NTNU
Department of electronics and Telecommunications
2016-08-28

TFE4141 Design of Digital Systems 1

Time and delta-delay in VHDL

Events and transactions

When a signal changes its value, this is called an ***event***. Assume that a signal **A** changes value from ’0’ to ’1’. If the signal name (here: **A**) appears on the right hand side of an expression, e.g.:

```
Y <= and (A, B) after 2 ns;
Z <= or (A, B) after 3 ns;
```

it triggers a ***transaction***: The expression with a change (here: **A**) on the right hand side, is executed. Let us assume that **B** = ‘1’, and that **B** is not changed at the same time. Consequently **Y** will also change its value ’0’ to ’1’, while Z will remain unchanged, and equal to ‘1’.

Since the expression states that it takes 2 **ns** (nanoseconds) before this change is valid at the output, an event will be put in an ***event queue*** at the right point in time (current time + 2 ns). If there were more expressions with A on the right hand side, they would be executed and values would be calculated for all signals on the left hand sides. If such a calculation gave a new value different from the current value, the new value would be put in the event queue at the designated point in time.

You probably by now understand that a dynamic event queue is generated when the simulation proceeds. The best is to sort this queue with increasing time, as this saves computation time when searching for the next active event.

Straight forward!

But how should we handle the following:

```
Y <= and (A, B);
Z <= or (A, B);
X <= xor (Y, Z);
```

In this case no time delay has been specified. The **default** understanding would be a delay of ***zero***. *In this case*, this traditional thinking would not be a problem: When **A** has changed its value, the new value of **Y** will be calculated, then **Z** will be calculated without changing its value, and finally **X** will be calculated resulting in a new value.

However, in VHDL we want the end result to be the same no matter in what order the code lines (expressions) are written. This is in a sense equivalent with the fact that it does not matter in what order you draw the gates in a circuit schematic.

What if we had written the code in the following order:

```
Z <= or (A, B);
X <= xor (Y, Z);
Y <= and (A, B);
```

With the same sequential execution of the code, **Z** and **X** would remain unchanged, while **Y** would receive its new value. Obviously, we cannot allow the end result to depend on the (arbitrary) order in which the code is written.

The solution is to introduce a so-called ***delta-delay*** (Δ), and wait a time period equal to delta before we update the right hand sides of the expressions above. The new value of **X** will then be updated after two delta-cycles. First, the **Z** and **Y** expressions will be evaluated, and after one delta-cycle the result appears at the output of **Y** (**Z** does not change). This will result in an evaluation of the **X** expression since **Z** appears on its right hand side. After another delta-cycle, the **X** value is updated. In this case the order in which the code lines are written does not matter since we anyhow will end up with the correct values for all signals.

It is important to recognize that a delta-cycle really does not have any real length of time, but should be understood as an infinitesimal short time. It is still handy to show it with a certain length to be able to demonstrate its use, and when we want to see what happens step by step in a simulation. For practical reasons, most simulators use 1 fs (10^{-15}) as the length of a delta-cycle, this then also being the smallest time resolution.

Concurrency

A challenge when simulating VHDL-code on a sequential computer is to handle concurrent events (Norwegian: samtidige hendelser) correctly.

Let us first have a look at the concepts of a ***process*** and a ***signal***. Note that:

1. Each process is executed continuously, concurrently with all other processes. A process is activated each time a change occurs in one (or more) of the signals in its ***sensitivity list***. The process is suspended (Norwegian: passivisert) when all transactions generated by this signal change (i.e., event) have been handled.
2. The mechanism used to track all events and make them appear at the right time, is an event queue. A special monitor is used to place future changes into the ***event queue***, at the right moment in time.

A process can be declared explicitly, or established implicitly. These two ways are equivalent:

Explicit:

```
AND: process (A, B) is
begin
  Y <= and (A, B) after 2 ns;
end process AND;
```

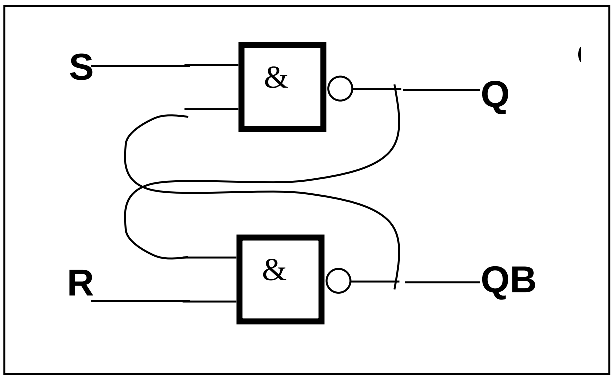
Implicit:

```
Y <= and (A, B) after 2 ns;
```

In both cases the processes are activated by a change in **A** or **B**. The explicit version has a sensitivity list, where **A** and **B** are listed. If such a list is not included, all signals appearing on the right hand side of the code inside the process will be part of the sensitivity list. If we do have a sensitivity list like here, but then forget to include, e.g., **B** in the list, then **Y** will not be updated when **B** is changed, and we would not get the functionality we wanted.

With a sensitivity list it is possible to ignore changes that cannot give any chances to the outputs. This is typically the case when we model synchronous logic, where we, e.g., have flip-flops triggered by a clock input. Changes to a flip-flops data input can be ignored as long as the clock input does not change.

Example of delta-delay: model of an SR-latch



Given the **SR-latch** in the Figure above (European style NAND-gate symbols). We can model this with the following two concurrent (implicit) processes written inside an architecture (not shown here):

```
Q <= S nand QB; -- no delay.
QB <= R nand Q; -- no delay.
```

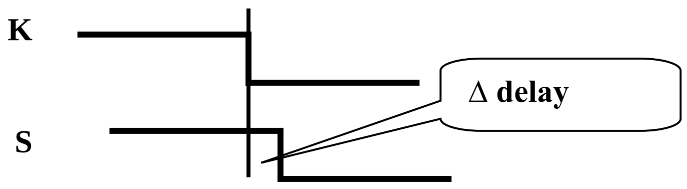
Assume the following stimuli:

```
S <= ‘1’ ; R <= ‘0’ ;
wait for 10 ns ;
S <= ‘0’ ;              -- after 10 ns.
wait for 10 ns ;
S <= ‘1’ ; R <= ‘1’ ; -- after 20 ns.
wait for 10 ns ;
R <= ‘0’ ;              -- after 30 ns.
wait for 10 ns ;
R <= ‘1’ ;
```

We have mentioned that a delta-delay is used to handle new events. You can look at it this way: A signal with a source **K** and a sink **S** gives the following VHDL-statement (code line):

S <= K; -- no delay explicitly modeled.

Here a change in **K** will appear at **S** after a delta-delay.



We assume that a signal type which includes the logic values **U, X, 0, 1** is used. It is possible to declare you own logic type, and ours can then look like this:

```
TYPE my_logic IS (  'U',  -- Uninitialised
                    'X',  -- Forcing Unknown
                    '0',   -- Forcing 0
                    '1',   -- Forcing 1);
```

When simulation starts, a signal that is not initialized to a specific value will be given the first value in the list, in this case '**U**' – (**U**ninitialized). Later on the signal will typically be assigned the values '**X**', '**1**' or '**0**'.

The following resulting values from transactions can be assumed:

nand (0,U) => 1
nand (0,X) => 1
nand (1,U) => X
nand (1,X) => X
nand (1,1) => 0
nand (1,0) => 1

Remember that we have established two processes, **Q** and **QB**. We use the notation Q+ and QB+ for the results from the right hand side of the expressions before they have been transferred to the signals on the left hand side. We write unchanged values in parenthesis. The resulting time diagram will be as follows:

Time	**S**	**R**	**Q**	**QB**	**Q+**	**QB+**	ACTIVATED PROCESS
0	U	U	U	U		?	Initializing. **S,R** activated
0+Δ	1	0			X	1	**Q,QB**
0+2Δ			X	1	0	(1)	**Q,QB**
0+3Δ			0	(1)	(0)		**QB**
0+4Δ			(0)				No new change. QB suspended.
10							**S** activated, visible next cycle.
10+Δ	0				1		**Q**
10+2Δ			1			(1)	**QB**
20							**S, R** activated
20+Δ	1	1			0	0	**Q,QB**
20+2Δ			0	0	1	1	**Q,QB**
20+3Δ			1	1	0	0	**Q,QB**
20+4Δ			0	0	1	1	**Q,QB**
0+**n**Δ							**Q,QB oscillates!**
30		0					**Will never get here, because the time will not get past 20 ns!**

Note that we get oscillations in the feedback loop with zero time delay! The simulation will terminate by a mechanism in the simulator (typically max 1000 delta-cycles at each point in time).

How can we model the oscillation in a more realistic way?