# Inferring Storage Elements

In our designs, we usually use flip-flops as our storage elements. Sometimes we use latches, but not often. Latches are smaller in size, but create special, often difficult situations for testing and static timing analysis.

Latches are inferred in VHDL by using the IF statement without its matching ELSE. This causes the synthesis to make the logical decision to “hold” the value of a signal when not told to do anything else with it.

The inferred latch is a transparent latch. That is, for as long as enable is high, the q output “sees” the d input transparently.

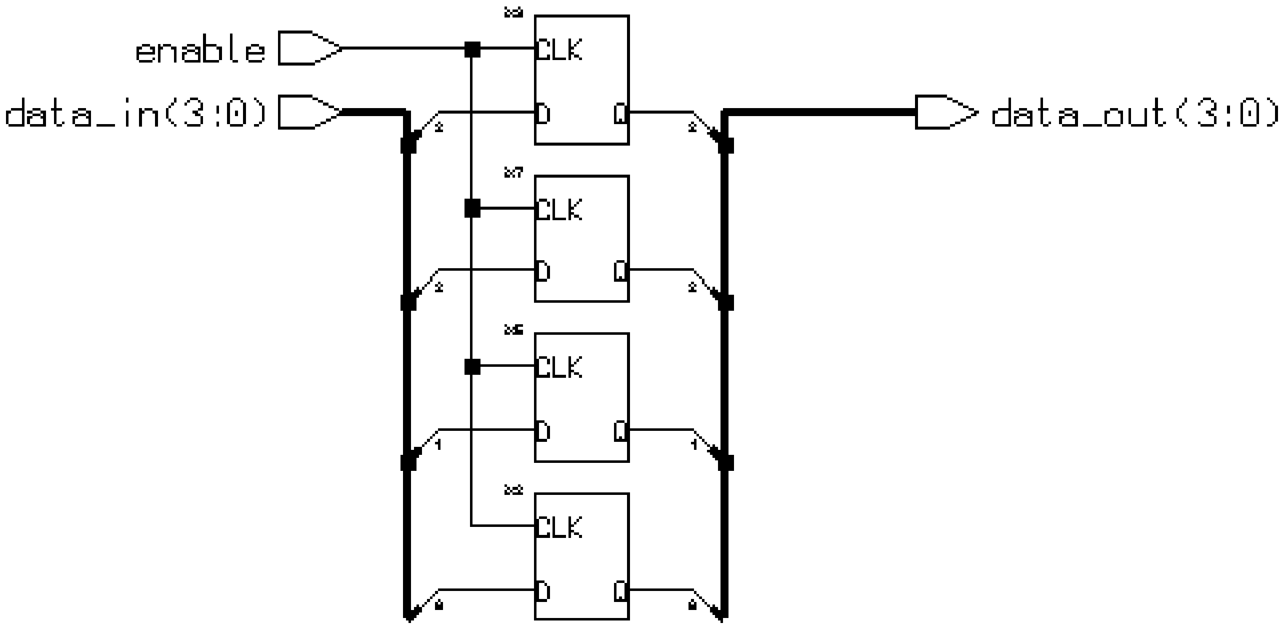
```
--infer 4-bit wide latch
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_vector_arith.ALL;


ENTITY storage  IS
 PORT (
       data_in  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
       data_out : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
       enable   : IN  STD_LOGIC);
END storage;

ARCHITECTURE wed OF storage IS
  BEGIN
  infer_latch:
  PROCESS (enable, data_in)
    BEGIN
      IF enable = ‘1’ THEN
        data_out <= data_in;
      END IF;  --look ma, no else!
  END PROCESS infer_latch;
END ARCHITECTURE wed;
```

When synthesized, we see the following structure:

# Latch Inference



In our library, the enable is shown as going to the “CLK” input of the latch. This is misleading as the input should properly be called “EN” or something like that. If I find the time maybe I’ll change these someday.

The small size of the latches is reflected in the area report:

```
Cell      Library      References      Total Area
latch    ami05_typ        4 x      2     10 gates

Number of gates :                      10
```

This is of course relative to the size of a 2-input NAND gate. In other words, the area of each latch is about the same as 2, 2-input NAND gates!

When we synthesized, the transcript told of the impending latch inference:

```
-- Compiling root entity storage(wed)
"/nfs/guille/u1/t/traylor/ece574/src/storage.vhd",line 8: Warning,
data_out is not always assigned. latches could be needed.
```

Always watch tool transcripts. They can be very informative. Sometime they can save your bacon.

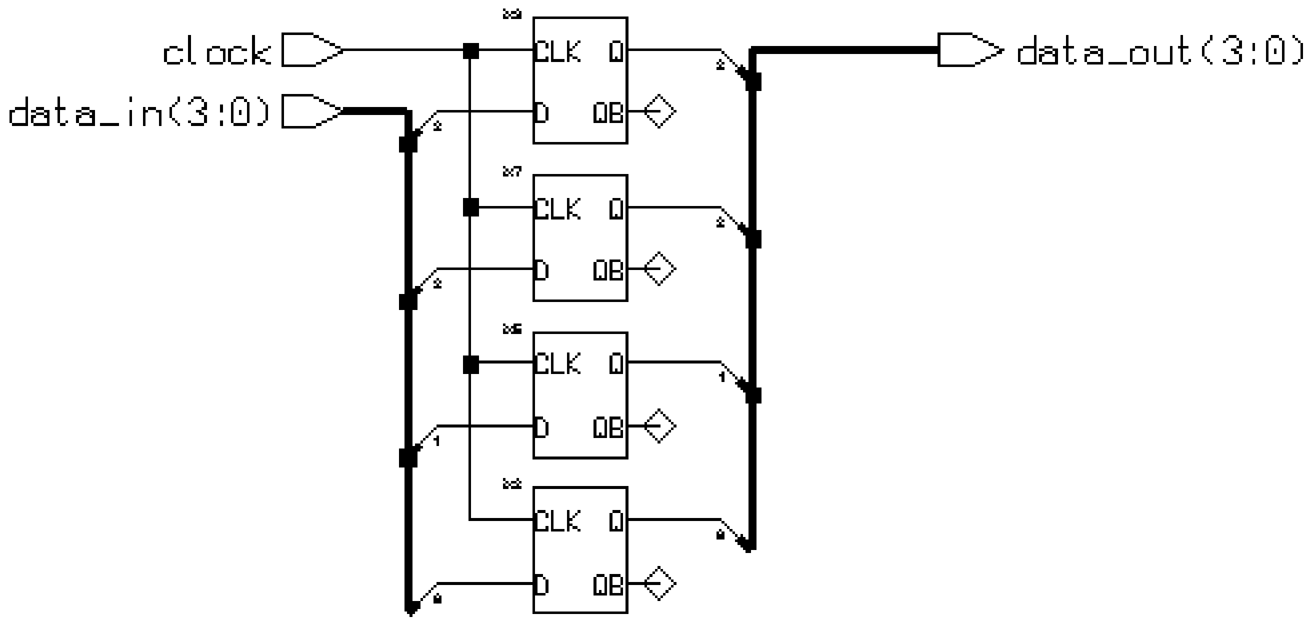
# Inferring D-type Flip Flops

Usually, we want to infer D-type, edge triggered flip flops. Here’s how.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_vector_arith.ALL;

ENTITY storage  IS
 PORT (
       data_in  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
       data_out : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
       clock    : IN  STD_LOGIC);
END storage;

ARCHITECTURE wed OF storage IS
  BEGIN
  infer_dff:
  PROCESS (clock, data_in)
    BEGIN
      IF (clock’EVENT AND clock = ‘1’) THEN
        data_out <= data_in;
      END IF;  --look ma, still no else!.... what gives?
  END PROCESS infer_dff;
END ARCHITECTURE wed;
```



Sometime back we stated that IF with ELSE infers a latch. Well... that is usually true. Here is an exception. The line:

```
     IF (clock’EVENT AND clock = ‘1’) THEN
```

is special to the synthesis tool. The conditional statement for the IF uses the attribute which looks for a change in the signal *clock* (`clock’EVENT`). This is ANDed with the condition that *clock* is now ‘1’ (`AND clock = ‘1’`). The conditional is looking for a rising edge of the signal *clock*.

Therefore, if there is a rising edge, the statement under the IF will be executed and at no other time. So when the clock rises, data_out will get the value present at data_in. Since a D flip-flop is the only cell that can satisfy this condition and can hold the value once it is acquired it is used to implement the circuit. The conditional `(clock’EVENT AND clock = ‘1’)`really forms the recipe for a D-type rising edge flip flop.

A ELSE clause could be added to the IF statement that explicitly tells the old value to be held. This is not at all harmful, but is redundant and is ignored by the synthesis tool. An example of this is shown below:

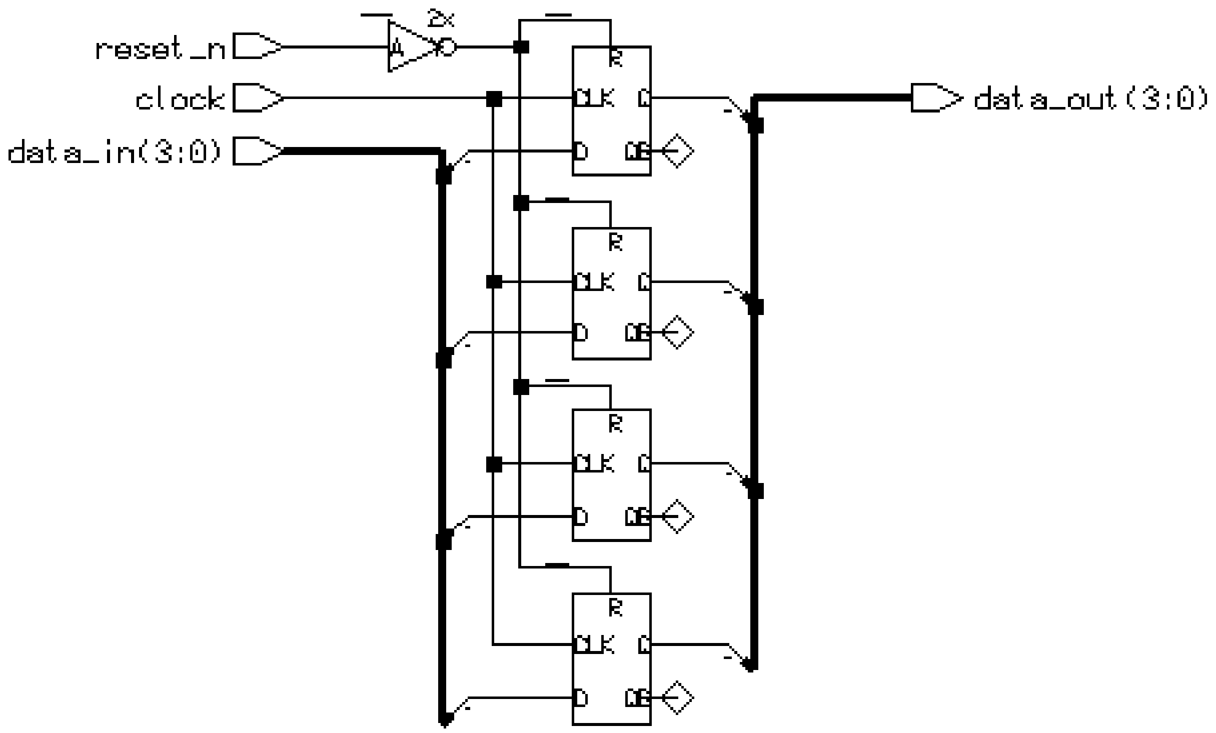
```
infer_dff:
PROCESS (clock, data_in)
  BEGIN
    IF (clock’EVENT AND clock = ‘1’) THEN
      data_out <= data_in;  --get new value
    ELSE
      data_out <= data_out; --hold old value...UNNECESSARY
    END IF;
END PROCESS infer_dff;
```

# Adding an Asynchronous Reset

We almost never want a flip flop without a reset. Without a reset, how can the simulator determine initial state? It cannot. It is very rare to find flip-flops with out a reset. Here is how to code a flip flop with a asynchronous reset:

```
ARCHITECTURE wed OF storage IS
  BEGIN
  infer_dff:
  PROCESS (reset_n, clock, data_in)
    BEGIN
      IF (reset_n = ‘0’) THEN
        data_out <= “0000”;  --aysnc reset
      ELSIF (clock’EVENT AND clock = ‘1’) THEN
        data_out <= data_in;
      END IF;
  END PROCESS infer_dff;
END ARCHITECTURE wed;
```

When synthesized, we get:



## How big is a flip flop/latch?

From the area_report.txt file we see:

```
Cell     Library   References     Total Area
dffr    ami05_typ     4 x      6     24 gates
inv02   ami05_typ     1 x      1      1 gates
Number of gates :     24
```

This looks a little fishy. 24 + 1 = 24? At any rate, (assuming round off error) the flip flops are roughly 6 gates a piece.

So to summarize the relative sizes of latches and flip flops
:

| **CASE** | **CELL** | **SIZE** |
|---|---|---|
| latch no reset | latch | 2 gates |
| latch with reset | latchr | 3 gates |
| flip flop with no reset | dff | 5 gates |
| flip flop with reset | dffr | 6 gates |

These numbers are valid only for our library. Other libraries will vary. However, the relative sizes are consistent with most any CMOS library.