

Clocks and Registers

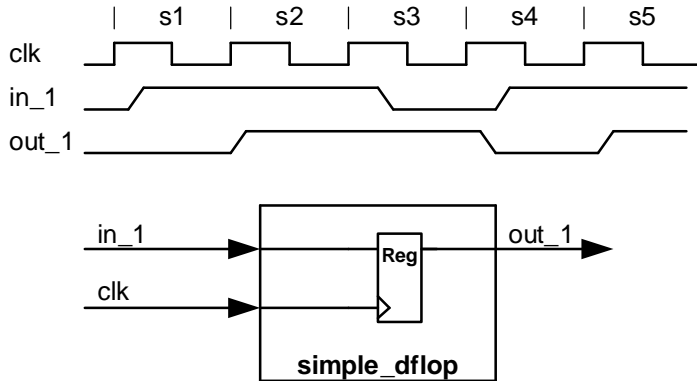
In the introduction, I indicated that this book assumes that you have a working familiarity with digital design. The rubber is about to meet the road.

Clocked state logic comprises the vast majority of the workings of modern FPGAs, and it is here that the true complexity and sophistication of any hardware descriptive language unfolds. The fundamental principles of clocked operation in verilog, though, are straightforward, and easy to grasp if we take them a step at a time.

Until now, our code has consisted of continuous assignments, i.e., direct combinatorial logic. These “assign” statements are continuous in the sense that the output signal (the one being assigned) is continuously responsive to any and all inputs. Any input that changes (and is not gated off by the intervening logic) will immediately affect the output (ignoring physical delays). Contrary to this, registers hold or store information, and therefore require a different coding mechanism called a structured procedural statement. The most common structured procedural statement, and the one used almost exclusively for register implementations, is the “always block.” There are a variety of flavors of this, but for implementation (i.e., synthesis) of clocked registers, we use exclusively the sequential, non-blocking version. That probably doesn’t mean much to you, and that’s okay for now. It is helpful to know that there are other forms in case you may happen across them, but for the time being, an always-block is synonymous with a register.

We’ll begin by implementing the simplest form of a D-flop. Since this represents the basis for the various forms of registers we will continue to encounter, it is labeled as a “Reg.” As shown in the timing diagram, output “out_1” follows “input in_1” at the clocked edges.

Verilog by Example



Simple D-flop

For the sake of brevity we've modified the file format a bit in the code on the opposite page (you'll get used to this as you look across different people's code).

We've added a new declaration for a "reg." This is necessary since we will be implementing output signal "out_1" as a register type. This is in contrast to the "wire" declaration. We have not previously needed to declare outputs explicitly as wires since in verilog outputs default to wire types (it wouldn't have been wrong to declare all the previous outputs as wires, just not necessary).

The section of code shown as the always-block implements the D-flop register. The information inside the parenthesis next to the "@" symbol is called the sensitivity list, and defines which signals can contribute to changes inside the block. Specifically, no activity inside the block can occur unless something in the sensitivity list changes. In the case of our simplest of D-flop registers, the sensitivity list contains just the clock signal. Further, "posedge" defines the flop as rising-edge triggered ("negedge" would be falling-edge triggered).

The operation is easy to see: at every rising clock edge (and only at a rising clock edge), the value of "in_1" is assigned to "out_1". You may wonder why we use the two-part "<=" symbol instead of a simple "=" for the assignment like we did with the combinatorial assignments, and the answer is that this defines it as a

non-blocking assignment. This allows individual elements of more complex always-block structures to operate independently, but the important point is that all synthesized registers use this non-blocking assignment, so get used to it.

The “begin” and “end” lines define the body of the always-block. In this case where there is only one assignment line, the begin/end pair is actually optional, but I recommend always using them for consistency.

```
////////////////////////////////////////
// Simple D-flop
////////////////////////////////////////

module simple_dflop (  clk,
                      in_1,
                      out_1
                      );

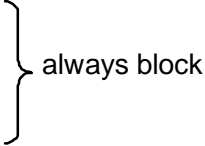
    input      clk;
    input      in_1;
    output     out_1;

    reg        out_1;

    // ----- Design implementation -----

    always @( posedge clk )
    begin
        out_1 <= in_1;
    end

endmodule
```

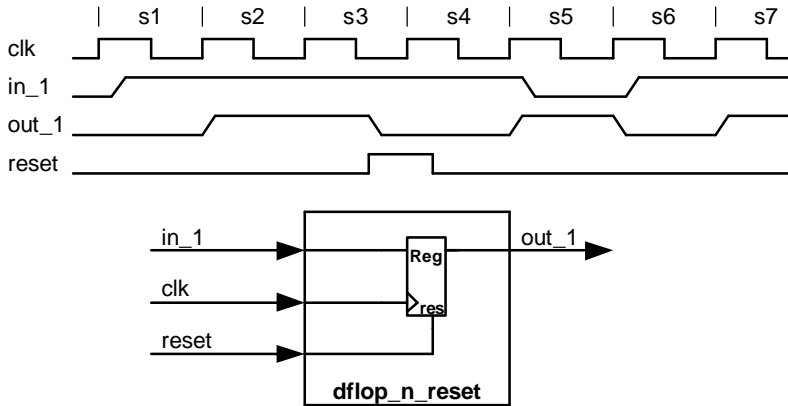


Simple D-flop

Verilog by Example

Next we add an asynchronous reset to our simple D-flop. The timing diagram shows the operation where “reset” forces “out_1” low immediately during state s3, and “out_1” then remains low until clocked again back high at state s5.

Our convention will be that asynchronous controls (resets and presets) will enter the register box at the top or bottom, while all synchronous controls will connect to the front.



D-flop with reset

In the code on the opposite page you can see that the always-block has now grown to accommodate the reset. Since the reset is asynchronous and results in activity immediately, it must be included in the sensitivity list. Tagging it as “posedge” means that it will be high-active—the flop resets as soon as the reset goes high, but after the reset is lifted, the flop doesn’t change until the next clock edge, thus only the rising edge of the reset requires immediate attention.

The body of the always-block has now become more complicated as we introduce if/else conditional statements to accommodate the reset. Any time “reset” is high, “out_1” is forced to zero. Since this happens as soon as reset goes active (reset is part of the sensitivity list), and at every rising clock edge, you can see that this effects an asynchronous clear. When reset is

not high, then the “else” original in-to-out register assignment is selected (occurring only at rising clock edges).

Note that the reset zero assignment is made with “1'b0”. Verilog uses a specific format for static values. The first field defines the number of bits (i.e., the width of the vector), the next field, separated by the apostrophe, defines the radix, and the last field defines the actual value. Since in this case we have a simple one-bit zero, the first field is “1”, and we let the value be defined as binary.

```
////////////////////////////////////////
// D-flop with reset
////////////////////////////////////////

module dflop_n_reset (  clk,
                        reset,
                        in_1,
                        out_1
                        );

    input      clk;
    input      reset;
    input      in_1;
    output     out_1;

    reg        out_1;

    // ----- Design implementation -----

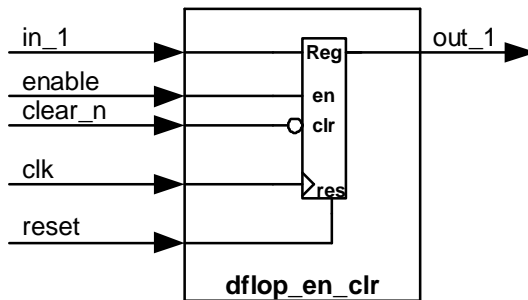
    always @( posedge clk or posedge reset )
        begin
            if ( reset )
                out_1 <= 1'b0;
            else
                out_1 <= in_1;
            end
endmodule
```

D-flop with reset

Verilog by Example

Pressing on, we now add more functionality to our nascent register. Here we introduce two synchronous controls: an enable, and a low-active synchronous clear. We forgo a timing diagram since the operation is self-evident.

Note that the asynchronous reset remains. Besides benefiting from simple consistency, this demonstrates an important point about FPGA design in general: we invariably choose one reset method (synchronous or asynchronous), which is then used globally on all the registers. At a minimum, global resets are necessary for simulation, but additionally may be a practical necessity for proper testing in-circuit. In our case, we will always be using a global asynchronous reset. We should also note that on very large and/or fast designs, the global reset may be segmented into functional domains, but the premise that every flop shares a (semi)common reset remains.



D-flop with enable and clear

The `always`-block in the code on the opposite page expands with the additional synchronous control functions. The asynchronous reset still takes priority (it comes first), but now a low “clear” signal will also force the output to zero as well. However, since this clear signal is not included in the sensitivity list, the change occurs at the next rising clock edge (thus, rendering it synchronous).

Notice that the “`else if`” conditional expression uses a logical equality test, whereas the “`if`” reset line did not. This is because the conditional expression is evaluated as either Boolean true or false. When “`reset`” is a one, its Boolean equivalent is by definition true.

The conditional expression can be as complicated as you like, spanning many lines of code, as long as the synthesis tool is able to determine a final Boolean result.

The final conditional statement implements the clock enable, and here again, since “enable” is high-active, no logic equality test is necessary. Notice that there is no final “else” statement. If there were, the latching operation of the clock enable would be defeated.

As with most other languages, the order of the conditional statements determines the priority.

```
////////////////////////////////////////
// D-flop with enable and clear

module dflop_en_clr (  clk,
                      reset,
                      in_1,
                      enable,
                      clear,
                      out_1
                      );

    input      clk;
    input      reset;
    input      in_1;
    input      enable;
    input      clear;
    output     out_1;

    reg        out_1;

    // ----- Design implementation -----

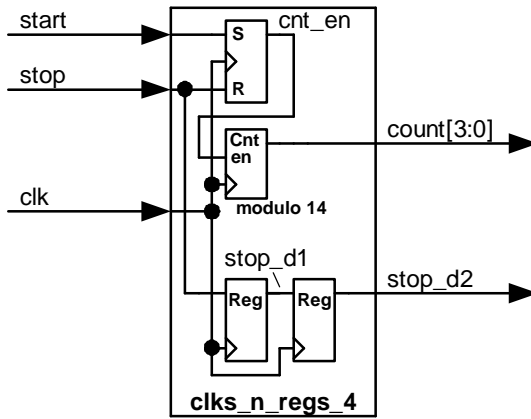
    always @( posedge clk or posedge reset )
    begin
        if ( reset )
            out_1 <= 1'b0;
        else if ( clear == 1'b0 )
            out_1 <= 1'b0;
        else if ( enable )
            out_1 <= in_1;
        end
    endmodule
```

D-flop with enable and clear

Verilog by Example

We now introduce a few common state-type operations to show how increasingly sophisticated register-based functions are implemented in always-blocks. A four-bit counter is enabled by a “start” event, and stopped by a “stop” event. The SR flop allows the start and stop events to be short, e.g. one-clock pulses, rather than a continuously enabling flag. Additionally, for further illustration, we delay the start signal two clocks and send it out.

You’ll notice that we have not shown the asynchronous reset. This is done for clarity; from this point forward it is assumed. It is implemented in the code, and always will be (in this book).



SR flop and counter

The code includes two register declarations for internal signals (`cnt_en` and `stop_d1`), and two register declarations for the two external signals (`count[3:0]` and `stop_d2`). We now have multiple always-blocks. Note that always-blocks operate concurrently, meaning they run simultaneously, independent of each other, just like two registers in a design.

Each always-block is associated with a coherent register function: one for the SR flop, one for the counter, and one for the two delays. The SR flop always-block needs no explanation beyond noting that there is no “else” statement, resulting in a latch function (which is indeed what we desire). The counter always-block also has no “else” statement, but since it is an enabled counter, it is also

in a sense a latch. Notice that since the counter is modulo 14, the first “else if” statement clears it when the count is 13. A couple of things to note here: “4’d13 ” indicates a decimal thirteen, and we’re now using a double “&&” in the conditional expression. This is because “&&” is a Boolean AND (versus the bitwise “&”), which is required for the conditional decision. In the same sense, “||” is a Boolean OR (versus the bitwise “|”). Note that we use “4’h0 ” for clearing the counter. This indicates a hex zero. It could just as well have been 4’b0000, or 4’d0. Similarly, the 4’d13 modulo rollover could have been the slightly less readable 4’hD, or even 4’b1101.

```

////////////////////////////////////////
// SR flop and counter
////////////////////////////////////////

module srflop_n_cntr (  clk,
                        reset,
                        start,
                        stop,
                        count
                        );

    input      clk;
    input      reset;
    input      start;
    input      stop;
    output [3:0] count;

    reg        cnt_en;
    reg [3:0]  count;
    reg        stop_d1;
    reg        stop_d2;

    // ----- Design implementation -----

    // SR flop
    always @( posedge clk or posedge reset )
    begin
        if ( reset )
            cnt_en <= 1'b0;
        else if ( start )
            cnt_en <= 1'b1;
        else if ( stop )
            cnt_en <= 1'b0;
    end

```

Verilog by Example

```
// Counter
always @( posedge clk or posedge reset )
begin
    if ( reset )
        count <= 4'h0;
    else if ( cnt_en
              && count == 4'd13
            )
        count <= 4'h0;
    else if ( cnt_en )
        count <= count + 1;
end

// delay
always @( posedge clk or posedge reset )
begin
    if ( reset )
    begin
        stop_d1 <= 1'b0;
        stop_d2 <= 1'b0;
    end
    else
    begin
        stop_d1 <= stop;
        stop_d2 <= stop_d1;
    end
end
endmodule
```

SR flop and counter

The last always-block implements the two sequential delays. The points to note here are that multiple register signals can be grouped into the same always-block (when it makes sense), and that additional begin/end block boundaries are needed around each pair of signal assignments. Without these, the synthesis software might interpret, for example, that “stop_d2 <= stop_1” is not associated with the “else,” but stands alone.

Finally, we should note that the three always-blocks could be collected together into one. This is shown on the next page.

```
always @( posedge clk or posedge reset )
begin
  if ( reset )
    begin
      cnt_en  <= 1'b0;
      count   <= 4'h0;
      stop_d1 <= 1'b0;
      stop_d2 <= 1'b0;
    end
  else
    begin
      if ( start )
        cnt_en <= 1'b1;
      else if ( stop )
        cnt_en <= 1'b0;

      if ( cnt_en
          && count == 4'd13
        )
        count <= 4'h0;
      else if ( cnt_en )
        count <= count + 1;

      stop_d1 <= stop;
      stop_d2 <= stop_d1;
    end
  end
end
```

SR flop and counter, one always-block

This of course results in more compact code, but the benefit comes with a danger. Extreme care must be taken to make sure there is no ambiguity about what goes with what. If there's any doubt, begin/end block groupings are always available for clarifications.

