Differences between Verilog/VHDL Timing Model

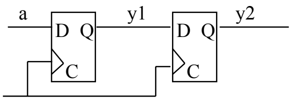
- Verilog and VHDL have very different timing models
- You need to understand the differences, pitfalls of the Verilog timing model.
- These notes are based on the SNUG 2000 paper by Clifford Cummings, 'Nonblocking Assignments inVerilog Synthesis, Coding Styles that Kill!".

Timetest (VHDL)

```
signal a,y1,y2: std_logic;
process (clk)
  begin
    if (clk'event and clk='1') then
      y1 <= a;
      y2 <= y1;
    end if;
  end process;
```

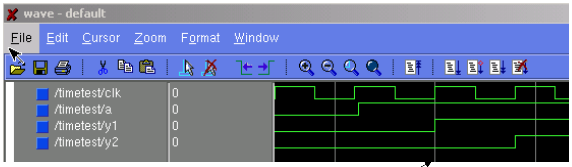
Rising clock edge

Signal assignment does not occur until process suspends, so y2 gets old value of y1, which simulates chain of DFFs.



Synthesis correctly produces chain of DFFs

VHDL Simulation of RTL



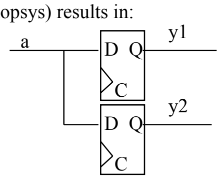
Y1, Y2 behave as expected.

Verilog TimeTest

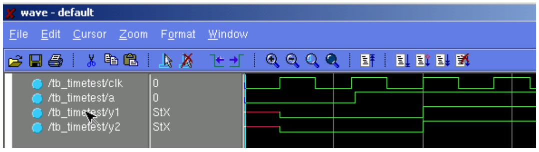
```
module timetest (y1,y2,a,clk);
output y1,y2;
input a,clk;
reg y1,y2;
 always @(posedge clk) begin
   y1 = a;
   y2 = y1;
 end
endmodule
```

A *blocking* assignment

Synthesis (Synopsys) results in:



Verilog Simulation of RTL (Modelsim)



Note that Y1, Y2 change at the same time.

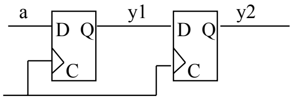
Y1, Y2 act like variables in VHDL, not as signals

Timetest, 2nd try

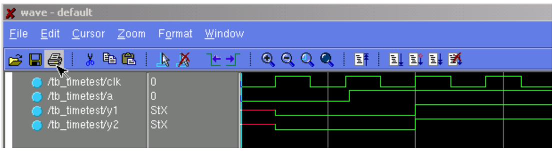
```
module timetest (y1,y2,a,clk);
output y1,y2;
input a,clk;
reg y1,y2;
 always @(posedge clk) begin
   y1 = a;
 end

 always @(posedge clk) begin
   y2 = y1;
 end
endmodule
```

Try Separate processes

Synthesis results in DFF chain



Verilog Simulation of RTL



/tb_timetest/clk 0
/tb_timetest/a 0
/tb_timetest/y1 StX
/tb_timetest/y2 StX

Note that Y1, Y2 **still** change at the same time.

This is scary – RTL simulation results do not match what is synthesized. Verilog zero-delay RTL using blocking assignments is dangerous to use.

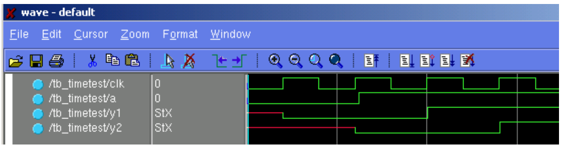
Timetest, 3rd try

```
module timetest (y1,y2,a,clk);
output y1,y2;
input a,clk;

reg y1,y2;

 always @(posedge clk) begin
   y1 = #1 a;
 end


 always @(posedge clk) begin
   y2 = #1 y1;
 end

endmodule
```

Delays are added. Note that the delays are added on the right hand side, in front of the ‘a’ signal. This means that the ‘a’ value is sampled on the rising edge, but the assignment is delayed by 1 time unit, and so simulates a clock-to-q delay.

Synthesis results in DFF chain.

Verilog RTL Simulation



/tb_timetest/clk 0
/tb_timetest/a 0
/tb_timetest/y1 StX
/tb_timetest/y2 StX

Note that now Y1, Y2 now simulate chained DFFs as expected.

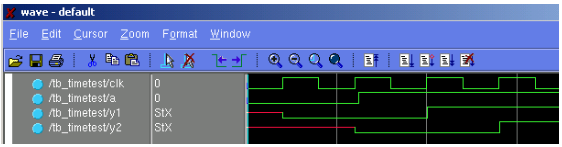
While this works, this considered *poor coding style* to use delays on right hand side of operator in blocking assignment.

Timetest, another example

```
module timetest (y1,y2,a,clk);
output y1,y2;
input a,clk;

reg y1,y2;

 always @(posedge clk) begin
   y2 = y1;
 end

 always @(posedge clk) begin
   y1 = a;
 end


 endmodule
```

Removed delays, reversed ordering of *always* blocks.

Verilog RTL Simulation



/tb_timetest/clk 0
/tb_timetest/a 0
/tb_timetest/y1 StX
/tb_timetest/y2 StX

Simulation now has Y2 changing after Y1.

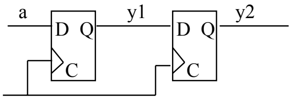
With zero delay code, ordering of *always* blocks affects RTL simulation results when blocking assignments are used.

Nonblocking Assignments

```
module timetest (y1,y2,a,clk);
output y1,y2;
input a,clk;

reg y1,y2;

 always @(posedge clk) begin
   y1 <= a;
   y2 <= y1;
 end

endmodule
```

Nonblocking assignment

Synthesis results in DFF chain



nonblocking vs blocking assignments

- A nonblocking assignment (<=) samples right hand side (RHS) at beginning of timestep; with the actual assignment (the LHS) taking place at the end of the timestep
  - Works like a signal assignment in VHDL
- A blocking assignment (=) will evaluate the RHS and perform the LHS assignment without interruption from another Verilog statement
  - Works like a variable assignment (:=) in VHDL
- Should use nonblocking assignments in *always* blocks used to synthesize/simulate sequential logic.

More on nonblocking assignments

```
module timetest (y1,y2,a,clk);
output y1,y2;
input a,clk;
reg y1,y2;
 always @(posedge clk) begin
   y1 <= a;
 end

 always @(posedge clk) begin
   y2 <= y1;
 end
endmodule
```

With nonblocking assignments, ordering of these *always* blocks does not affect RTL simulation or synthesized gates.

When to use blocking assignments

Use blocking assignments for always blocks that are purely combinational

```
  reg  y, t1, t2;
 always @(a or b or c or d) begin
    t1 = a & b;
    t2 = c & d;
    y  = t1 | t2;
  end
```

RTL simulation and synthesis results match

Nonblocking and combinational processes

```
always @(a or b or c or d) begin
   t1 <= a & b;
   t2 <= c & d;
   y  <= t1 | t2;
 end
```

The problem with this is that during RTL simulation, ‘y’ will get the old value of t1, t2; not the current value (this also happens in VHDL if these are signals).

```
always @(a or b or c or d or t1 or t2) begin
   t1 <= a & b;
   t2 <= c & d;
   y  <= t1 | t2;
 end
```

Adding t1, t2 to the sensitivity list fixes this problem (as it would in VHDL), but results in inefficient simulation since always block triggered twice to get correct value.

Some Rules

- The paper by Cummings lists several rules for writing Verilog in which RTL simulation will match synthesized gate level simulation. The most important of these rules are:
  - Use blocking assignments in *always* blocks that are purely combinational
  - Use only nonblocking assignments in *always* blocks that are either purely sequential or have a mixture of combinational and sequential assignments.
- If you understand the differences between blocking and nonblocking assignments in terms of simulation, then these rules are self-evident.

A Subtle Error if using blocking assignments for sequential logic

```
module dff (q,a,clk);
output q;
input a,clk;
reg q;
 always @(posedge clk) begin
   q = #1 a;
 end
 endmodule
```

Correct DFF simulation, ‘a’ sampled on rising edge, assigned 1 time unit after rising edge.

```
module dff (q,a,clk);
output q;
input a,clk;
reg q;
 always @(posedge clk) begin
   #1 q = a;
 end
 endmodule
```

Delays 1 time unit after rising edge, then samples ‘a’ value, and assigns this to ‘q’. This is modeling *negative* setup time!!!!