EE 231L

Using AHDL to Design Sequential Circuits

In order to design a sequential circuit, you need to use a logic element with memory – a flip-flop or a latch. AHDL has several types of such elements – a latch, a D flip-flop, a JK flip-flop, an SR flip-flop and a toggle flip-flop. Here we will discuss two of these elements — the latch and the D flip-flop.

Latch A latch in AHDL has two inputs – `D` and `En`, and one output `Q`. When `En` is low, the output `Q` does not change. When `En` is high, the output `Q` is equal to the input `D`. Thus, when `En` is low, it will hold the value which was on the `D` input when `En` went from high to low. To use a latch in AHDL, declare it in the `VARIABLE` section of the program:

```
VARIABLE
A       : LATCH;
```

will define a one-bit latch. To specify what should on the `D` input of `A`, use `A.d`. To specify what should on the `En` input of `A`, use `A.ena`. To use the `Q` output, refer to `A.q`.

Here is the way to specify an eight-bit latch:

```
VARIABLE
A[7..0]       : LATCH;
```

Here is AHDL code to connect the inputs of the latch to input lines called `data_in`, the outputs of the latch to output lines called `data_out`, and the the enable lines to an input called `latch_enable`:

```
SUBDESIGN my_latch
(
data_in[7..0] : INPUT;
latch_enable : INPUT;
data_out[7..0] : OUTPUT;
)
VARIABLE
A[7..0] : LATCH;
BEGIN
A[].d = data_in[];
A[].ena = latch_enable;
data_out[] = A[].q;
END;
```

D flip-flop There are two D-type flip-flops in AHDL – `DFF` and `DFFE` (D flip-flop with enable). `DFF` has the standard D flip-flop inputs and outputs – D input (`D`), clock input (`CLK`), active-low asynchronous clear input (`CLRN`), active-low asynchronous set input (`prn`), and the Q output (`Q`). `DFFE` has another input – enable (`ENA`). The `ENA` input to a `DFFE` must be high for the flip-flop to change state – with `ENA` low, the `Q` output will not change on a clock edge.

To use a `DFFE`, declare it in the `VARIABLE` section of the program:

```
VARIABLE
B      : DFFE;
```

Let's build a simple 3-bit up counter: it will count 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, ... Here is the state transition table:

Present State			Next State		
y_2	y_1	y_0	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

This can be implemented with three D flip-flops with the following Boolean equations:

$Y_2 = y_2\overline{y}_0 + \overline{y}_2 y_1 + \overline{y}_2 y_1 y_0$
$Y_1 = \overline{y}_1 y_0 + y_1\overline{y}_0$
$Y_0 = \overline{y}_0$

Here is an AHDL program to implement a three-bit counter:

```
SUBDESIGN 3count
(   count[2..0]  : OUTPUT;
    clock        : INPUT;
)

VARIABLE
    y[2..0]      : DFFE;     % Three D flip-flops with enable %

BEGIN
    DEFAULTS
        y[].ena  = VCC;      % flip-flops always enabled %
        y[].clrn = VCC;     % flip-flops always enabled %
        y[].prn  = VCC;     % flip-flops always enabled %
    END DEFAULTS;

    y[].clk = clock;        % Use input clock to run flip-flops %
    y2.d = (y2.q & !y0.q) # (y2.q & !y1.q) # (!y2.q & y1.q & y0.q);
    y1.d = (!y1.q & y0.q) # (y1.q & !y0.q);
    y0.d = !y0.q;

    count[] = y[].q;        % Assign the outputs of the flip-flops to the
```

```
                              output of the system %
END;
```

You can also design the counter by specifying the transition table and let AHDL determine the Boolean equations:

```
SUBDESIGN 3count
(   count[2..0]  : OUTPUT;
    clock        : INPUT;
)

VARIABLE
    y[2..0]      : DFFE;     % Three D flip-flops with enable %

BEGIN
    DEFAULTS
        y[].ena  = VCC;      % flip-flops always enabled %
        y[].clrn = VCC;     % flip-flops always enabled %
        y[].prn  = VCC;     % flip-flops always enabled %
    END DEFAULTS;

    y[].clk = clock;        % Specify the clock for the D flip-flops %

    TABLE
        y[2..0].q   =>   y[2..0].d;
        B"000"      =>   B"001";
        B"001"      =>   B"010";
        B"010"      =>   B"011";
        B"011"      =>   B"100";
        B"100"      =>   B"101";
        B"101"      =>   B"110";
        B"110"      =>   B"111";
        B"111"      =>   B"000";
    END TABLE;

    count[] = y[].q;        % Assign the outputs of the flip-flops to the
                              output of the system %
END;
```

However, there is a much easier way to design counters. The inputs to the D flip-flops are the outputs of the D flip-flops plus one:

```
SUBDESIGN 3count
(   count[2..0]  : OUTPUT;
    clock        : INPUT;
)
```

```
VARIABLE
    y[2..0]      : DFFE;     % Three D flip-flops with enable %

BEGIN
    DEFAULTS
        y[].ena  = VCC;      % flip-flops always enabled %
        y[].clrn = VCC;     % flip-flops always enabled %
        y[].prn  = VCC;     % flip-flops always enabled %
    END DEFAULTS;

    y[].clk = clock;        % Specify the clock for the D flip-flops %

    y[].d = y[].q + 1;      % Next count is current count plus one %

    count[] = y[].q;        % Assign the outputs of the flip-flops to the
                              output of the system %
END;
```

This gives you the ability to design very large counters which would be hard to do using other techniques. A 16-bit counter has 2^{16} or 65,536 states. It is difficult to develop the Boolean equations, and impractical to enter a transition table with 65,536 lines. Here is a design for a 16-bit counter:

```
SUBDESIGN 16count
(   count[15..0]   : OUTPUT;
    clock          : INPUT;
)

VARIABLE
    y[15..0]       : DFFE;   % Sixteen D flip-flops with enable %

BEGIN
    DEFAULTS
        y[].ena  = VCC;      % flip-flops always enabled %
        y[].clrn = VCC;      % flip-flops always enabled %
        y[].prn  = VCC;      % flip-flops always enabled %
    END DEFAULTS;

    y[].clk = clock;         % Specify the clock for the D flip-flops %

    y[].d = y[].q + 1;       % Next count is current count plus one %

    count[] = y[].q;         % Assign the outputs of the flip-flops to the
                               output of the system %
END;
```

Another, more powerful, way to design sequential circuits is with state machines. We will discuss how to do this in Lab 4.

When you simulate a sequential circuit, you need to do a *Timing Analysis*. Before you can do a timing analysis, you have to tell Quartus which input is your clock signal. To do this, go to **Assignments — Timing Analysis Settings**. Click on **Individual Clocks**, and select **New**. Give the clock input a name (perhaps *clock*), and select the node which is used for the clock. The **Required fmax** setting is the maximum speed for the clock signal (for now, just choose 1 MHz). The timing analysis will tell you if your circuit can work at that frequency.

After setting up your clock signal, recompile your design. Quartus should no longer give the warning "Found pins functioning as undefined clocks and/or memory enables". Go to **Processing — Simulator Tool**, and make sure the **Simulation mode** is set for **Timing**. You can now create a **Vector Waveform File** to simulate your design.

When you design with sequential circuits, one (or more) of your inputs will function as a clock.