The Mystery Delta Cycle

Petter Källström

Many students have heard about the so called delta cycle in VHDL simulations, but have no really clue of what it is. Here is an informal and example based explanation.

A delta cycle is the 0 ns delay that differs an assignment (e.g. *a*<=*b*) of a signal (*a*), from the assignment of the signal it depends on (*b*) in a simulation. **The delta cycle is only a simulation term. It does not apply to hardware realization.**

A short example. Consider the concurrent VHDL statements:

```
x <= '0', '1' after 10 ns;
y <= not x;
```

Here, *x* will get the value '0', and then '1' after 10 ns. *y* will be '1' and then change to '0' one delta cycle after *x* changes to '1'. Since a delta cycle is 0 ns long, they appear to change in the same time in a simulation waveform. With a third statement;

```
z <= x and y;
```

z will get the value '1' one delta cycle after *x* gets '1' (in the same time as *y* gets '0'), and then back to '0', one delta cycle later. In the waveform this is shown as just a vertical spike at the time 10 ns.

Some simulation rules:

```
process(clk) begin
  -- with sensitivity list
  -- execute on clk event
end process;
process begin
  -- no sensitivity list
end process;
```

- A process is executed sequentially. It has its own "program pointer" (pointing out the current executing statement).
- A process with sensitivity list, i.e. process(<sensitivity list>), is executed once every time there is a change in the sensitivity list (including once at time 0 ns). Such a process must not have any wait statements.
- A process without sensitivity list is executed at time 0 ns. The process starts over again as soon as it comes to "end process;", and can run any number of times – i.e. it can hang the simulation in an infinity loop. Such a process should have at least one wait statement (or it will never end).
- A concurrent statement is executed every time there is a change on any signal that the statement depends on.
- The simulator runs the following simulation cycle over and over again:
    1. Find next time point with an action (initially: time 0 ns).
    2. Update all signals that should be updated (initially: initial value). These signals gets the boolean attribute 'event = true. All other signals gets the attribute 'event = false. Initially, all values get 'event=false.
    3. Execute all processes and concurrent statements that depends on the signals (initially: execute all).
        - Each signal assignment is noted into an assignment table. The signals are *not* immediately updated.
        - The assignment table contains information about signals, values, and update.
    4. As soon as all concurrent statements and processes are done (processes with sensitivity list) or reached a wait statement (processes without sensitivity list only), the simulation cycle ends.
        - If any signal was assigned *without* the "after <time>", the next time point with an action is "now". I.e., the simulation time is not updated in the next cycle. **This is a delta cycle.**
        - There might not be any processes or concurrent statements to execute, since no one depends on the updated signal. Then the simulation cycle immediately ends. The next simulation cycle will then have a new simulation time.
- A process (without sensitivity list) may wait for some time to elapse (e.g. using "wait for 10 ns;"). Then the continuation of this process is considered as an action in step 1. above. The process is of course executed in step 3.
- A process (without sensitivity list) may wait for a condition of a signal (e.g. "wait until rstn = '1';"). The wait condition is evaluated in the beginning of step 3 above (if a signal in the condition has changed). As a result, there must be a wait of at least one delta cycle. That is, "wait until clk='1';" is equivalent to "wait until rising_edge(clk);"

The delay between two signals, during a delta cycle, is sometimes called a delta delay.

A bigger example.

Assume the VHDL code below (*clk* is initialized to '1'. rising_edge(*x*) is equivalent to *x*'event AND *x*='1'):

```
rstn <= '0', '1' after 30 ns; -- (c1)
clk <= not clk after 10 ns;   -- (c2)
z <= x xor y;                 -- (c3)
f <= z after 15 ns;           -- (c4)
c2 <= clk;                    -- (c5)
p1 : process(clk, rstn) begin
     if rstn = '0' then
          x <= '0'; y <= '0';
     elsif rising_edge(clk) then
          y <= x;
          x <= not y;
          cb <= clk;
          c2b <= c2;
     end if;
end process;
p2 : process begin
     wait until y = '0';  -- (w1)
     g <= '0';
     wait until x = '1';  -- (w2)
     g <= '1';
end process;
```

Initial cycle: Initially, the time is 0 ns, all signals are given the default values:

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
0 ns	+0	U	1	U	U	U	U	U	U	U	U	(start of p2)

signal	time	value
rstn	0 ns	0
z	0 ns	U
c2	0 ns	1
clk	10 ns	0
f	15 ns	U
rstn	30 ns	1

Execute all concurrent statements and processes (all 'event attributes are set to false):

To execute:	c1	c2	c3	c4	c5	p1	p2

After execution, the assignment table is given to the right. The initial cycle is done.

Cycle 1: Next action time is still time 0 ns (but delta 1). The bolded values are updated (and their 'event = true):

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
0 ns	**+1**	**0**	1	U	U	U	U	U	**1**	U	U	**w1**

To execute:	p1
Why?	rstn'event

signal	time	value
x	0 ns	**0**
y	0 ns	**0**
clk	10 ns	0
f	15 ns	U
rstn	30 ns	1

No change is made to *z*, so it's 'event is kept false.

Cycle 2: Still time 0 ns. Note how it is the updated signals that triggers the next execution.

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
0 ns	**+2**	0	1	U	**0**	**0**	U	U	1	U	U	w1

To execute:	c3	p2
Why?	x'event, y'event	y'event, y=0

signal	time	value
z	0 ns	**0**
g	0 ns	**0**
clk	10 ns	0
f	15 ns	U
rstn	30 ns	1

Cycle 3:

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
0 ns	**+3**	0	1	**0**	0	0	U	**0**	1	U	U	**w2**

To execute:	c4
Why?	z'event

signal	time	value
clk	10 ns	0
f	15 ns	**0**
rstn	30 ns	1

Cycle 4: Nothing more to do at 0 ns. Jump to a new time.

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
10 ns	+0	0	**0**	0	0	0	U	0	1	U	U	w2

To execute:	c2	c5	p1
Why?	clk'event	clk'event	clk'event

signal	time	value
c2	10ns	**0**
x	10ns	**0**
y	10ns	**0**
f	15 ns	0
clk	**20ns**	**1**
rstn	30 ns	1

Cycle 5: Here, x and y should be assigned 0, but they are already 0, so nothing happens.

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
10 ns	**+1**	0	0	0	0	0	U	0	**0**	U	U	w2

To execute:	
Why?	

signal	time	value
f	15 ns	0
clk	**20ns**	**1**
rstn	30 ns	1

Nothing to execute this time (c2 does not trigger anything)....

Cycle 6:

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
15 ns	+0	0	0	0	0	0	**0**	0	0	U	U	w2

To execute:	
Why?	

signal	time	value
clk	20 ns	1
rstn	30 ns	1

...nor this time (nothing depends on f).

Cycle 7:

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
20 ns	+0	0	**1**	0	0	0	0	0	0	U	U	w2

To execute:	c2	c5	p1
Why?	clk'event	clk'event	clk'event

signal	time	value
c2	20ns	**1**
x	20ns	**0**
y	20ns	**0**
clk	**30ns**	**0**
rstn	30 ns	1

Cycle 8:

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
20 ns	**+1**	0	1	0	0	0	0	0	**1**	U	U	w2

To execute:	
Why?	

signal	time	value
clk	30ns	0
rstn	30 ns	1

Cycle 9: Time to release the reset.

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
30 ns	+0	**1**	**0**	0	0	0	0	0	1	U	U	w2

To execute:	c2	c5	p1
Why?	clk'event	clk'event	clk'event, rstn'event

signal	time	value
c2	30ns	**0**
clk	**40ns**	**1**

Cycle 10:

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
30 ns	**+1**	1	0	0	0	0	0	0	**0**	U	U	w2

To execute:	
Why?	

signal	time	value
clk	40ns	1

Cycle 11: First rising_edge(clk) after released reset.

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
40 ns	+0	1	**1**	0	0	0	0	0	0	U	U	w2

To execute:	c2	c5	p1
Why?	clk'event	clk'event	clk'event

signal	time	value
c2	40ns	**1**
y	40ns	**0**
x	40ns	**1**
cb	40ns	**1**
c2b	40ns	**0**
clk	**50ns**	**0**

Cycle 12:

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
40 ns	**+1**	1	1	0	**1**	0	0	0	**1**	**1**	**0**	w2

To execute:	c3	p2
Why?	x'event	x'event, x=1

signal	time	value
z	40ns	**1**
g	40ns	**1**
clk	50ns	0

Cycle 13:

time	delta	rstn	clk	z	x	y	f	g	c2	cb	c2b	p2 pointer
40 ns	**+2**	1	1	**1**	1	0	0	**1**	1	1	0	**w1**

To execute:	c4
Why?	z'event

signal	time	value
clk	50ns	0
f	55ns	**1**

And so on

In the ModelSim waveform, you can see the delta delays, if you select:

- Wave -> Expanded Time -> Deltas Mode
- Wave -> Expanded Time -> Expand All

In (assertion) reports, in the transcript window, the time and iteration identifies in which cycle the report where performed.

Why is the delta delay a problem?

The answer is: Since it produces behaviours that depends on things they should not depend on.

An example.

In the RTL code:

```
process(fast_clk) begin
   if rising_edge(fast_clk) then
      slow_clk <= not slow_clk;
   end if;
end process;
process(slow_clk) begin
   if rising_edge(slow_clk) then
      out_data <= "yada yada";
   end if;
end process;
out_clk <= slow_clk;
```

In the test bench:

```
process(out_clk) begin
   if rising_edge(out_clk) then
      -- clock in the result.
      res <= out_data;
   end if;
end process;
assert res = "yada yada"
  report "No no!" severity note;
```

Here. Assume fast_clk is updated at delta cycle +0. Then

- slow_clk is updated at delta cycle +1.
- out_data is updated at delta cycle +2.
- out_clk is updated at delta cycle +2.
- res is updated at delta cycle +3 (and gets the new out_data value, not the old as one can think).
- The assert will be performed on delta cycle +4.

Probably this was not what the programmer wanted. The problem here comes from the statement “out_clk <= slow_clk”. Since it's not allowed to read from an output port signal, the problem will easily occur when generating clocks inside a module.