is one or more clock cycles behind output C during simulation of the design. However, because HDL Compiler ignores the delay information, A and B change values at the same time and so do C and D. This behavior is not the same as in the post-synthesis simulation.

When using delay information in your designs, make sure that the delays do not affect registered values. In general, you can safely include delay information in your description if it does not change the value that gets clocked into a flip-flop.

Understanding the Limitations of Register Inference

HDL Compiler cannot infer the following components. You must instantiate these components in your Verilog description.

- Flip-flops and latches with three-state outputs
- Flip-flops with bidirectional pins
- Flip-flips with multiple clock inputs
- Multiport latches
- Register banks

  Note:

  Although you can instantiate flip-flops with bidirectional pins, Design Compiler interprets these cells as black boxes.

Multibit Inference

A multibit component (MBC), such as a 16-bit register, reduces the area and power in a design. But the primary benefits of MBCs are the creation of a more uniform structure for layout during place and route and the expansion of the synthesized area of a design.

Multibit inference allows you to map registers, multiplexers, and three-state cells to regularly structured logic or multibit library cells. Multibit library cells (macro cells, such as 16-bit banked flip-flops) have these advantages:

- Smaller area and delay, due to shared transistors (as in select or set/reset logic) and optimized transistor-level layout

  In the use of single-bit components, the select or set/reset logic is repeated in each single-bit component

- Reduced clock skew in sequential gates, because the clock paths are balanced internally in the hard macro implementing the MBC
- Lower power consumption by the clock in sequential banked components, due to reduced capacitance driven by the clock net
- Better performance, due to the optimized layout within the MBC
- Improved regular layout of the data path

  Note:

  The term *multibit component* refers, for example, to a 16-bit register in your HDL description. The term *multibit library cell* refers to a library macrocell, such as a flip-flop cell.

Controlling Multibit Inference

To direct HDL Compiler to infer multibit components, do one of the following:

- Embed a directive in the Verilog description.

  The directive gives you control over individual wire and register signals.

- Use a dc_shell variable.

  dc_shell variables apply to an entire design.

Directives That Control Multibit Inference

The directives for Verilog are `infer_multibit` and `dont_infer_multibit`.

Set the Verilog directives on wire and register signals to infer multibit components (see Example 6-54 on page 6-61 and Example 6-55 on page 6-61).

Variable That Controls Multibit Inference

The following dc_shell variable controls multibit inference:

`hdlin_infer_multibit`

This variable controls multibit inference for all bused registers, multiplexers, and three-state cells you input in the same dc_shell session. Set this variable before reading in the HDL source. You can select from the following settings for this variable.

`default_none`
Infers multibit components for signals that have the `infer_multibit` directive in the Verilog description. This is the default value.

`default_all`
Infers multibit components for all bused registers, multiplexers, and three-state cells. Use the `dont_infer_multibit` directive to disable multibit mapping for certain signals.

Design Compiler infers multibit components for all bused register, multiplexer, or three-state cells that are larger than 2 bits. If you want to implement as single-bit components all buses that are more than 4 bits, use the following command:

```
set_multibit_options -minimum_width 4
```

This sets a `minimum_multibit_width` attribute on the design.

`never`
Does not infer multibit components, regardless of the attributes or directives in the HDL source.

Inferring Multibit Components

There are two methodologies for inferring multibit components:

- Directing multibit inference from the HDL source

  This is the best methodology for designers who are familiar with the design's layout and able to determine where multibit components have the largest impact.

- Directing multibit inference from a mapped design

  This is the best methodology for designers who complete an initial synthesis run and then a quick placement and routing.

  At that point, it is easier to determine

  - If multibit components would benefit certain areas of the design
  - If the multibit components already inferred are causing routing congestion

To adjust the design after layout, use the following commands:

`create_multibit`

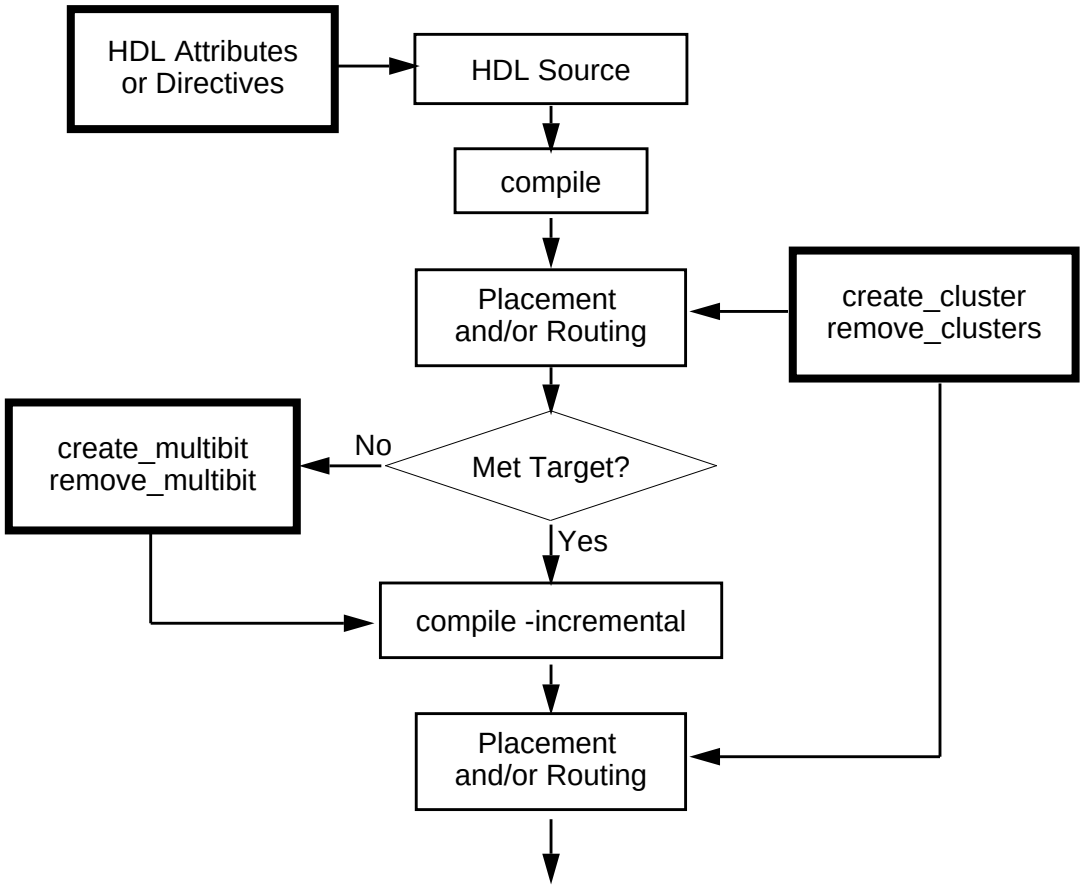
Infers multibit components in a mapped design

`remove_multibit`

Removes multibit components from a mapped design

Figure 6-22 illustrates these methodologies.

Figure 6-22 Design Flow of User-Directed Multibit Cell Inference



Inferring Multibit Cells From HDL Source With the hdlin_infer_multibit variable

If you know where multibit components will work well in your design, inferring multibit cells from HDL source is the best methodology to use.

Use the `hdlin_infer_multibit` variable to indicate the default behavior of all bused register, multiplexer, and three-state cells in your design.

Set the `hdlin_infer_multibit` variable before reading in the HDL source. Unless you change the variable, it will control multibit inferencing in all subsequent HDL files read in during the current dc_shell session.

Inferring Multibit Cells From HDL Source With infer_multibit and dont_infer_multibit Directives

In conjunction with the `hdlin_infer_multibit` variable (described in “Variable That Controls Multibit Inference” on page 6-57), use the `infer_multibit` and `dont_infer_multibit` directives to describe designs that are primarily multibit or primarily single-bit.

Multibit components may not be efficient in the following instances:

- As state machine registers
- In small bused logic that would benefit from single-bit design

Example 6-54 and Example 6-55 show the use of the `infer_multibit` and `dont_infer_multibit` directives.

Example 6-54 shows the use of `infer_multibit` to infer multibit inference of certain signals.

Example 6-55 shows the same HDL code but illustrates how to prevent multibit inference of certain signals.

Example 6-54 Inferring a 6-Bit 4-to-1 Multiplexer

```
module mux4to1_6 (select, a, b, c, d, z);
input [1:0] select;
input [5:0] a, b, c, d;
output [5:0] z;
reg [5:0] z;
//synopsys infer_multibit "z"

    always@(select or a or b or c or d)
    begin
        case (select)         // synopsys infer_mux
        2'b00: z <= a;
        2'b01: z <= b;
        2'b10: z <= c;
        2'b11: z <= d;
    endcase
    end
endmodule
```

Example 6-55 Not Inferring a 6-Bit 4-to-1 Multiplexer

```
module mux4to1_6 (select, a, b, c, d, z);
input [1:0] select;
input [5:0] a, b, c, d;
output [5:0] z;
reg [5:0] z;
//synopsys dont_infer_multibit "z"

    always@(select or a or b or c or d)
    begin
        case (select)         // synopsys infer_mux
        2'b00: z <= a;
        2'b01: z <= b;
        2'b10: z <= c;
        2'b11: z <= d;
    endcase
    end
endmodule
```

Reporting Multibit Inference

HDL Compiler generates an inference report, which shows the information HDL Compiler passes on to Design Compiler about the inferred devices.

Example 6-56 shows a multibit inference report.

Example 6-56 Multibit Inference Report

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
q_reg	Latch	4	Y	Y	N	N	-	-	-

block name/line	Inputs	Outputs	# sel inputs	MB
proc1/23	4	7	2	Y

Three-State Device Name	Type	MB
q_tri_0	Three-State Buffer	Y
s_tri_3	Three-State Buffer	N
q_tri_1	Three-State Buffer	Y
s_tri_0	Three-State Buffer	N

Example 6-56 indicates which cells are inferred as multibit components. The column MB, for sequential cells, indicates whether the vectored component is inferred as a multibit component. The MB column also appears in inference reports for three-state cells and multiplexer cells.

Using the report_multibit Command

In addition to receiving the inference report HDL Compiler generates, you can issue the `report_multibit` command, which lets you report all multibit components in the current design. The report, viewable before and after compile, shows the multibit group name and what cells implement each bit.

Example 6-57 shows a multibit component report.

Example 6-57 Multibit Component Report

Multibit Component : alt178/syn11718					
Cell	Reference	Library	Area	Width	Attributes
U813	mx4a1x16	cba_core_mb	96.00	16	
U9101	mx4a1x16	cba_core_mb	96.00	16	
Total 2 cells			192.00	32	

Multibit Component : data_reg					
Cell	Reference	Library	Area	Width	Attributes
data_reg[0:15]	1d1a2x16	cba_core_mb	48.00	16	n
data_reg[16:31]	1d1a2x16	cba_core_mb	48.00	16	n
Total 2 cells			96.00	32	

The multibit group name for registers and three-state cells is set to the name of the bus. In the cell names of the multibit registers with consecutive bits, a colon separates the outlying bits.

If the colon conflicts with the naming requirements of your place and route tool, you can change the colon to another delimiter by using the `bus_range_separator_style` variable.

For multibit library cells with nonconsecutive bits, a comma separates the nonconsecutive bits. This delimiter is controlled by the `bus_multiple_separator_style` variable. For example, a 4-bit banked register that implements bits 0, 1, 2, and 5 of bus `data_reg` is named `data_reg[0:2,5]`.

For multiplexer cells, the name is set to the cell name of the MUX_OP before optimization.

Listing All Multibit Cells in a Design

To generate a list of all multibit cells in the design, use the new Design Compiler object multibit in a find command, as shown here:

```
find (multibit, “*”)
```

Understanding the Limitations of Multibit Inference

You can infer as multibit components only register, multiplexer, and three-state cells that have identical structures for each bit.

Note:

Multibit inference of other combinational multibit cells occurs only during sequential mapping of multibit registers. Multibit sequential mapping does not pull in as many levels of logic as single-bit sequential mapping. Thus, Design Compiler might not infer a complex multibit sequential cell, such as a JK flip-flop, which could adversely affect the quality of the design.

See the *Design Compiler Reference Manual: Optimization and Timing Analysis* for more information about how Design Compiler handles multibit components.

Multiplexer Inference

Hardware designers often use multiplexers to implement conditional assignments to signals. HDL Compiler can infer a generic multiplexer cell (MUX_OP) from case statements in your Verilog description. Unlike with register inference, HDL Compiler also can infer MUX_OPs from case statements in subprograms. Design Compiler maps inferred MUX_OPs to multiplexer cells in the target technology library.

Note:

If you want to use the multiplexer inference feature, the target technology library must contain at least a 2-to-1 multiplexer.

MUX_OPs are hierarchical cells similar to Synopsys DesignWare components. Design Compiler determines the MUX_OP implementation during compile, based on the design constraints. See the *Design Compiler Reference Manual: Optimization and Timing Analysis* for information about how Design Compiler maps MUX_OPs to multiplexers in the target technology library.

Reporting Multiplexer Inference

HDL Compiler generates an inference report that shows the information the compiler passes on to Design Compiler about the inferred devices. The `hdlin_report_inferred_modules` variable has no effect on the multiplexer inference report.

Example 6-58 shows a MUX_OP inference report.

Example 6-58 MUX_OP Inference Report

Statistics for MUX_OPs

block name/line	Inputs	Outputs	# sel inputs	MB
blk1/20	2	1	1	N

The first column of the report indicates the block that contains the case statement for which the MUX_OP is inferred. The line number of the case statement in Verilog also appears in this column. The remaining columns indicate the number of inputs, outputs, and select lines on the inferred MUX_OP.

Controlling Multiplexer Inference

You can embed an HDL Compiler directive in the Verilog description or use dc_shell variables to direct HDL Compiler to infer MUX_OPs. The directive gives you control over individual `case` statements, whereas dc_shell variables apply to an entire design.

HDL Compiler Directive That Controls Multiplexer Inference

Set the `infer_mux` directive on a block to direct HDL Compiler to infer MUX_OPs for all case statements in that block. You can also set the `infer_mux` directive on specific `case` statements to limit MUX_OP inference to that `case` statement.

Attach the `infer_mux` directive to a block, by using the following syntax:

```
// synopsys infer_mux block_label_list
```

Attach the infer_mux directive to a case statement by using the following syntax:

```
case (var) //synopsys infer_mux
```

Variables That Control Multiplexer Inference

The following dc_shell variables control multiplexer inference:

hdlin_infer_mux = default

This variable controls MUX_OP inference for all designs you input in the same dc_shell session. You can select from the following settings for this variable:

Use default to infer MUX_OPs for case statements in blocks that have the infer_mux directive attached. This is the default value.

The value of none does not infer MUX_OPs, regardless of the directives set in the Verilog description. HDL Compiler generates the following message during MUX_OP inference when this variable is set to none:

```
Warning: A mux for process %s was not inferred because
the variable hdlin_infer_mux was set to none. (HDL-384)
```

The value of all treats each case statement in the design as if the infer_mux directive is attached to it. This can negatively affect the quality of results, because it might be more efficient to implement the MUX_OPs as random logic instead of using a specialized multiplexer structure. Use this setting only if you want MUX_OPs inferred for every case statement in your design.

`hdlin_dont_infer_mux_for_resource_sharing = true`

When this variable is true, HDL Compiler does not infer a MUX_OP when two or more synthetic operators drive the data inputs of the MUX_OP. HDL Compiler generates the following message during MUX_OP inference when this variable is true:

```
Warning: No mux inferred for the case %s because it would
lose the benefit of resource sharing. (HDL-380)
```

When this variable is false, HDL Compiler infers a MUX_OP but resource sharing does not share the data pins that the synthetic operators drive. This can have a negative impact on the area of the final implementation. HDL Compiler generates the following message during MUX_OP inference when this variable is false:

```
Warning: A mux has been inferred for case %s which may
lose the benefit of resource sharing. (HDL-381)
```

`hdlin_mux_size_limit = 32`

If the number of branches in a `case` statement exceeds the maximum size specified by this variable, HDL Compiler generates the following message:

```
Warning: A mux was not inferred because case statement
%s has a very large branching factor. (HDL-383)
```

This variable sets the maximum size of a MUX_OP that HDL Compiler can infer. If you set this variable to a value greater than 32, HDL Compiler takes longer to process the design.

The following dc_shell variables control how Design Compiler maps the MUX_OPs:

`compile_create_mux_op_hierarchy = true`

When this variable is true, the `compile` command creates all MUX_OP implementations with their own level of hierarchy. When it is false, the `compile` command removes this level of hierarchy.

`compile_mux_no_boundary_optimization = false`

When this variable is false, the `compile` command performs boundary optimization on all MUX_OP implementations. When true, the `compile` command does not perform these boundary optimizations.

For more information about these variables, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

Inferring Multiplexers

This section contains Verilog examples that infer MUX_OPs.

The size of the inferred MUX_OP depends on the number of unique values that are read in the `case` statement. During compilation, Design Compiler attempts to map the MUX_OP to an appropriately sized multiplexer in the target technology library. If the library does not contain a large enough multiplexer, Design Compiler builds the multiplexer with smaller multiplexer cells (such as 4-to-1 multiplexer cells).

Example 6-59 attaches the `infer_mux` directive to the block blk1. HDL Compiler infers a MUX_OP for each case statement in the block. The first `case` statement reads eight unique values and infers an 8-

to-1 MUX_OP. The second `case` statement reads four unique values and infers a 4-to-1 MUX_OP. HDL Compiler generates the inference report shown in Example 6-60.

Example 6-59 Multiplexer Inference for a Block

```
module muxtwo(DIN1, DIN2, SEL1, SEL2, DOUT1, DOUT2);
  input [7:0] DIN1;
  input [3:0] DIN2;
  input [2:0] SEL1;
  input [1:0] SEL2;
  output DOUT1, DOUT2;
  reg DOUT1, DOUT2;

//synopsys infer_mux ”blk1”

always @(SEL1 or SEL2 or DIN1 or DIN2)
begin: blk1
  // this case statement infers an 8-to-1 MUX_OP
  case (SEL1)
    3’b000: DOUT1 <= DIN1[0];
    3’b001: DOUT1 <= DIN1[1];
    3’b010: DOUT1 <= DIN1[2];
    3’b011: DOUT1 <= DIN1[3];
    3’b100: DOUT1 <= DIN1[4];
    3’b101: DOUT1 <= DIN1[5];
    3’b110: DOUT1 <= DIN1[6];
    3’b111: DOUT1 <= DIN1[7];
  endcase

  // this case statement infers an4-to-1 MUX_OP
  case (SEL2)
    2’b00: DOUT2 <= DIN2[0];
    2’b01: DOUT2 <= DIN2[1];
    2’b10: DOUT2 <= DIN2[2];
    2’b11: DOUT2 <= DIN2[3];
  endcase
end
endmodule
```

Example 6-60 Inference Report for a Block

Statistics for MUX_OPs

block name/line	Inputs	Outputs	# sel inputs	MB
blk1/53	4	1	2	N
blk1/29	8	1	3	N

Example 6-61 uses the `infer_mux` directive for a specific case statement. This `case` statement reads eight unique values, and HDL Compiler infers an 8-to-1 MUX_OP. HDL Compiler generates the inference report shown in Example 6-62.

Example 6-61 Multiplexer Inference for a Specific case Statement

```
module mux8to1 (DIN, SEL, DOUT);
  input [7:0] DIN;
  input [2:0] SEL;
  output DOUT;
  reg DOUT;

always@(SEL or DIN)
begin: blk1
  case (SEL)  // synopsys infer_mux
    3'b000: DOUT <= DIN[0];
    3'b001: DOUT <= DIN[1];
    3'b010: DOUT <= DIN[2];
    3'b011: DOUT <= DIN[3];
    3'b100: DOUT <= DIN[4];
    3'b101: DOUT <= DIN[5];
    3'b110: DOUT <= DIN[6];
    3'b111: DOUT <= DIN[7];
  endcase
end
endmodule
```

Example 6-62 Inference Report for case Statement

Statistics for MUX_OPs

block name/line	Inputs	Outputs	# sel inputs	MB
blk1/19	8	1	3	N

Understanding the Limitations of Multiplexer Inference

HDL Compiler does not infer MUX_OPs for

- `if...else` statements
- `case` statements that contain two or more synthetic operators, unless you set the following variable to false before inputting the Verilog description:
  `hdlin_dont_infer_mux_for_resource_sharing`
- `case` statements in `while` loops

HDL Compiler does infer MUX_OPs for incompletely specified `case` statements, but the resulting logic might not be optimal. HDL Compiler generates the following message when inferring a MUX_OP for an incompletely specified `case` statement:

```
Warning: A mux has been inferred for case %s which has either
a default clause or an incomplete mapping. (HDL-382)
```

HDL Compiler considers the following types of `case` statements incompletely specified:

- `case` statements that have a missing `case` statement branch or a missing assignment in a `case` statement branch
- `case` statements that contain an `if` statement or `case` statements that contain other `case` statements

Three-State Inference

HDL Compiler infers a three-state driver when you assign the value of z to a variable. The z value represents the high-impedance state. HDL Compiler infers one three-state driver per block. You can assign high-impedance values to single-bit or bused variables.

Reporting Three-State Inference

HDL Compiler can generate an inference report that shows the information the compiler passes on to Design Compiler about the inferred devices. The `hdlin_report_inferred_modules` variable controls the generation of the three-state inference report. See “Reporting Register Inference” on page 6-2 for more information about the `hdlin_report_inferred_modules` variable. For three-state inference, HDL Compiler generates the same report for the default and the verbose reports.

Example 6-63 shows a three-state inference report.

Example 6-63 Three-State Inference Report

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N

The first column of the report indicates the name of the inferred three-state device. The second column indicates the type of three-state device HDL Compiler inferred.

Controlling Three-State Inference

HDL Compiler always infers a three-state driver when you assign the value of z to a variable. HDL Compiler does not provide any means of controlling the inference.

Inferring Three-State Drivers

This section contains Verilog examples that infer the following types of three-state drivers:

- Simple three-state drivers
- Registered three-state drivers

Simple Three-State Driver

This section provides a template for a simple three-state driver. In addition, it provides examples of how allocating high-impedance assignments to different blocks affects three-state inference.

Example 6-64 provides the Verilog template for a simple three-state driver. HDL Compiler generates the inference report shown in Example 6-65. Figure 6-23 shows the inferred three-state driver.

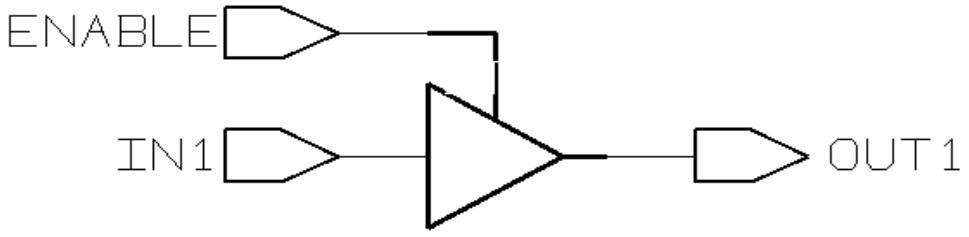
Example 6-64 Simple Three-State Driver

```
module three_state (ENABLE, IN1, OUT1);
  input IN1, ENABLE;
  output OUT1;
  reg OUT1;

always @(ENABLE or IN1) begin
  if (ENABLE)
    OUT1 = IN1;
  else
    OUT1 = 1'bz;  //assigns high-impedance state
end
endmodule
```

Example 6-65 Inference Report for Simple Three-State Driver

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N

Figure 6-23 Schematic of Simple Three-State Driver



Example 6-66 provides an example of placing all high-impedance assignments in a single block. In this case, the data is gated and HDL Compiler infers a single three-state driver. Example 6-67 shows the inference report. Figure 6-24 shows the schematic the code generates.

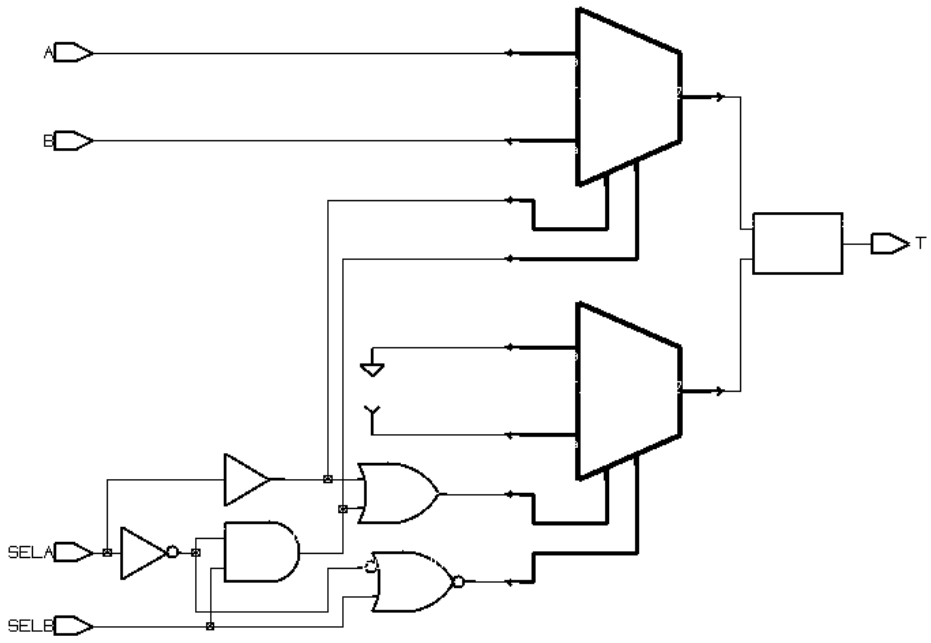
Example 6-66 Inferring One Three-State Driver From a Single Block

```
module three_state (A, B, SELA, SELB, T);
  input  A, B, SELA, SELB;
  output T;
  reg T;

always @(SELA or SELB or A or B) begin
  T = 1'bz;
  if (SELA)
    T = A;
  if (SELB)
    T = B;
end
endmodule
```

Example 6-67 Single Block Inference Report

Three-State Device Name	Type	MB
T_tri	Three-State Buffer	N

Figure 6-24 One Three-State Driver Inferred From a Single Block



Example 6-68 provides an example of placing each high-impedance assignment in a separate block. In this case, HDL Compiler infers multiple three-state drivers. Example 6-69 shows the inference report. Figure 6-25 shows the schematic the code generates.

Example 6-68 Inferring Three-State Drivers From Separate Blocks

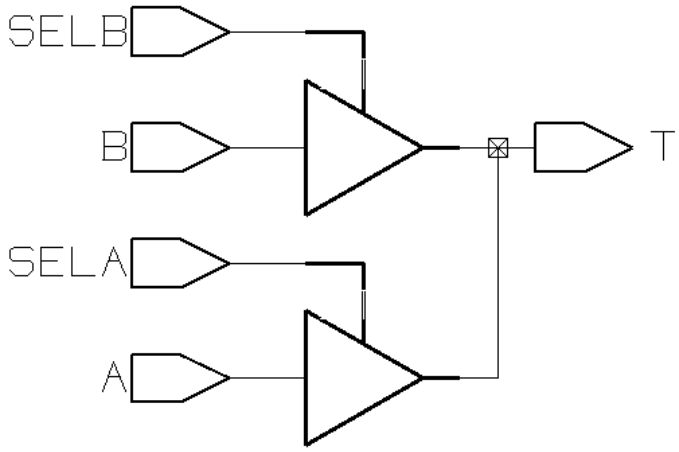
```
module three_state (A, B, SELA, SELB, T);
  input  A, B, SELA, SELB;
  output T;
  reg T;

always @(SELA or A)
  if (SELA)
    T = A;
  else
    T = 1'bz;

always @(SELB or B)
  if (SELB)
    T = B;
  else
    T = 1'bz;
endmodule
```

Example 6-69 Inference Report for Two Three-State Drivers

Three-State Device Name	Type	MB
T_tri	Three-State Buffer	N

Three-State Device Name	Type	MB
T_tri2	Three-State Buffer	N

Figure 6-25 Two Three-State Drivers Inferred From Separate Blocks



Registered Three-State Drivers

When a variable is registered in the same block in which it is defined as three-state, HDL Compiler also registers the enable pin of the three-state gate. Example 6-70 shows an example of this type of code. Example 6-71 shows the inference report. Figure 6-26 shows the schematic generated by the code.

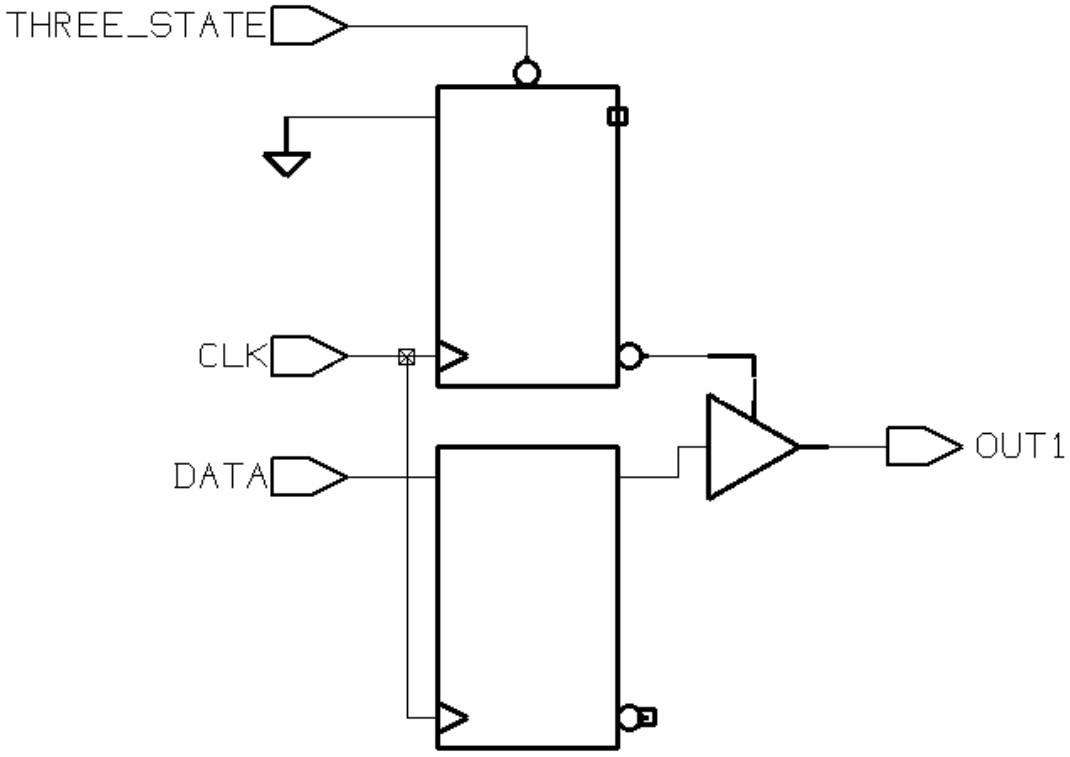
Example 6-70 Three-State Driver With Registered Enable

```
module ff_3state (DATA, CLK, THREE_STATE, OUT1);
  input DATA, CLK, THREE_STATE;
  output OUT1;
  reg OUT1;

always @ (posedge CLK) begin
  if (THREE_STATE)
    OUT1 = 1'bz;
  else
    OUT1 = DATA;
end
endmodule
```

Example 6-71 Inference Report for Three-State Driver With Registered Enable

Three-state Device Name	Type	MB
OUT1_tri	Three-State Buffer	N
OUT1_tr_enable_reg	Flip-flop (width 1)	N

Figure 6-26 Three-State Driver With Registered Enable



In Figure 6-26, the three-state gate has a register on its enable pin. Example 6-72 uses two blocks to instantiate a three-state gate, with a flip-flop only on the input. Example 6-73 shows the inference report. Figure 6-27 shows the schematic the code generates.

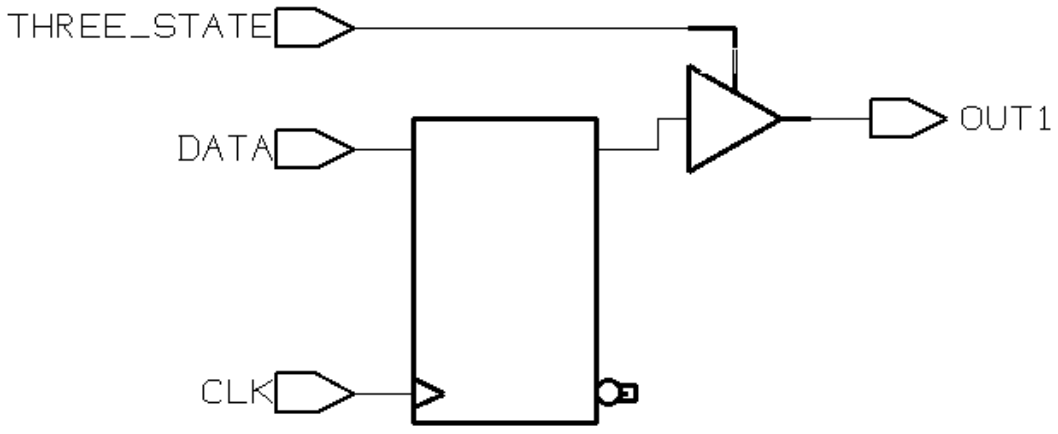
Example 6-72 Three-State Driver Without Registered Enable

```
module ff_3state (DATA, CLK, THREE_STATE, OUT1);
  input DATA, CLK, THREE_STATE;
  output OUT1;
  reg OUT1;

  reg TEMP;

always @(posedge CLK)
  TEMP <= DATA;

always @(THREE_STATE or TEMP)
  if (THREE_STATE)
    OUT1 = TEMP;
  else
    OUT1 = 1'bz;
endmodule
```

Example 6-73 Inference Report for Three-State Driver Without Registered Enable

Three-State Device Name	Type	MB
OUT1_tri	Three-State Buffer	N

Figure 6-27 Three-State Driver Without Registered Enable



Understanding the Limitations of Three-State Inference

You can use the z value in the following ways:

- Variable assignment
- Function call argument
- Return value

You cannot use the z value in an expression, except for comparison with z. Be careful when using expressions that compare with the z value. Design Compiler always evaluates these expressions to false, and the pre-synthesis and post-synthesis simulation results might differ. For this reason, HDL Compiler issues a warning when it synthesizes such comparisons.

This is an example of incorrect use of the z value in an expression:

```
OUT_VAL = (1'bz && IN_VAL);
```

This is an example of correct use of the z value in an expression:

```
if (IN_VAL == 1'bz) then
```