Chapter 4: One-Shots, Counters, and Clocks

I. The Monostable Multivibrator (One-Shot)

The timing pulse is one of the most common elements of laboratory electronics. Pulses can control logical sequences with precise timing. For example, if your detector "sees" a charged particle or a photon, you might want to signal a clock to store the time that the event occurred. In that case, you will use the event to generate a standard pulse so that your clock always responds in the same way. Alternatively, you might need to reset your electronics after the event. Clearly you want the reset pulse to arrive as soon as possible after the data has been processed. This requires a precision time *delay generator*.

A simple type of delay generator is a D type flip-flop that charges up a capacitor after receiving a clock edge. The charged capacitor also serves as the clear input to the D flip-flop, so that after a fixed time (roughly *RC*) the flip-flop resets back to its initial state. The net result is a single pulse that has a duration (or *pulse width*) determined by the combination of the resistor and capacitor. The exact relationship between the time constant and the pulse width is specified in the datasheet for each chip type.

If the falling edge of this pulse triggers other electronics, then you can introduce whatever delay you wish by choosing an appropriate pulse width. This device is called a *monostable multivibrator*, but the common name is the descriptive *one-shot*.

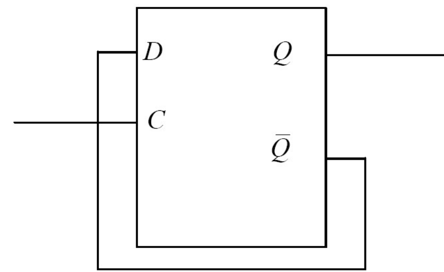
Many one-shots have two clock inputs so that they can be triggered by either a rising edge or a falling edge. The typical one-shot will also have two outputs (Q and $\overline{Q}$) and an reset or clear input, which instantly sets the output to a standard condition regardless of the current state or clock level.

You will find one-shots in all electronic circuits that use pulses and pulse sequences. They are not, however, the best sources of timed pulses. Two effects limit their reliability: (1) a one-shot's pulse length varies with temperature; (2) a one-shot's pulse length varies with *duty cycle*. It they stay high too long they do not reset as fast as they would for short pulses. Thus, one-shots are generally a bad choice for generating square waves. However, they can be very handy in getting signal timings just right in an asynchronous digital circuit.

II. Counters

Last week, you used a D-type flip flop to transfer the data from the *D* input to the *Q* output on the falling edge of a clock. With one more level of feedback, we can convert this into a device that changes state every time the clock edge falls.

If you connect the inverted output to the input then every time the clock edge fall the flip-flop will reverse its output (i.e. $Q_{n+1}=\bar{Q}_n$). This is shown in Figure 5-1. With a square-wave clock input, the output will change on each falling clock edge generating a square wave at half the frequency. This is called a divide-by two circuit.



You can cascade these flip-flops one after another to continue dividing the output frequency. You simply drive the clock of another flip-flop with the output of an earlier flip-flop.

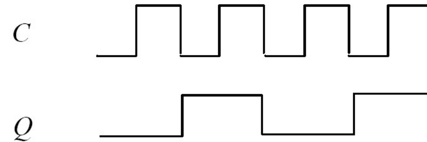


Figure 5-1: D type flip-flop as a divide by two counter.

We can call the state of the first gate b_1 and the state of the second gate b_2 and create a *state table* of the sequence of the states of the two gates for successive clock pulses. From Figure 5-2, you can see that the two bits are actually count in binary. By making a cascade of divide by two circuits you have created a *counter*.

This counter is conceptually simple but it takes time for the clock pulse to propagate down the line of flip-flops. If you imagine many flip-flops connected together in a ripple counter, then each will trigger only after a propagation delay. One triggers the next just like a series of falling dominos. This type of counter is dubbed *ripple counter* to describe this propagating trigger edge.

In *synchronous* counters, however, all stages make their transitions simultaneously. This is usually a much better choice if you have lots of stages (binary digits) in your counter. Of course, the logic is more difficult because you only want a stage to flip states if all the previous stages were set to 1. We will play with synchronous counters next week.

b_1	b_0	next b_1
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5-2: State table for bits b_1 of synchronous counter.

Shift Registers

You can also construct a *shift register* by cascading D-type flip-flop without feedback. To make this device connect all of flip-flops use the same clock. The output of one flip-flop is the input of the next flip-flop.

If data is presented to the first, it works its way down the line of gates at each clock tick. These are great devices to convert between serial data (one bit follows the next in time) and parallel data (several lines holding simultaneous information). It is an example of queuing circuit known as *first in/first out* or *FIFO* buffer. It will store the data in time order and present at it at its output as requested by the clock.

III. Timing with FPGAs and Verilog

FPGAs work best when they are used for synchronous circuits. In fact FPGAs do not include capacitors so you cannot use a one-shot in an FPGA circuit. While, this may seem like a problem, it does not pose any real difficulties since a high-frequency synchronous circuit can easily mimic a one-shot.

Synchronous circuits in Verilog

Synchronous FPGA circuits are implemented in Verilog with the `always` block. All the code, or circuitry, inside an `always` block executes on trigger indicated at the beginning of the `always` block. Here is generic Verilog code for an `always` block:

```
module always_block_example (inputs …, outputs …);
	input input1, input2, …;
	output output1, output2, …;

	output reg [N:0] output_register;
	reg [M:0] variable_register;

	always@ (trigger)
		begin
		…
		[put your always block code here]
		…
		end

endmodule
```

The trigger can be an edge trigger such as `always@ (posedge input1)` or `always@ (negedge input1)`. The trigger can also be a level trigger such as `always@ (input2)`, which means the always block will execute whenever there is a level change in the `input2` value. You can even use an `always` block without a trigger (though this is a little dangerous, since you will then have an infinite loop, and the timing is not well defined):

```
always
	begin
	…
	end
```

The variables that are manipulated and changed inside an `always` block must be declared as type `reg` (i.e. a memory register of flip-flops). The always block can include the following statements:

Blocking assignment: `a = b`

The blocking assignment is executed and then the code moves on to the next instruction (line of code).

Non-blocking assignment: `a <= b`
The non-blocking assignment is executed at the same time as any other sequential block of non-blocking assigments (i.e. all the non-blocking assignments are executed in parallel).

Conditional statement:
```
if (a == b)
    begin
    …
    [the code here will execute if the "if" condition is satified]
    …
    end
```
Conditional statements can be included inside an always block and are a powerful way of manipulating registers or variables.

As a general rule, if you are making a circuit in which timing must be included or in which it could be an issue, then you should use an always block. An always block guarantees that you will be constructing a synchronous circuit. In other words, **always use always.**

Some important coding structures to avoid when using an always block:
1. Nested always blocks.
2. Registers or variables which are manipulated in several different always blocks. This means that several output wires are connected and trying to assign a voltage to the "D" input of a register flip-flop (remember last week's warning: "never tie outputs together").

Register initialization in Verilog

Variables and registers can be initialized in Verilog with an initial block. The initial block is placed at the beginning of a module and is only executed once. Here is an example of how to code an initial block:

```
module always_block_example (inputs …, outputs …);
    input input1, input2, …;
    output output1, output2, …;

    output reg [N:0] output_register;
    reg [M:0] variable_register;

    initial
        begin
        output_register = N'b1011110…0011;
        variable_register = M'b1111100…1101;
        end

    …
```

```
	[the rest of your code goes here]
	…
endmodule
```

A Verilog counter

A counter is easy to implement in Verilog. You use an always block and increment a register variable by one at each trigger, as in the following 4-bit counter example:

```
module counter_verilog(input_clock, counter_register);
	input input_clock;				// declares the input
	output reg [3:0] counter_register;	// declares the output to be a 4-bit
						// register

	initial		// initial block to set the counter to zero
		begin
		// The next line sets counter register to zero
		counter_register = 4'b0000;
		end

	always@ (posedge input_clock)
		begin
		// the following line increments the register by
		// 1 at each clock trigger
		counter_register <= counter_register + 4'b0001;
		end
endmodule
```

Clocks for FPGAs

A synchronous circuit must be triggered by a clock which has a period longer than any of the timing delays in the circuit. A crystal oscillator is frequently used to provide a periodic square wave. The DE2 board is provided with two crystal oscillators, one at 50 MHz and the other at 27 MHz, which are connected to the FPGA at pins PIN_N2 and PIN_D13, repectively. A connection for an external clock is also provided via pin PIN_P26 (see p. 32-33 of DE2 development board manual). Alternatively, the TTL square wave of the function generator can be used as a clock signal.

If actual timekeeping is not important, the frequency of the clock does not have to be very stable, but must only have a period longer than the longest internal timing delay in the circuit.

Design Exercises:

Design Exercise 4-1: Using information from the datasheet for an 74LS123 pick resistors and capacitors to make a pulse of roughly 1 ms and 30 μs.

Design Exercise 4-2: Layout a circuit that uses two one-shots to generate a 30 μs pulse that starts 1 ms after a trigger.

Design Exercise 4-3: Use a single `always` block to construct a Quartus II FPGA project which will generate a 4 clock cycle output pulse that starts 23 clock cycles after an external input trigger goes from low to high. You can assume that the triggering pulse is longer than a single clock cycle.

Design Exercise 4-4: Construct a Quartus II FPGA project for a divide-by-8 circuit which will convert a 1 MHz square wave to a 125 kHz rectangular wave.