Sequential Circuits with Verilog

CS141 Spring 2019

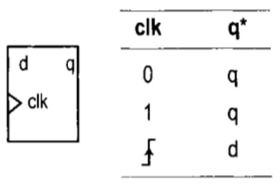
Introduction

We have seen how to build combinational circuits using Verilog, where the output of the circuit depends only on its current inputs. Now we'll look at building sequential circuits, which are circuits with memory, where the output now depends on both the inputs and some internal state. Modern development of sequential circuits follows *synchronous design methodology*. This means that a common clock signal controls all global storage elements and data is stored in these elements only at the rising or falling edge of the clock signal. This allows us to separate the storage components from the rest of the circuit and greatly simplifies development. This methodology is important for designing and verifying large and complex digital systems.

The figures and examples are adapted from *FPGA Prototyping by SystemVerilog Examples* By Pong P. Chu.

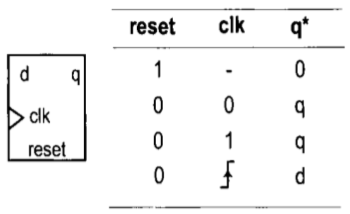
The D FF

The most fundamental data storage component is the D flip flop. The `d` signal is sampled on the rising edge of the clock and stored to the flip flop. A D FF may also contain an asynchronous reset signal which resets the storage to 0 and is independent of the clock signal. The symbol and function of the D FF is shown below.



clk	q*
0	q
1	q
↑	d

(a) D FF



reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↑	d

(b) D FF with asynchronous reset

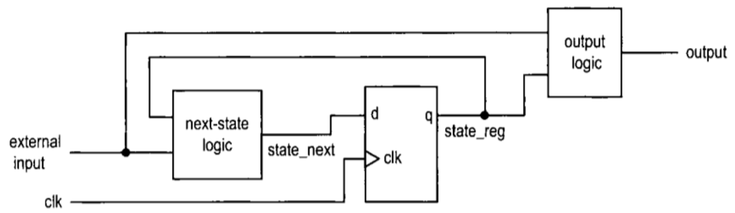
On the left is a simple D FF, and on the right a D FF with asynchronous reset.

A D FF stores one bit, and a collection of D FFs can be put together, called a *register*, to store multiple bits.

General block diagram

A general block diagram for building a sequential circuit is shown below. It consists of the following parts:

- *state register*: a collection of D FFs that store the current internal state of the circuit.
- *next-state logic*: combinational logic that uses the internal state and the input signals to determine the new value of the state register (i.e. what the next state should be).
- *output logic*: combinational logic that generates the output signal from the inputs and the current internal state.



Our code development will follow this basic block diagram. The key is to separate the memory from the rest of the circuit, since the rest is simply combinational logic which we already know how to make.

Always blocks

Here is the simplified syntax for an always block with a *sensitivity list*:

```
always @([sensitivity list])
begin : [optional name]
    [optional local variable declaration];

    [procedural statement];
    [procedural statement];
    ...
end
```

The `[sensitivity list]` is a list of signals to which the always block responds. When any of these signals changes, the always block is triggered. We are especially interested in two special cases of the syntax for the sensitivity list.

The first special case of the sensivity list uses the `posedge` or `negedge` keyword. When this is placed in front of a signal the always block only responds on the specified edge transition of that signal. We will use `@(posedge clk)` to refer to an always block that should be triggered on the rising edge of the clock (i.e. a register).

The second special case is the `*` syntax. If the sensivity list simply consists of `*` this means it should respond to all signals used within it. This is useful for making combinatonial logic, as with combinational logic the outputs should update whenever any inputs are changed. The syntax `always @(*)` may also be abbreviated as `always @*`.

An example will be shown of each type after we discuss what procedural statements can be used in an always block.

Procedural statements

There are many possible procedural statements that can be used within an always block. Many of them don't have a clear physical counterpart and therefore cannot be synthesized. We are only interested in those that can be synthesized into hardware. Our focus will be limited to the following statements:

- Blocking assignment
- Nonblocking assignment
- If statement
- Case statement

Blocking and nonblocking assignments will be discussed next. If statements synthesize to priority routing networks (not discussed) and case statements synthesize to multiplexing routing networks (not discussed).

Procedural assignments

Procedural assignments consist of two types: nonblocking and blocking. They can only be used within an always block or an initial block. The two types are shown below

```
[variable_name] = [expression]; // blocking assignment
[variable_name] <= [expression]; // nonblocking assignment
```

In the blocking assignment, the value of the expression is evaluated and immediately assigned to the variable, thus *blocking* execution of further procedural statements until finished. This is similar to how a variable would behave in C. In a nonblocking assignment, the evaluated expression is only assigned at the end of the always block (thus it does not block the execution of further statements). It may be better to think of nonblocking assignment as "deferred" and blocking assignment as "immediate."

Note: continuous assign statements (i.e. those that use the `assign` keyword) are not valid in always blocks.

Note: procedural assignments cannot assign to variables of the `wire` type and must assign to `reg` type variables. This will be discussed during the first example

Style guidelines

Here are some simple guidelines to follow when designing circuits with `always` to avoid bugs:

- Use `always @(posedge clk)` and nonblocking assignments for a register.
- Use `always @*` and blocking assignments for combinational logic.
- Never assign to a variable in more than one `always` block.

Combinational Example

As an example, we will use an always block to create a 2-to-4 binary decoder:

en	a[1]	a[0]	y
0	-	-	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100

en	a[1]	a[0]	y
1	1	1	1000

```
module decoder_2_4
    (
        input wire [1:0] a,
        input wire en,
        output reg [3:0] y
    );

    always @*
    begin
        if (~en) // no need for 'begin' with one line contents
            y = 4'b0000;
        else if (a == 2'b00)
            y = 4'b0001;
        else if (a == 2'b01)
            y = 4'b0010;
        else if (a == 2'b10)
            y = 4'b0100;
        else
            y = 4'b1000;
    end
endmodule
```

From this point onward, you may use any synthesizable operators in your code such as `==`, `+`, `*`, etc... since we have seen more or less how to implement them at a gate level.

This decoder could also be rewritten using a case statement, but we won't show that for brevity. Take a look on Canvas under the `verilog examples` to see that.

Note: we declare `y` as the `reg` data type instead of `wire`. Contrary to the name, a `reg` variable may not infer to a register. It may simply be a combinational wire (as it is in this case). However, when using procedural assigns, we must assign to `reg` variables, and additionally `reg` variables cannot be passed to output ports of instantiated modules.

Note: make sure when designing combinational logic that every output is assigned to for every branch. Combinational logic cannot store previous values so if you do not assign to a variable during a combinational always block, a register will be inferred for that variable and it will no longer be combinational or what you meant to design.

Sequential Example

The code for a vanilla D flip flop is shown below:

```
module d_ff
    (
        input wire clk,
        input wire d,
        output reg q
    );
    always @(posedge clk)
        q <= d;
endmodule
```

Notice the use of a nonblocking assignment. In general we will always use nonblocking assignments inside of sequential always blocks.

Note: in this case we must again use `reg` for `q` but this time `q` is in fact inferred to a register. This means that the variable `q` is holding state, or in other words its value may depend on previous inputs to the circuit.

More on blocking vs nonblocking

In fact, using a blocking assignment in this case would lead to a subtle bug. Here is an incorrect version:

```
always @(posedge clk)
    q = d;
```

This block is synthesizable and although the code works properly for an isolated FF, there will be problems when multiple registers interact.

Consider two registers that swap data every clock cycle:

```
always @(posedge clk)
    a = b;

always @(posedge clk)
    b = a;
```

At the rising edge of the clock, both always blocks are activated. With a blocking assignment, one assignment will happen before the other (the order of execution is undefined) and when the second assignment happens it will use the new value of the other register, which is in fact its current value. As such, the second register will not change value which is not the intended behavior. This code does not describe the swapping behavior and leads to a race condition where it will behave differently based on which always block is executed first.

Now with a nonblocking assignment, both evaluations will wait before assignment until the end of the always block, where they will both happen at once, giving the intended data swapping behavior.

D FF with reset

Returning to the D FF example, we can also create a flip flop with asynchronous reset:

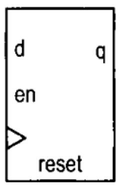
```
module d_ff_reset
    (
        input wire clk, reset,
        input wire d,
        output reg q
    );

    always @(posedge clk, posedge reset)
    begin
        if (reset)
            q <= 1'b0;
        else
            q <= d;
    end
endmodule
```

In this example, the reset may be triggered at any time regardless of the current clock signal and this will change the storage of the flip flop (hence the “asynchronous”).

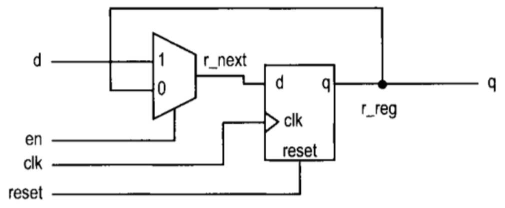
Even though asynchronous reset somewhat violates our synchronous design methodology, we include it because it is often used at machine startup to reset all registers to 0. A flip flop may also contain a `syn_clr` signal which synchronously resets the value of the register at a clock edge.

D FF with synchronous enable

Some D FF implementations include an additional control signal `en` such that the D FF can only sample an input when `en` is high. The `en` signal should only be examined at the rising edge of the clock thus meaning that it is synchronous. The function table and schematic are shown below.



reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↑	0	q
0	↑	1	d



Here is the Verilog code implementing the D FF.

```
module d_ff_en
    (
        input wire clk, reset,
        input wire en,
        input wire d,
        output reg q
    );

    always @(posedge clk, posedge reset)
    begin
        if (reset)
            q <= 1'b0;
        else if (en)
            q <= d;
    end
endmodule
```

Universal Binary Counter

Now we will combine combinational and sequential building blocks to make a universal binary counter. This circuit stores a given value and can do multiple operations on the value each clock cycle depending on the inputs. The following quasi truth table describes the operation:

syn_clr	load	en	up	q*	Operation
1	-	-	-	0	synchronous clear
0	1	-	-	d	parallel load
0	0	1	1	q+1	count up
0	0	1	0	q-1	count down
0	0	0	-	q	pause

Note that parallel load simply means you load all bits of a signal into the register at the same time instead of serially reading one per clock cycle.

The code for this circuit is given below:

```
module univ_bin_counter
    #(parameter N = 8)
    (
        input wire clk, reset, syn_clr, load, en, up,
        input wire [N-1:0] d,
        output wire [N-1:0] q
    );

    // register and wire for next register value
    reg [N-1:0] r_reg, r_next;

    // register
    always @(posedge clk, posedge reset)
    begin
        if (reset)
            r_reg <= 0; // async reset
        else
            r_reg <= r_next;
    end

    // next-state logic (combinational)
    always @*
    begin
        if (syn_clr)
            r_next = 0;
        else if (load)
            r_next = d;
        else if (en & up)
            r_next = r_reg + 1;
        else if (en & ~up)
            r_next = r_reg - 1;
        else
            r_next = r_reg;
    end

    // output logic
    assign q = r_reg;
endmodule
```

Take the time to look over the code for this example and see how the next-state logic and the register logic work together to create the circuit described in the function table. Also note that in this case, there is no complex output logic (we just assign `r_reg` to `q`), but in other circuits there may be output logic to consider and that we why we separate `q` (the output for the circuit), from `r_reg` (the current value stored in the register).