The D-Type Flip-Flop

The D-type flip-flop is an edge-triggered memory device (cell primitive) that transfers a signal's value on its D input, to its Q output, when an active edge transition occurs on its clock input. The output value is held until the next active clock edge. The Q-bar output signal is always the inverse of the Q output signal, see Figure 7.3. A bank of flip-flops clocked from a common clock signal is often referred to as a register.

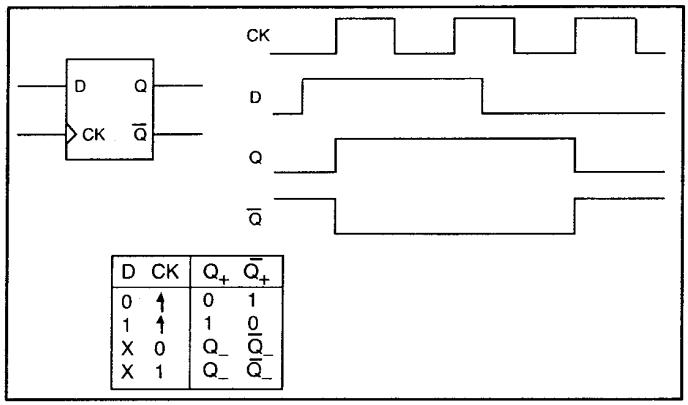


Figure 7.3 The edge triggered D-type flip-flop

Like the latch, there are usually many variants of the flip-flop in an ASIC or FPGA technology library. A flip-flop may have a rising or falling edge triggered clock. It may, or may not, have preset and clear inputs which may be active high or low, and which may be synchronous or asynchronous with the clock.

A circuit, whose sequential elements consist only of D-type flip-flops, can be designed and verified quicker and easier than if latches were used. For this reason, flip-flops are usually preferred over latches. Latches with phased enable signals are used to reduce circuit timing when timing becomes a critical issue.

Flip-Flops are inferred differently in VHDL and Verilog and are described separately.

VHDL flip-flop inference

Flip-flops are inferred in VHDL using **wait** or **if** statements within a process. The difference from latch inferencing is that instead of detecting the occurrence of a signal's level, a signal's edge is now detected. Example edge detecting expressions are:

```
Clock'event and Clock = '1'        -- rising edge detection using 'event attribute
Clock'event and Clock = '0'        -- falling edge detection using 'event attribute
not Clock'stable and Clock = '1'   -- rising edge detection using 'stable
not Clock'stable and Clock = '0'   -- falling edge detection using 'stable
rising_edge(Clock)                 -- rising edge detection using a function call
falling_edge(Clock)                -- falling edge detection using a function call
```

Example use of these edge expressions in **wait** or **if** statements are as follow:

```
Wait until (Clock'event and Clock = '1');
if (Clock'event and Clock = '0') then
wait until rising_edge(Clock);
if falling_edge(Clock) then
```

The above edge detection methods use either VHDL attributes, for example 'event, or function calls, for example rising_edge or falling_edge. The functions rising_edge and falling_edge also use these VHDL attributes. Use of function calls simplifies a model slightly and is preferred, especially

if using multi-valued data types, like for example std_logic, that has nine possible values, {U, X, 0, 1, Z, W, L, H, -}. The reason function calls are preferred is that in order to detect a rising edge (logic 0 to 1 transition) for a signal of type std_logic, it is necessary to ensure transitions like X to 1 are not detected.

example,

```
Clock is of type std_logic.

-- Attribute 'event detects X to 0 and X to 1 transitions which may not be a transition at all
if (Clock'event and Clock = '0') then  -- Detects X to 1 transitions

-- Attribute 'event detects only 0 to 1 transitions
if (Clock'event and Clock'last_value = '0' and Clock = '1') then

-- Detects only logic 0 to logic 1 transitions and has simplified code
if rising_edge(Clock) then
```

Models that are to be simulated and synthesized, an assumption made throughout this book, should use multi-valued data types, and so from the above description, it is better to use function calls. Almost all edge detections throughout this book use function calls, mostly rising_edge, except for the examples in this section showing the use of attributes. Functions rising_edge and falling_edge are defined in the IEEE 1164 package STD_Logic_1164 for clock signals of type std_logic and in the IEEE 1076.3 synthesis package NUMERIC_BIT for clocks of type bit.

Wait *versus* **if**. The **wait** and **if** statements can be used for level detection to infer latches and edge detection to infer flip-flops. The **wait** statement delays the execution of the whole process until its expression becomes true. This means all other signal assignments in the process will infer one or more flip-flops depending on a signal's bit width. Synthesis tools only allow one **wait** statement in a process and it should be the first statement within the process. Because the **if** statement does not stop the execution of the whole **process** it does not prohibit separate purely combinational logic from also being modeled in the same **process**. For this reason the **if** statement is normally preferred over the **wait** statement.

Examples 7.7 and 7.8 use both **wait** and **if** statements, though for the reason just stated, all other examples in this book use **if** statements.

Verilog flip-flop inference

Flip-flops are only inferred using edge triggered **always** statements and so this is similar to using the **wait** statement in VHDL. The Verilog **always** statement is edge-triggered by including either a **posedge** or **negedge** clause in the event list. Combinational logic may be modeled on the inputs to the flip-flops, but independent combinational logic may not be modeled in the same **always** statement. Purely combinational logic must be modeled in a separate **always** statement. For this reason, certain VHDL models may need to be modeled differently in Verilog. Example 7.10 in the LFSR section shows one such case where two **always** statements in Verilog equate to one **process** statement in VHDL.

Example sequential **always** statements:

```
always @(posedge Clock)
always @(negedge Clock)
always @(posedge Clock or posedge Reset)
always @(posedge Clock or negedge Reset)
always @(negedge Clock or posedge Reset)
always @(negedge Clock or negedge Reset)
```

If an asynchronously reset flip-flop is being modeled a second **posedge** or **negedge** clause is needed in the event list of the **always** statement. Also, most synthesis tools require that the reset must be used in an **if** statement directly following the **always** statement, or after the **begin** if it is in a sequential **begin-end** block.

example

```
// Active low asynchronous reset
always @(posedge Clock or negedge Reset)
  begin
    if (! Reset)
      ...
      ...
  end
```

Example 7.8 shows VHDL **if** and **wait** statements and Verilog synchronous **always** statements used to model flip-flops with a positive or negative edge triggered clock.

Example 7.9 shows the inference of numerous flip-flop variants having active high (logic 1) or low (logic 0) synchronous and asynchronous set, reset and enable inputs.

Example 7.7 Flip-flops (+ve/-ve clocked) - VHDL attributes and function calls

This is the only example that uses VHDL attributes, for example, 'event, for signal edge detection. The normal function call edge detection is also included for comparison. The model infers flip-flops with a positive or negative edge triggered clock. If the target technology does not contain negative edge triggered flip-flops a positive edge triggered flip-flop will be inferred and the clock signal will be inverted through a separately inferred inverter.

VHDL. Both **if** and **wait** statements use the 'event attribute and rising_edge and falling_edge function calls. Outputs Y1, Y2, Y3 and Y4 are derived using the 'event attribute while outputs Y5, Y6, Y7 and Y8 are derived using function calls. Modeled are four different ways of modeling a positive edge-triggered flip-flop (Y1, Y3, Y5 and Y7), and four different ways of modeling a negative edge-triggered flip-flop (Y2, Y4, Y6 and Y8).

Verilog. There is only one way to model either a positive edge-triggered flip-flop or negative edge triggered flip-flop as indicated below.

+ve and -ve clocked flip-flops - VHDL model uses attributes and function calls

VHDL

```
library IEEE;
use IEEE.STD_Logic_1164.all, IEEE.Numeric_STD.all;

entity FF_POS_NEG_CLK is
   port ( Clock:          in  std_logic;
          A1, A2, A3, A4: in  bit;
          A5, A6, A7, A8: in  std_logic;
          Y1, Y2, Y3, Y4: out bit;
          Y5, Y6, Y7, Y8: out std_logic);
end entity FF_POS_NEG_CLK;

architecture RTL of FF_POS_NEG_CLK is
begin
   P1: process (Clock)
   begin
      if (Clock 'event and Clock = '1') then
         Y1 <= A1;
      end if;
      if (Clock 'event and Clock = '0') then      continued
```

Verilog

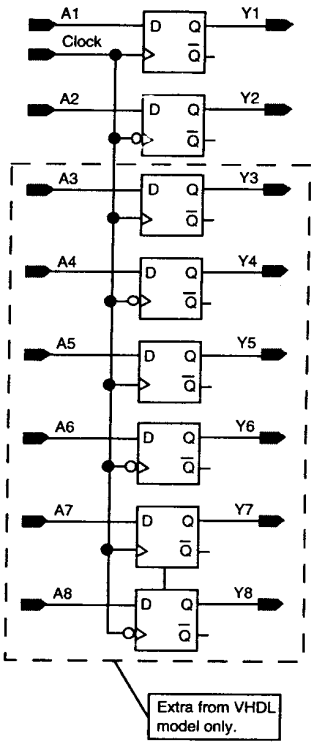
```
module FF_POS_NEG_CLK (Clock, A1, A2, Y1, Y2);
      input Clock;
      input A1, A2;
      output Y1, Y2;

      reg Y1, Y2;

   always @(posedge Clock)
      Y1 = A1;

   always @(negedge Clock)
      Y2 = A2;

endmodule
```

+ve and -ve clocked flip-flops - VHDL model uses attributes and function calls

VHDL

```
        Y2 <= A2;
    end if;
  end process P1;

  P2: process
  begin
    wait until (Clock 'event and Clock = '1');
        Y3 <= A3;
  end process P2;

  P3: process
  begin
    wait until (Clock 'event and Clock = '0');
        Y4 <= A4;
  end process P3;

  P4: process (Clock)
  begin
    if rising_edge(Clock) then
        Y5 <= A5;
    end if;

    if falling_edge(Clock) then
        Y6 <= A6;
    end if;
  end process P4;

  P5: process
  begin
    wait until rising_edge(Clock);
        Y7 <= A7;
  end process P5;

  P6: process
  begin
    wait until falling_edge(Clock);
        Y8 <= A8;
  end process P6;

end architecture RTL;
```

Synthesized Circuit



Example 7.8 Various flip-flop models

Different flip-flops with enable inputs, and asynchronous and synchronous resets are modeled. The coding style conforms to that described earlier in this section. An ASIC library, or more probably an FPGA library, may not have all the flip-flop types modeled in this example. This means extra logic gates are inferred with a flip-flop that is in the library to ensure the synthesized circuit maintains correct functionality.

Various filip-flop inferences

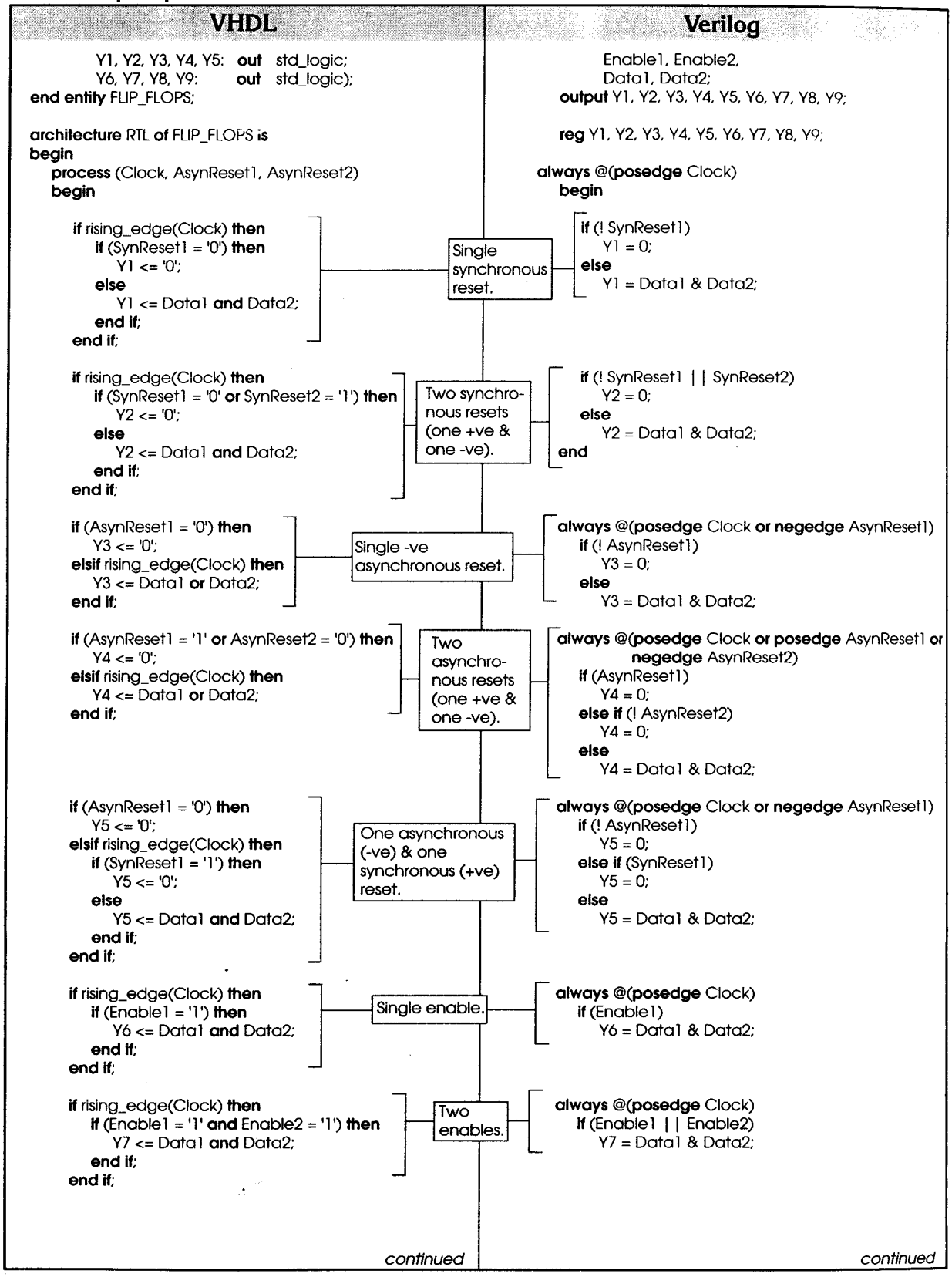
VHDL

```
library IEEE;
use IEEE.STD_Logic_1164.all;

entity FLIP_FLOPS is
  port ( Clock,
         SynReset1, SynReset2,
         AsynReset1, AsynReset2,
         Enable1, Enable2,
         Data1, Data2:  in   std_logic;        continued
```

Verilog

```
module FLIP_FLOPS (Clock,
  SynReset1, SynReset2, AsynReset1, AsynReset2, Enable1,
  Enable2, Data1, Data2, Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8, Y9);
    input Clock,
          SynReset1, SynReset2,
          AsynReset1, AsynReset2,        continued
```

Various filip-flop inferences



Various filip-flop inferences

VHDL

```
if rising_edge(Clock) then
   if (SynReset1 = '1') then
      Y8 <= '0';
   elsif (Enable1 = '1' and Enable2 = '1') then
      Y8 <= Data1 and Data2;
      Y8 <= Y8;
end if;

if (AsynReset1 = '1') then
   Y9 <= '0';
elsif rising_edge(Clock) then
   if (SynReset1 = '1') then
      Y9 <= '0';
   elsif (Enable1 = '1' and Enable2 = '1') then
      Y9 <= Data1 and Data2;
   end if;
end if;

end process;
end architecture RTL;
```

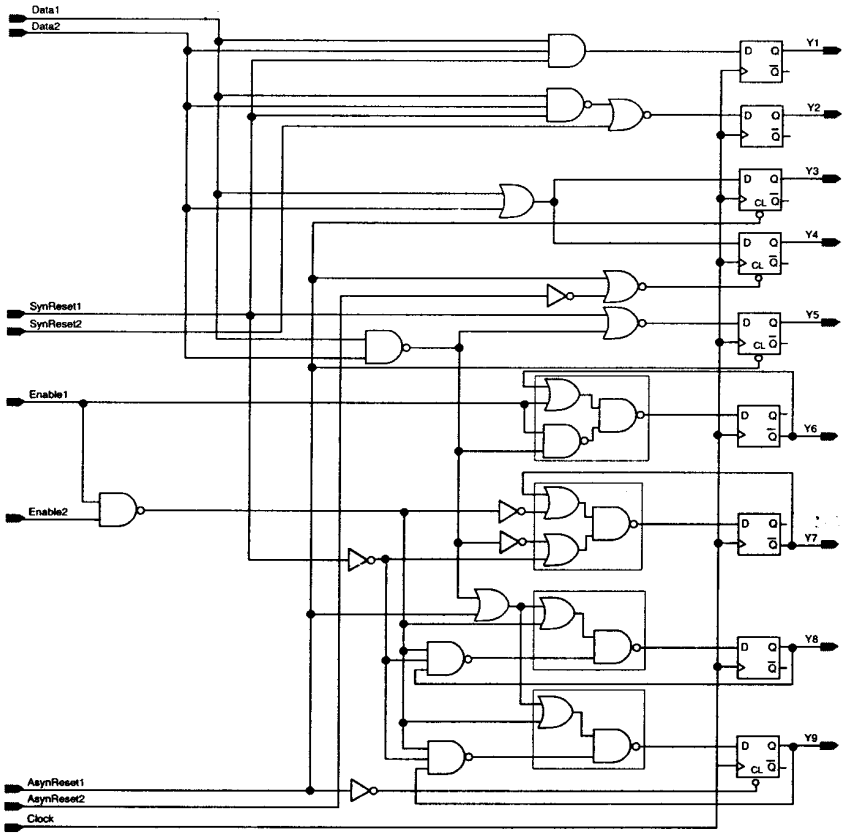
Synchro-nous reset and enable.

One asynchronous reset (+ve), one synchronous reset (+ve) and two enables.

Verilog

```
always @(posedge Clock)
   if (SynReset1)
      Y8 = 0;
   else if (Enable1)
      Y8 = Data1 & Data2;

always @(posedge Clock or posedge AsynReset1)
   if (AsynReset1)
      Y9 = 0;
   else if (SynReset1)
      Y9 = 0;
   else if (Enable1 | | Enable2)
      Y9 = Data1 & Data2;

endmodule
```

Synthesized Circuit



Example 7.9 Combinational logic between two flip-flops

This example is similar to Example 7.1, but infers flip-flops instead of latches. Two flip-flops are modeled with combinational logic on the input to the first flip-flop and between them both. This is achieved with a single **process** (VHDL)/**always** (Verilog) statement.

VHDL. Signal assignments in an edge triggered section of code infer one or more flip-flops. In this example signals M and Y both infer a single flip-flop. Because signal M is used in the expression for the assignment to Y, the output from one flip-flop feeds the input to the other. As data object N is a variable, it does not infer a flip-flop. The new computed value of N in the second assignment is used in computing the value of Y in the third assignment.

Verilog. The explicit assignment to N must appear in a separate, non-edge sensitive, **always** block to avoid inferring a third flip-flop. Also, the assignment to M uses a non-blocking signal assignment so that the NAND of A and B appears on the input to the first flip-flop. If a blocking signal assignment were used the NAND of A and B would feed the input to the NOR gate and the first flip-flop would be redundant.

Combinational logic between two flip-flops

VHDL

```
library IEEE;
use IEEE.STD_Logic_1164.all, IEEE.Numeric_STD.all;

entity FLIP_FLOP_COMB is
   port ( Clock, A, B, C, D, E: in  std_logic;
          Y:                    out std_logic);
end entity FLIP_FLOP_COMB;

architecture RTL of FLIP_FLOP_COMB is
   signal M: std_logic;
begin
   process (Clock)
      variable N: std_logic;
   begin
      if rising_edge(Clock) then
         M <= (A nand B);
         N := (C or D);
         Y <= not (M or N or E);
      end if;
   end process;
end architecture RTL;
```

Signal M used in assignment to Y in the same edge trigger section of code. Variable N does not infer a flip-flop.

Verilog

```
module FLIP_FLOP_COMB (Clock, A, B, C, D, E, Y);
      input Clock, A, B, C, D, E;
      output Y;

      reg M, N;
      reg Y;

   always @(C or D)
      N = (C | D);

   always @(posedge Clock)
      begin
         M <= ! (A & B);
         Y  = ! (N | M | E);
      end

endmodule
```

Signal M assigned using a non-blocking signal assignment and then used in assignment of Y.

Synthesized Circuit