VE270 Introduction to Logic Design

Recitation Class 5

CHEN Yuchi

citrate@sjtu.edu.cn

June 17, 2020

Part 1. Verilog

Verilog is a hardware description language (HDL), which is different from other languages you have learned in JI, such as MatLab, C, C++, or Python. So, when using Verilog, you should remember to think "hardwarely".

What you do is to construct a circuit, the output of the circuit is not fixed and depends on the input, but the inner logic will not change.

Syntax

Module

```
module ModuleName(inputs, outputs);
    <some code>;
endmodule
```

Pay attention to the semicolon, which is different from a definition of function in C.

Vivado will create a module by default as following, you should pay attention.

```
module (

);
endmodule
```

Input , Output Declaration

After create a module, you always need to declare which of the parameters is input and which of the parameters is output. You can declare an array input or output by

```
module ModuleName(ip, ipt, opt_1, opt_2);
    input ip;
    input [3:0] ipt;
    output [4:1] opt_1, opt_2;
endmodule
```

The former array is preferred because index always starts from zero. The data type of inputs and outputs are `wire` by default (see next section).

```
module ModuleName(input ip, input [3:0] ipt, output reg [3:0] opt_1, opt_2);
    code;
endmodule
```

Above form is also acceptable.

Data Type

There are three types of data that you will mostly use.

`wire`: Just like true physical wires.

`reg`: Registers, which can store a value.

`<bit width>'<base><number>`: Numbers. For example: `4'b0010`, `3'o5`, `4'ha`, `4'd12`.

Do not omit the quotation mark `'`.

Please pay attention that `<bit width>` is counted by binary numbers, so `1'd9` is not allowed.

Logic

```
and(opt, ipt_1, ipt_2);
or(opt, ipt_1, ipt_2, ipt_3);
xor(opt, ipt_1, ipt_2);
not(opt, ipt);
```

Relational Operators

Operator	Description
a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

Equality Operators (not consider x or z)

Operator	Description
a == b	a equal to b
a != b	a not equal to b

Logical Operators

Operator	Description
!	logic negation
&&	logic and
||	logic or

Bit-wise Operator

Operator	Description
~	negation
&	and
|	inclusive or
^	exclusive or
^~ or ~^	exclusive nor

Assignment

`assign`: Can only be used **outside** `always` block. Can only assign a `wire` variable.

```
wire opt, ipt_1, ipt_2;
opt=ipt_1&ipt_2;
```

Above syntax is not allowed. Below is the correct version.

```
wire opt, ipt_1, ipt_2;
assign opt=ipt_1&ipt_2;
```

To assign a value to a `reg` variable, you can both use `=` and `<=`. A `reg` variable can only be assigned a value **inside** `always` block.

`=` is called blocking assignment and `<=` is called non-blocking assignment. When you have multiple assignment, a non-blocking assignment will evaluate the RHS simultaneously, but a blocking assignment will evaluate current RHS and then evaluate next one (sequentially). Here's an example:

```
reg [3:0] d_0, d_1, d_2, d_3;
wire [3:0] ipt;
always @ (posedge clk) begin
    d_0<=ipt;
    d_1<=d_0;
    d_2<=d_1;
    d_3<=d_2;
end
```

```
reg [3:0] d_0, d_1, d_2, d_3;
wire [3:0] ipt;
always @ (posedge clk) begin
    d_0=ipt;
    d_1<=d_0;
    d_2=d_1;
    d_3=d_2;
end
```

However, the output of below one will not change no matter you write `=` or `<=`. Why?

```
reg [3:0] d_0, d_1, d_2, d_3;
wire [3:0] ipt;
always @ (posedge clk) begin
    d_3<=d_2;
    d_2<=d_1;
    d_1<=d_0;
    d_0<=ipt;
end
```

Block

A block starts with `begin` and ends with `end`, just like the braces in C. While the braces in Verilog means combine two variables together like `{cout, sum}`.

Branch

Usually you will use `if` and `case`. Both need to be used **inside** `always` block.

The usage of `if` is quite similar to C, just replace the braces with `begin` and `end`.

The syntax of `case` is as following:

```
case (variable)
    value_1: code;
    value_2: begin
        code;
        code;
    end
    default: code;
endcase
```

Loop

In fact, `always` is **not** a loop. There are real loops in Verilog but we seldom use them. `always` is like "monitoring" a few values and when these values change, it responds.

```
always @ (Q or posedge clk or negedge clk) begin
    code;
end
```

The values an `always` block monitors can be the change of a `wire`, `reg` or the positive edge or negative edge of a signal.

Unwanted Latch Problem

If you do not set a default value or you do not specify every situation, a latch may appear. The reason is that Verilog will try to maintain the original value if you do not specify a new value. It can be fixed by specifying all possibilities.

Initialize Problem

- All usage related to time (`#100`) is not practical. If you want something controlled by time, you must specify a clock signal.
- All initial values of registers are not practical. Modern FPGA board may initialize a register as the initial value you give, but it's improper.

You cannot write `reg Q=0;`, but you should write

```
always @ (posedge clk or res) begin
    if(res==1)
        Q<=0;
    else
        Q<=Q+1;
end
```

Please remember, no matter you are writing Verilog or you are drawing a circuit, a reset or clear function is always necessary.

Simulation

When you try to simulate a module, you need to create a simulation module "outside" the test module. The simulation module should have no input or output. It should have registers to provide values to the test module and it should have wires to read the output of the test module.

```
module sim();
    reg ipt, clk;
    wire opt;
    test_module TM(ipt, opt);
    initial begin
        #0   ipt=0; clk=0;
        #100 ipt=1;
        #100 ipt=0;
        #200 ipt=1;
    end
    always #10 clk=~clk;
endmodule
```

Above is a typical simulation module. Remember that everything in `initial` block and everything related to `#time` is not practical.

Part 2. Counter

Asynchronous Counter

Delays will accumulate and cause timing issue. So, it's not very useful.

Synchronous Counter

State Table

The first thing you learned about synchronous counter is the state table, which will be quite useful when you start to learn about finite state machine (FSM). So, it's important to understand it. Here's a general state table for counter.

Present State			Next State			D Flip Flop Input		
Q2	Q1	Q0	Q2+	Q1+	Q0+	D2	D1	D0
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	0
0	1	0	0	1	1	0	1	1
0	1	1	1	0	0	1	0	0
1	0	0	1	0	1	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0

Please notice that you should first think about what the values of registers in next state, then you derive a proper input for registers. It can be implemented with D flip-flop, T flip-flop and JK flip-flop. Have a try.

Usually D flip-flops are used because you can just connect Q+ to D.

Combinational Circuit with Register

The next thing you need to do is to relate the register input with the register output. Then you can design a combinational circuit with above truth table. A typical structure of counter or FSM, is the combination of combinational circuit and registers (called state register).

For a counter implemented with D flip-flops, the Boolean equation for D is

$$D_0 = Q_0'$$
$$D_n = Q_n \oplus (Q_{n-1} \cdots Q_2 Q_1 Q_0)$$

Reset, Load, and CE

Usually the function of reset is to clear registers to zero. Reset function is always necessary. However, sometimes we want the registers to start from a particular value, that's why a load signal is needed. Basically, the structure is to connect the value you want to load and the count up value to a MUX and use load signal to select from them.

The principle of CE is similar. Please notice that the MUX closer to register has a higher priority. Because it can always reject former signal. You always need to know the priority to design such a counter. If the question doesn't state the priority, you should mention it by yourself. A priority table is a good choice.

reset	load	CE	Action on the Rising Clock Edge
1	X	X	Clear (Qn<=0)
0	1	X	Load (Qn<=Dn)
0	0	1	Count (Qn<=Qn+1)
0	0	0	Hold (No Change)

How to modify the table if we add a function called up_down, which has a lower priority than CE? How to implement it?

The CEO output basically is a sign that the counter has counted to its maximum value. If you only want to counter numbers, it seems that the CEO output is useless. But If you want to design a circuit that do some action after a counter reaches maximum value, or even a clock divider, the CEO output is important. It can also become the CE signal for next counter to combine two counters together.

Customize Counting Sequence

You can choose to activate the load signal when the counter counts to a particular value. Then, the particular value will be loaded to the counter.

Why we use load rather than reset? What about time diagram? What we need to do if we also want to load manually?

Clock Divider

Basically, a clock divider is a counter whose counting sequence has been modified. If you want to divide the clock by N, then your counter should have N states in total. For example, to divide a clock by 4, you need a 2-bit counter and it can be 0, 1, 2, 3, four states in total.

So, when designing a clock divider, the first thing you need to do is to decide the bit-width of the counter. Foe example, if I want to divide a clock signal by 1000, how many bits do I need?

Then, when the counter counts to the particular value, you need to reset it to zero by a load signal or a **synchronous** reset signal. In above example, when the counter counts to 999, it should trigger the load signal (1000 states in total). You should also output such signal as the divided clock.

How to generate such signal? What about time diagram? What's the problem here?

Synchronizer

Basically a D flip-flop. After the gated clock signal passes the synchronizer, the clock skew is reduced, minimized, and fixed. But **not** eliminated. What about time diagram?