ENCS 533 – Advanced Digital Design
Lecture 7
Processes in VHDL and Synchronous Logic

1 Introduction

In this lecture, we will look at how to describe *synchronous* logic, logic whose operation is synchronized to the edges of a clock signal. In order to do this, we will need to introduce a very important feature of VHDL, the *process*.

2 Concurrent VHDL

Here is the VHDL code that we saw previously, which implements the functionality of a full adder:

```
1  ARCHITECTURE simple OF fulladd IS
2  BEGIN
3     cout <= ( x AND y ) OR ( cin AND x ) OR ( y AND cin );
4     sum <= cin XOR x XOR y;
5  END ARCHITECTURE simple;
```

This style of code is often referred to as a *dataflow* description. The output signals are on the LHS of assignment statements, and the inputs are on the RHS. The outputs are connected to the inputs through arithmetic and/or logical connectives, such as AND, OR, XOR, +, -, etc. Let's think through what this piece of code means.

Statement 3 will run whenever a RHS value changes. So it runs when x, y or cin changes, i.e. when there is an *event* on x, y or cin. In the jargon of VHDL, statement 3 is *sensitive* to signals x, y, cin. Its *sensitivity list* is x, y, cin.

Similarly, statement 4 has a sensitivity list of x, y, cin. It will run when there is an event on any of the signals on its sensitivity list.

Statements 3 and 4 are *concurrent*, i.e. they are both active at the same time, and are triggered by an event on a signal on their sensitivity lists.

For concurrent code, the order of statements doesn't matter, so we could scramble the order of statements 3 and 4, and the effect would be the same:

```
1  ARCHITECTURE scrambled OF fulladd IS
2  BEGIN
3     sum <= cin XOR x XOR y;
4     cout <= ( x AND y ) OR ( cin AND x ) OR ( y AND cin );
5  END ARCHITECTURE scrambled;
```

This illustrates the "normal" behaviour of statements in VHDL, and for many circumstances it is fine. However, there exist some devices that are *not* sensitive to all of their inputs. Examples include latches and flip flops. For these devices, the "normal" behaviour of VHDL makes life difficult. It is useful to be able to explicitly say what we want the sensitivity list of a piece of code to be. This is done by a VHDL feature called a PROCESS.

3 Processes

A process looks like this

```
PROCESS ( sensitivity list )
BEGIN
  Statement 1;
  Statement 2;
  Statement3;
END PROCESS;
```

The way this works is as follows:

- The process waits until it is triggered by an event on one of the signals in its sensitivity list.
- When it is triggered it executes each of the statements in its body *sequentially*, i.e. one after the other, first statement 1, then statement 2, then statement 3. Note that this is quite different from "normal" VHDL, where statement order has no significance.

So here is our original full adder:

```
1  ARCHITECTURE simple OF fulladd IS
2  BEGIN
3     cout <= ( x AND y ) OR ( cin AND x ) OR ( y AND cin );
4     sum <= cin XOR x XOR y;
5  END ARCHITECTURE simple;
```

And here is a different version, which achieves exactly the same thing:

```
1  ARCHITECTURE with_proc OF fulladd IS
2  BEGIN
3     PROCESS (x, y, cin)
4     BEGIN
5           cout <= ( x AND y ) OR ( cin AND x ) OR ( y AND cin );
6     END PROCESS;
7
8     sum <= cin XOR x XOR y;
9  END ARCHITECTURE with_proc;
```

The process (lines 3-6) runs concurrently with the statement at line 8. Both are active at the same time, waiting for an event on their sensitivity list. For line 8, the sensitivity list is the signals on the RHS. For the process, the sensitivity list is the list explicitly declared in line 3. We can go even further, and use processes for both assignments:

```
1  ARCHITECTURE with_2proc OF fulladd IS
2  BEGIN
3     PROCESS (x, y, cin)
4     BEGIN
5           cout <= ( x AND y ) OR ( cin AND x ) OR ( y AND cin );
6     END PROCESS;
7
8     PROCESS (x, y, cin)
9     BEGIN
10          sum <= cin XOR x XOR y;
11    END PROCESS;
12 END ARCHITECTURE with_2proc;
```

The two processes (lines 3-6 and 8-11) run concurrently, i.e. both simultaneously monitor their sensitivity lists waiting to be triggered into life. Finally we could merge the two processes together to get this:

```
1  ARCHITECTURE all_in_one OF fulladd IS
2  BEGIN
3     PROCESS (x, y, cin)
4     BEGIN
5           cout <= ( x AND y ) OR ( cin AND x ) OR ( y AND cin );
6           sum <= cin XOR x XOR y;
7     END PROCESS;
8  END ARCHITECTURE all_in_one;
```

When there is an event on x, y or cin, the process runs and new values are queued for cout and sum.

3.1 Sequential VHDL

Sequential code "flows" from one line to the next, in blocks of code. This means that that we can build up complicated sequences of statements that build up a required behaviour over many lines. This is a very powerful way of doing things, and corresponds fairly closely to the way that the C programming language works.

In *concurrent* VHDL by contrast, each statement is completely independent of neighbouring statements, so each statement must be self contained. Whatever behaviour we are trying to describe for a particular signal has to be bunched together into a single statement of code. This can make description of certain types of behaviour difficult or even impossible.

There are many constructs of a language that *do* make sense if there is a flow of code from one line to the next, but *don't* make any sense at all if each statement is independent of its neighbours. So, in sequential VHDL there are many additional features of the language that we can use that are not available in concurrent VHDL.

So wrapping up a description in a process gives us the opportunity to write sequential code, which makes many additional features of the language available to us. Used wisely, this can be a good thing. However, it does give us the ability to write VHDL that cannot be synthesised to hardware, or may synthesise very inefficiently. By contrast, the dataflow approach more or less forces us to write code that will synthesise very nicely.

3.2 Sequential and concurrent conditionals

The syntax of the sequential IF block is shown below[1]

```
IF condition_1 THEN
      sequence of statements
ELSIF condition_2 THEN
      sequence of statements
ELSE
      sequence of statements
END IF;
```

[1] Notice that ELSIF (one word) is not the same thing as ELSE IF (two words)

Notice that this assumes a sequential flow of control from one statement to the next. So the IF block can only be used inside a process.

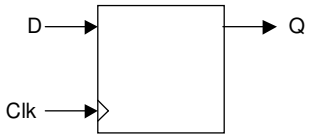
In concurrent code, each line stands alone and is triggered into life by a change on its RHS. So in order to achieve conditional assignment in a piece of concurrent code, we need a version of the IF statement that bundles all the functionality into one (possibly quite long) line of code. This is the WHEN statement.

a <= value1 WHEN condition1 ELSE value2 WHEN conditon2 ELSE value3;

4 The D-type flip-flop

The vast majority of hardware designs use synchronous logic, where system operation is driven by a clock signal distributed throughout the system. Each hardware block in the system will update its outputs at the rising edge[2] of the clock signal, and the outputs will then stay stable throughout the rest of the cycle until the next rising[3] edge.

The basic device that is used to accomplish this is the D-type flip flop.



Here is the ENTITY definition for this device[4]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

-- D-type flip-flop
ENTITY dff IS
    PORT ( d  : IN  STD_LOGIC;     -- Data input
           clk: IN  STD_LOGIC;     -- Clock input
           Q  : OUT STD_LOGIC);    -- Output gets new value on
                                   -- rising edge of clock
END ENTITY dff;
```

The behaviour of this device is as follows. When the clock is stable, Q simply holds its value constant. When a rising clock edge occurs, the output Q takes on the value that D has at the moment when the clock edge occurred. It then holds that value constant until the next rising clock edge occurs, at which time it updates itself again.

Note that the output q does *not* update its value whenever d changes. So to write this

[2] or sometimes falling edge
[3] or falling
[4] Note that in previous lectures the description have been written in poor style, with no comments, and bunched up as much as possible so that they don't take up too much room on the page. The style used here is better, with the inputs and outputs clearly spread out, and individually commented. In professional code there would be additional comments at the beginning of the definition giving information such as author name, date of completion of the first version, and revision history saying what each major revision was and when it was completed.

```
ARCHITECTURE wrong OF dff IS
BEGIN
    q <= d;
END ARCHITECTURE wrong;
```

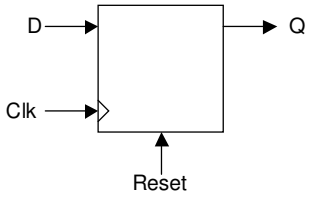
would give completely the wrong behaviour. Here is an architecture that correctly describes the behaviour of the device:

```
ARCHITECTURE correct OF dff IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF ( rising_edge(clk) ) THEN
            q <= d;
        END IF;
    END PROCESS;
END ARCHITECTURE correct;
```

Whenever *clk* changes its value, the process will run. However, clk might have changed due to a *falling* edge of the clock (which should not trigger an update to q) so we need to insert an IF statement which causes q to be updated only on the *rising* edge of clk. The *rising_edge* function is contained in the `STD_LOGIC_1164` package, and returns `TRUE` when clk has changed from 0 to 1 during the last *delta*.

4.1 The D-type with reset

When an electronic system is switched on, the contents of all flip-flops will go randomly to 1 or 0. *At power-up the contents of all flip-flops are garbage*. (It is this condition that the 'U' value in VHDL is designed to emulate.) It is useful to be able to put the flip-flops into a known state (usually '0', but sometimes '1'). This is done by means of a Reset signal.



The *Reset* signal may be synchronous, but is usually asynchronous. If the *Reset* is *synchronous*, then it is ignored until the rising edge of the clock. When the rising edge comes, if Reset='1' then q goes to '0'. If Reset='0', then the flip-flop exhibits normal behaviour, i.e. q<=d. This would be described like this:

```
ARCHITECTURE synch_reset OF dff IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF ( rising_edge(clk) ) THEN
            IF ( reset='1' ) THEN
                q <= '0';
            ELSE
                q <= d;
            END IF;
        END IF;
    END PROCESS;
END ARCHITECTURE synch_reset;
```

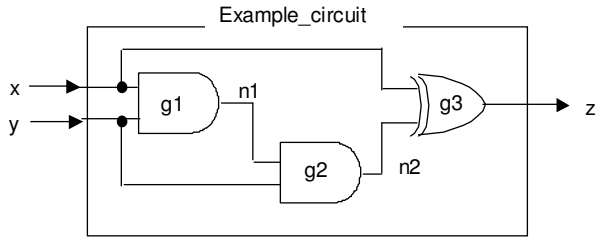
If the *Reset* is *asynchronous*, then it takes immediate effect, no matter what the clock is doing. This means that the flip-flop is always sensitive to its *Reset* input. This would be described like this:

```
ARCHITECTURE asynch_reset OF dff IS
BEGIN
    PROCESS (clk, reset)
    BEGIN
        IF ( reset='1' ) THEN
            q <= '0';
        ELSIF ( rising_edge(clk) ) THEN
            q <= d;
        END IF;
    END PROCESS;
END ARCHITECTURE asynch_reset;
```

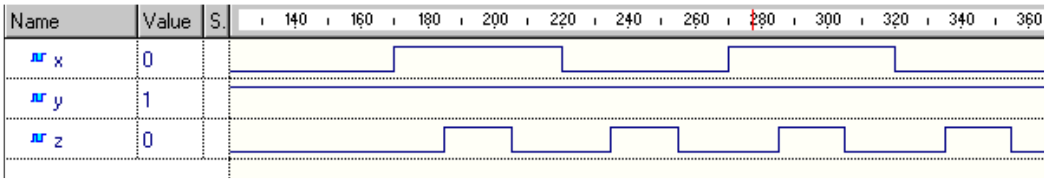
5 Synchronous logic

5.1 Applications of D-type flip-flops: cleaning up glitches

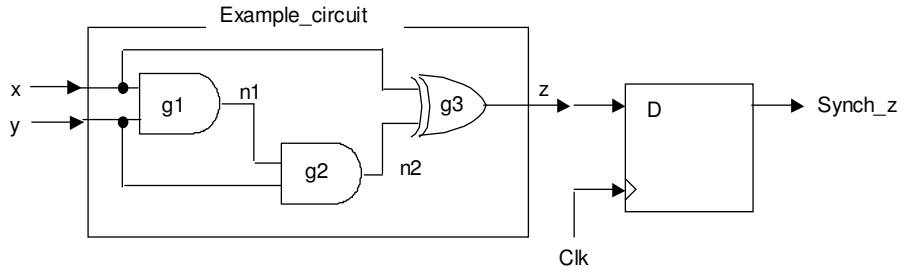
In lecture 4 we saw an example of a circuit which was glitchy.



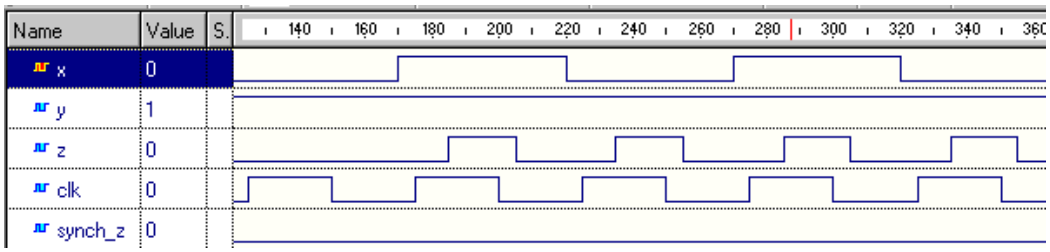
When x='0' and y='1' we expect z='0'.
When x='1' and y='1' we also expect z='0'.
Here is a simulation of the behaviour of the circuit as x transitions between '0' and '1'.



Whenever x transitions, z briefly goes to the "wrong" value. In many situations this can cause problems, so we would like to find a way of "cleaning up" the output z. A simple way to do this is like this:

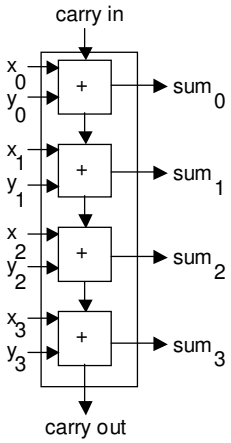


We attach a D-type flip-flop to the output. This will sample the value of z each time there is a rising edge of clk, and then hold this value steady over the whole of the remainder of the clock cycle. If we arrange for the clock edges to occur at a point when we know that all transient glitches have died away, then the output of the flip-flop will give us a "clean" signal.



5.2 Carry ripple in adders

Let's re-visit our structural description of the adder:

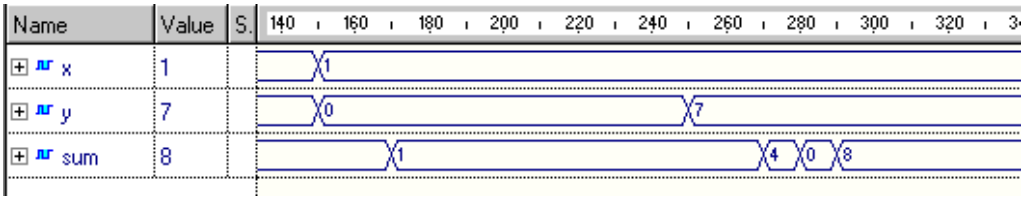


When we have a simple set of inputs, this behaves sensibly. Suppose x="0000" and y="0001". Then after a short delay as the input values move through the gate delays in the full adder, *sum* gets a new value of "0001".

Now suppose we have the values x="0001" and y="0111". After a brief delay *sum* becomes "0110". Then after a short while, fulladder 1 "notices" that fulladder 0 has generated a carry: its sum_1 output flips to '0', and its carry flips to '1'. So *sum*

becomes “0100”. Then after another short while, fulladder 2 notices that fulladder 1 generated a carry and *sum* becomes “0000”. Then after another while fullader 3 notices the carry that has just been generated by fulladder 2. Then *sum* becomes “1000”

Here is a simulation of how the adder behaves with the two different sets of inputs:

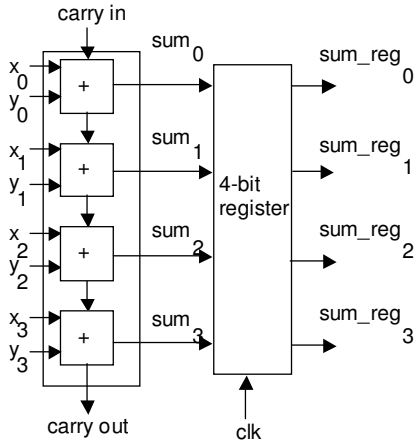


When x=1, y=0, the sum output goes quickly to the correct output, with no misbehaviour en route. However, when x=1 and y=7, we have a series of outputs which are garbage, and the sum takes a long time to settle to the correct value.

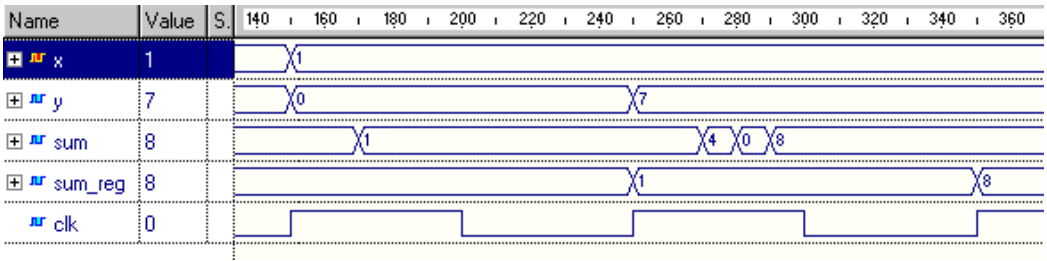
This effect is called *carry ripple*. For our little four bit adder, the problem was awkward enough. But realistic adders are more likely to be 16, 32 or even 64 bits in length, so the carry may have to ripple down a very long path. This can cause a long delay period during which the outputs of the adder are garbage.

5.3 The synchronous adder

The normal way to fix this is to register the outputs:



A group of D-type flip flops all controlled by the same clock signal is called a *register*. This is what the output of the register looks like:



The *registered sum* will be updated at the each rising edge of the clock signal. If the clock is slow enough that *sum* has completely settled before the next clock edge arrives, then the r*egistered sum* is a cleaned up version of *sum*. (But notice that if we ran the clock too fast, the rising edge would come during the period in which sum is garbage, and the registered output would therefore be wrong).

One very important point to notice here is that the output is acquiring its value one clock cycle *after* the corresponding inputs. So the output goes to 1 the cycle after the inputs were x=0, y=1. Similarly, the output goes to 8 one cycle after x=7, y=1. This is often a source of confusion, and you should make sure you understand why this happens.

5.4 VHDL description of the synchronous adder

So far we have been looking at a structural description of the adder. But usually humans don't produce structural code. They write behavioural code like this:

```
ARCHITECTURE behavioural OF adder IS
BEGIN
    sum <= a + b;
END ARCHITECTURE behavioural;
```

and leave it to the synthesis tool to produce the structural gate-level description of the design. Now if we are looking for a synchronous adder, the above description isn't good enough. There is nothing to tell the synthesis tool that we want the addition to be synchronized to a clock signal.

An obvious way to describe the adder with registered output is to instantiate 4 flip-flops at the output of the adder:

```
ARCHITECTURE synchronous OF adder IS
    SIGNAL clk: STD_LOGIC;
    SIGNAL reg_sum: STD_LOGIC_VECTOR ( 3 DOWNTO 0 );
BEGIN
    sum <= a + b;
    g1: ENTITY work.dff(correct)
        PORT MAP ( d=>sum(0), clk=>clk, q=>reg_sum(0) );
    g2: ENTITY work.dff(correct)
        PORT MAP ( d=>sum(1), clk=>clk, q=>reg_sum(1) );
    g3: ENTITY work.dff(correct)
        PORT MAP ( d=>sum(1), clk=>clk, q=>reg_sum(2) );
    g4: ENTITY work.dff(correct)
        PORT MAP ( d=>sum(1), clk=>clk, q=>reg_sum(3) );END
ARCHITECTURE synchronous;
```

However, this is pretty painful (even more so for a 32-bit adder). Instantiating lots of small pieces of hardware is a structural way to do things, and we normally don't want humans to operate in this way. This is much better:

```
ARCHITECTURE synchronous OF adder IS
    SIGNAL clk: STD_LOGIC;
    SIGNAL reg_sum: STD_LOGIC_VECTOR ( 3 DOWNTO 0 );
BEGIN
    sum <= a + b;

    process (clk)
    begin
        if ( rising_edge(clk ) ) then
            reg_sum <= sum;
        end if;
    end process;
END ARCHITECTURE synchronous;
```

Now we have made in clear to the synthesis tool that we want a registered version of sum to be created, synchronized to the rising edge of the clock. It is then up to the synthesis tool to figure out how to build a circuit that achieves this behaviour.

This description is better still:

```
ARCHITECTURE synchronous OF adder IS
    SIGNAL clk: STD_LOGIC;
    SIGNAL reg_sum: STD_LOGIC_VECTOR ( 3 DOWNTO 0 );
BEGIN
    process (clk)
    begin
        if ( rising_edge(clk) ) then
            reg_sum <= a + b;
        end if;
    end process;
END ARCHITECTURE synchronous;
```

If we want a reset signal, that can asynchronously reset the adder output to zero, this is achieved in a similar fashion:

```
ARCHITECTURE synchronous OF adder IS
    SIGNAL clk: STD_LOGIC;
    SIGNAL reg_sum: STD_LOGIC_VECTOR ( 3 DOWNTO 0 );
BEGIN
    process (clk, reset)
    begin
        IF ( reset='1' ) THEN
            reg_sum <= "0000";
        ELSIF ( rising_edge(clk) ) THEN
            reg_sum <= a + b;
        end if;
    end process;
END ARCHITECTURE synchronous;
```

This style of coding is called *register transfer level* coding. We are using dataflow statements, but wrapping them up in processes triggered by the clock (and possible some reset or enable signals) in order to make it clear on what clock cycle the outputs should assume their values. Notice that in RTL we are simply defining the behaviour we want (in this example reg_sum <= a+b on a rising edge of the clock). We are leaving it entirely up to the synthesis tool to infer what configuration of registers will be needed to give us this behaviour.

Summary

In this lecture we have looked at how to use clocked *processes* to build *register transfer level* (RTL) descriptions. These are behavioural descriptions that make it clear on which clock edge the outputs must assume their value. These descriptions can then be synthesised into synchronous circuits using the appropriate configuration of registers by a synthesis tool.

You should now know...

How to construct behavioural descriptions of synchronous circuits

The meaning of the following:

- Sensitivity list
- Sequential execution
- Concurrent execution
- Dataflow description
- Register transfer level (RTL) description
- Process
- Carry ripple