# ENCS 533 - Advanced Digital Design
# Lecture 8
# Concurrent and Sequential VHDL

## 1 Introduction

This lecture will introduce more features of VHDL, with an emphasis on flow of control and synchronisation in time.

## 2 Concurrent execution

We have seen that the normal way that statements are processed in VHDL is concurrent. So, for example, in the following code

```
a <= b;
b <= c;
c <= '1';
```

all statements are active simultaneously. A statement is triggered into life when a variable on its right hand side changes. Statements that have no variables on their right hand side are executed immediately.

So for the above code, the statement c<='1' is executed first (because the RHS is a literal), then b<=c is executed (because c has a new value) then a<=b is executed (because b has a new value).

The order of the statements is completely irrelevant, so the above code would have exactly the same function as this:

```
c <= '1';
b <= c;
a <= b;
```

### 2.1 Signal assignment always has an associated delay

It is very important to realise that when we make a signal assignment, there is always an associated delay. If we write

```
a <= b AFTER 10 NS;
```

then obviously a will take on its new value 10 ns after this statement is executed. If we write

```
a <= b;
```

then a does *not* get its new value immediately. VHDL will insert a delay, and will interpret this statement as

```
a <= b AFTER delta;
```

Delta is an infinitesimally small delay, the smallest time step that the simulator can cope with.

So, when VHDL runs the following code

```
a <= b;
b <= c;
c <= '1';
```

this is what will happen

| time | A | b | c |
|---|---|---|---|
| 0 | U | U | U |
| 0 + delta | U | U | 1 |
| 0 + 2 × delta | U | 1 | 1 |
| 0 + 3 × delta | 1 | 1 | 1 |

At the beginning (time zero) a, b and c all have the value U (uninitialised). At time zero, the statement c <= '1' will run. The change to c will not take place immediately. Instead, a transition (c gets 1 at time delta) is placed onto the event queue.

The simulator has run out of things to do at time zero, so it increments the time variable to the time of the next scheduled event, i.e. delta. At this point c gets its new value of 1. This event on c triggers the statement b <= c to execute. This places a transition (b gets 1 at time 2 × delta) onto the event queue.

And so on. The important things to notice here is that there *must* be a delay before a, b and c get their new values. This delay has very important implications.

### 2.3 This delay protects us from logical inconsistency

Suppose there wasn't a delay in signal assignment. Suppose that a signal received its new right hand side value *instantly*. How could we interpret this piece of code:

```
a <= b AFTER 0;
b <= NOT a AFTER 0;
```

There is no logically consistent interpretation. By contrast, if we have this

```
a <= b AFTER delta;
b <= NOT a AFTER delta;
```

there is no difficulty at all. If we assume a starting state of a=1, b=1 then the following will happen

| time | A | B |
|---|---|---|
| 0 | 1 | 1 |
| 0 + delta | 1 | 0 |
| 0 + 2 × delta | 0 | 0 |
| 0 + 3 × delta | 0 | 1 |
| 0 + 4 × delta | 1 | 1 |
| 0 + 5 × delta | 1 | 0 |

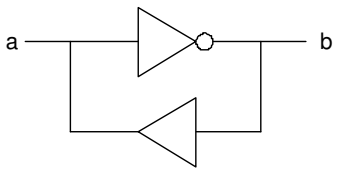
So at time zero, a <= b executes, and places a transition (a gets 1 at time delta) onto the event queue; also and b <= NOT a executes, and places a transition (b gets 0 at time delta) onto the event queue. It is only at time delta that a and b will get their new values.

So in response to this

```
a <= b;
b <= NOT a;
```

a and b will both oscillate between 0 and 1 with a time period of 4×delta[1].

So the delay has rescued us from the logical intractability of our code. Also, if we consider what the code means in real life, it is a description of a piece of hardware that looks like this:




This is a ring oscillator. a and b would both go into oscillation with a time period equal to twice the combined propagation delay of the two logic gates.

So the delay that VHDL enforces on signals is also ensuring that our code behaves in a way that realistically represents the behaviour of hardware.

## 3 Sequential execution

We can, if we want, force VHDL to interpret code sequentially, just as it would in a programming language like C. In order to do this, we wrap the code inside a process, like this[2]

```
PROCESS
BEGIN
    c <= '1';
    b <= c;
    a <= b;
END PROCESS;
```

In this case the statements are executed *in sequence one after the other*, so c<='1' happens first, b<=c happens next then a<=b happens. Although this may feel familiar, since you are used programming languages like C, the consequences of sequential execution are quite odd. Let's compare VHDL with C:

---

[1] Note that we had to assume that a and b had somehow been initialised. If they hadn't, then the result of running our code would have been that a and b would be both stuck at U for ever more.

[2] This process is given as an illustrative example. It would actually be thrown out by a VHDL compiler because it doesn't contain sufficient information about when the process should run. We'll return to this issue in section 5.

| Assume that initially a=b=c=0 | Assume that initially a=b=c=0 |
|---|---|
| Sequential VHDL:<br><br>`SIGNAL a,b,c: STD_LOGIC;`<br>`BEGIN`<br>`  c <= '1';`<br>`  b <= c;`<br>`  a <= b;`<br>`END;` | C programming language:<br><br>`int a,b,c;`<br>`{`<br>`  c=1;`<br>`  b=c;`<br>`  a=b;`<br>`}` |

**In C**

- c=1 runs first, so c is now 1
- b=c runs second, so b is now 1
- a=b runs third, so a is now 1

**In sequential VHDL**

- At time *now*
  c<='1' runs. It places a transition onto the event queue: (c becomes '1' at time delta later than *now*).

- We are still at time *now*, and
  b<=c runs.
  c has *not yet got its new value*, and is still at its old value of '0'.
  A transition is placed into the event queue (b becomes '0' at time delta later than *now*)

- We are still at time *now*, and
  a<=b runs.
  b is still at its old value of '0'.
  A transition is placed into the event queue (a becomes 0 at time delta later than *now*)

- VHDL runs at out statements to execute at the current time, so the time is incremented to the time of the next event on the queue, i.e. delta later.

- The transitions take effect, so a=0, b=0 and c=1.

*Because signal assignments cannot take immediate effect, the outcome of the VHDL code is different from that of the C code.*

This may seem startling, but as we will see later on, this behaviour is a realistic reflection of the behaviour of real hardware, and it is important that VHDL should behave like this.

### 3.1 Variables

As we implied in lecture 1, a feature of a good hardware description language is that it can do anything that a good programming language can. This is especially important when writing algorithmic specifications, but is often useful in other situations.

But we have just seen that the behaviour of sequential VHDL is radically different from the behaviour of C due to the delayed assignment behaviour of signals. Sometimes this is awkward, and we would like to write VHDL that behaves just like C.

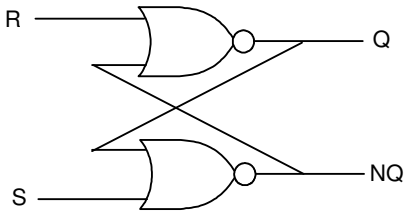
In order to achieve this VHDL has *variables*. These are like signals, with the exception that assignments take effect *immediately*. As a result, variables in VHDL behave exactly like variables in C. So the following two pieces of code are identical in their effect

| Assume that initially a=b=c=0 | Assume that initially a=b=c=0 |
|---|---|
| Sequential VHDL with variables:<br><br>VARIABLE a,b,c: `STD_LOGIC`;<br>BEGIN<br>  c := '1';<br>  b := c;<br>  a := b;<br>END; | C programming language:<br><br>int a,b,c;<br>{<br>  c=1;<br>  b=c;<br>  a=b;<br>} |

In order to remind us that the assignment is instantaneous, variables use the assignment operator := instead of the <= operator used by signals. Variable can only exist in sequential code. That means that they can only exist inside processes.

## 5 An example

To illustrate the different issues involved in structural, dataflow and algorithmic code, it will be useful to consider an example. Our example will be an RS latch, as shown below.



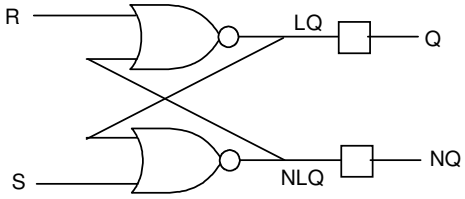

The entity for this is

```
ENTITY rs IS
      PORT ( s, r : IN STD_LOGIC; q, nq : OUT STD_LOGIC);
END ENTITY rs;
```

Before we go any further, there is one subtlety that must be addressed. When we declare a signal as being of mode IN or OUT we restrict the operations that can legally be carried out on them:

- it is illegal to read a value from a signal of mode OUT
- it is illegal to read a write from a signal of mode IN

In order to build our latch, we need to feed q and nq to the inputs of the NOR gates. But q and nq are of mode OUT, so it is illegal to use them in this way. To get round this problem, we have to invent a couple of internal nodes, which are used for feedback.




The values of these internal nodes are then short circuited to the outputs.

The behaviour of the RS latch is as follows:

| R | S | Q | NQ |
|---|---|---|---|
| 0 | 0 | Retains its previous value | Retains its previous value |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |

S is the *set* input, which sets Q to 1. R is the *reset* input that resets Q to 0. When neither *set* or *reset* is asserted (i.e. R=0, S=0) the latch remembers the last value of Q. NQ is the complement to Q. Normally whenever Q=0, NQ=1 and vice versa. R=1 and S=1 at the same time is a condition that should not occur in normal operation (it means that we are trying to set and reset the latch at the same time, which doesn't make sense). When this happens Q and NQ will both go to zero, which means that the complementarity of Q and NQ has been lost.

### 5.1 A dataflow architecture

A suitable dataflow implementation of the RS latch might be

```
ARCHITECTURE dataflow OF rs IS
      SIGNAL lq: STD_LOGIC:= '1';
      SIGNAL lnq : STD_LOGIC:= '0';
BEGIN
      lq <= '1' WHEN s='1' AND r='0'
            ELSE '0' WHEN r='1' and s='0'
            ELSE '0' WHEN r='1' and s='1'
            ELSE lq;
      lnq <= '0' WHEN s='1' AND r='0'
            ELSE '1' WHEN r='1' and s='0'
            ELSE '0' WHEN r='1' and s='1'
            ELSE lq;
      q <= lq;
      nq <= lnq;
END ARCHITECTURE dataflow;
```

### 5.2 A structural architecture

A structural version of this would be

```
LIBRARY gates;
ARCHITECTURE structural OF rs IS
      COMPONENT nor2
            PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
      END COMPONENT;
      FOR ALL: nor2 USE ENTITY gates.nor2(dataflow);
      SIGNAL lq: STD_LOGIC:= '1';
      SIGNAL lnq : STD_LOGIC:= '0';
BEGIN
      g1: nor2 PORT MAP ( r, lnq, lq );
      g2: nor2 PORT MAP ( s, lq, nq );
      q <= lq;
      nq <= lnq;
END ARCHITECTURE dataflow;
```

The above listing assumes that there exists a library called gates, which contains a gate nor2 with a dataflow architecture.

### 5.3 A sequential description

Suppose we use a sequential behavioural description. This would be written with a sequential piece of code, and would function just like a piece of C code. The behaviour of the latch would look like this

```
IF r='1' AND s='0' THEN
      q <= '1';
      nq <= '0';
ELSIF r='0' AND s='1' THEN
      q <= '0';
      nq <= '1';
END IF;
```

But when would this sequence of statements be executed? Sequential statements are executed in sequence, one at a time. The above code would be executed when its turn comes, after the preceding statements have been executed. The code would *not* be triggered by a change in r or s.

This is not what we want at all. Although we want this piece of code to be executed sequentially, we want its execution to be triggered by an event on r or s.

### 5.4 Processes

The triggering of sequential code is handled by means of a sensitivity list given to the process. This is illustrated below.

```
ARCHITECTURE sequential OF rs IS
BEGIN
      PROCESS ( r, s )
      BEGIN
            IF r='1' AND s='0' THEN
                  q<= '1';
                  nq <= '0';
            ELSIF r='0' AND s='1' THEN
                  q <= '0';
                  nq <= '1';
```

```
            ELSIF r='1' AND s='1' THEN
                  q <= '0';
                  nq <= '0';
            END IF;
      END PROCESS;
END ARCHITECTURE sequential;
```

The PROCESS statement has a sensitivity list ( r, s ). Any change on any signal in the sensitivity list triggers the execution of the process. Within the process, execution is strictly sequential. The process runs until it reaches the END PROCESS. It will then be dormant until another event occurs on its sensitivity list.

### 5.5 Processes without sensitivity lists: the WAIT statement

A process does not have to have a sensitivity list. There are other ways of controlling its execution. A PROCESS without a sensitivity list runs automatically from the start of the simulation. A process runs until it reaches the END statement, and then immediately resumes at its BEGIN. So

```
PROCESS
BEGIN
      Statements go here
END PROCESS;
```

Would run forever in infinite loop, which will rarely be useful. However, we can tell a process to suspend its operation until a condition becomes true. This is done by means of a WAIT statement. There are four types of WAIT statement:

`WAIT FOR` a certain amount of time
`WAIT ON` sensitivity list
`WAIT UNTIL` some Boolean condition is satisfied
`WAIT`

So our above example could have been written like this

```
ARCHITECTURE sequential OF rs IS
BEGIN
      PROCESS
      BEGIN
            IF r='1' AND s='0' THEN
                  q<= '1';
                  nq <= '0';
            ELSIF r='0' AND s='1' THEN
                  q <= '0';
                  nq <= '1';
            ELSIF r='1' AND s='1' THEN
                  q <= '0';
                  nq <= '0';
            END IF;
            WAIT ON r, s;
      END PROCESS;
END ARCHITECTURE sequential;
```

The process starts running immediately. When it gets to the WAIT statement, it is suspended until an event occurs on r or s. The process then continues, reaches the END PROCESS, at which point it immediately returns to its BEGIN statement,

### 5.6 Using variables instead

As an illustration of the use of variables, we let's rewrite our algorithmic description with variables.

```
ARCHITECTURE sequential OF rs IS
BEGIN
      PROCESS ( r, s )
            VARIABLE lq, lnq: STD_LOGIC;
      BEGIN
            IF r='1' AND s='0' THEN
                  lq := '1';
                  lnq := '0';
            ELSIF r='0' AND s='1' THEN
                  lq := '0';
                  lnq := '1';
            ELSIF r='1' AND s='1' THEN
                  lq := '0';
                  lnq := '0';
            END IF;
            q <= lq;
            nq <= lnq;
      END PROCESS;
END ARCHITECTURE sequential;
```

Variables (in this case lq and lnq) cannot exist outside a process; they must be declared as local to a particular process. In order that the results of the process can be seen externally, the values of the variables are copied to the signals q and nq;

### 5.7 Sequential and concurrent conditionals

The syntax of the IF block is shown below

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

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;

Similar issues arise for CASE blocks.

```
INTEGER n;
CASE n IS
      WHEN  0     =>    z <= '0';
      WHEN  1     =>    z <= '1';
      WHEN  OTHERS      => z <= NOT z;
END CASE;
```

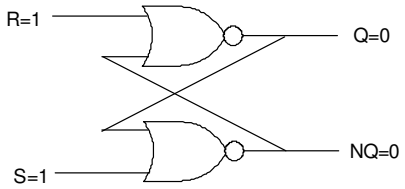
The CASE is spread across several statements (count the semicolons); it is assuming sequential flow of control from one statement to the next. So a CASE block can exist only inside a PROCESS. Within concurrent code, a different syntax is used, which bundles all the functionality into one statement:

```
WITH n SELECT
      z <=  '0' WHEN 0,
            '1' WHEN 1,
            NOT z WHEN OTHERS;
```

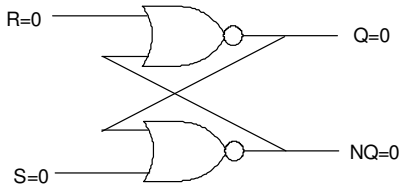
The case statement must exhaustively list **all** possible values for the its argument. This is facilitated by using the OTHERS clause, to indicate everything that has not been explicitly listed.

## 6 Failure of the RS latch

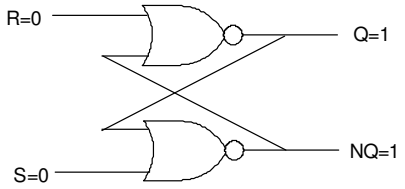
To complete our treatment of the RS latch, we will look at the failure mode of the device. Suppose we have the situation R=1, S=1 simultaneously. This is what happens.



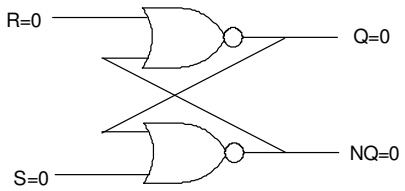

Now if R then changes to 0 and S stays at 1, then Q will become 0. If S changes to 0 and R stays at 1, then Q will become 1. But what happens if S and R both change to 0 *simultaneously*.




A NOR gate with 0 at both inputs will output a 1, so we get




A NOR gate with a 1 at one input will output a zero




And so on. *The device has gone into oscillation*[3].

It is usually the case with memory elements, such as latches and flip-flops, that if changes in their control signals are widely separated they will function correctly. But if changes in the control signals occur too close together, something bizarre happens. For the RS latch, the required time difference between changes in the R and S inputs is called the *recovery time*. If changes in R and S are too close together, this is called a *recovery time violation*.

This sort of failure can also happen in the D-type flip-flop. If the D input changes too close to a change in the clock input, the output becomes unpredictable and may oscillate. This is referred to as a *set-up time violation* if D is unstable *before* the controlling clock edge, or a *hold-time violation* if D is unstable *after* the clock edge.

It is an instructive exercise to go through all the VHDL descriptions that we have developed for the RS latch to see why they do correctly model the oscillatory behaviour caused by a recovery time violation. It is essentially due to the fact that signals cannot be assigned with zero delay. If we could assign a signal with zero delay, or if a variable could live outside a process, then our descriptions would not be able to capture the correct behaviour.

## 8 Summary

In this lecture we have looked at the following key issues

- Sequential and concurrent flow of control
- Synchronisation of sequential and concurrent code
- Sensitivity lists to control when pieces of code should be executed

[3] This oscillation is one of the failure modes of real life RS latches. Alternatively, due to analogue effects, the outputs may instead go to an invalid logic level for a random time interval, and then flip randomly out into a 1 or 0. This condition is called *metastability*.