

# Non-blocking assignment and race conditions

Blocking and non-blocking assignment in VHDL

Delta Delays

Race condition

DO-254 based design and verification

## Question 1:

blocking and non blocking assignment in VHDL

ChatGPT said:

In VHDL there is **no “blocking vs non-blocking” keyword** like in Verilog/SystemVerilog. But the same concept exists in a different form:

## ✓ The VHDL equivalents

Verilog/SystemVerilog	VHDL equivalent
Blocking assignment (=)	Variable assignment (:=)
Non-blocking assignment (<=)	Signal assignment (<=)

## 1. Signal Assignment (<=)

### ● Equivalent to Verilog non-blocking

- Updates happen **after the current delta cycle** (i.e. scheduled event).
- Used for modeling **hardware signals / registers / ports**.

Example:

```
signal a, b: std_logic;  
a <= b;  -- scheduled update
```

So if you write:

```
a <= b;  
b <= a;
```

both read old values before updating.

This works like Verilog non-blocking:

```
a <= b;  
b <= a;
```

---

## 2. Variable Assignment (:=)

### ● Equivalent to Verilog blocking

- Updates happen **immediately**.
- Only valid inside a process, procedure or function.

Example:

```
process  
  variable x, y : integer;  
begin  
  x := y;  
  y := x;  
end process;
```

Here, *y* receives the updated *x* instantly.

---

## When to use them?

### Signal assignment (<=)

- Most common
- For combinational **and** sequential logic
- Has scheduling (happens later)

Use for registers:

```
process (clk)  
begin  
  if rising_edge(clk) then  
    q <= d;  
  end if;  
end process;
```

### Variable assignment (:=)

- Temporary calculation inside a process
- Helpful in complex combinational code

Example:

```
process(a,b)
  variable tmp : integer;
begin
  tmp := a + b;
  result <= tmp * 2;
end process;
```

---

## Important rule:

You **can mix** signals and variables in a process.

But signals only update after the process suspends.

Example:

```
process
  variable x : integer;
begin
  x := a + 1;  -- immediate update
  b <= x;      -- scheduled update
end process;
```

---

## Practical hardware effect

### Good sequential:

```
process(clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
```

Creates a real flip-flop.

### Bad sequential:

```
process(clk)
begin
  if rising_edge(clk) then
    q := d; -- variable inside clocked process
  end if;
```

```
end process;
```

This compiles but is confusing and discouraged.

---

## Key Differences Summary

### Variable (:=)

- Immediate update
- Inside processes only
- Like local wires in software

### Signal (<=)

- Update scheduled
  - Global hardware element
  - Used for real signals between components
- 

## Quick Mental Model

### Think like this:

- Variables = act **instantly** in simulation
- Signals = act **after a short delay** (delta cycle)

That delay **models real hardware wires**.

---

--\*--\*--

## Question 2:

please give waveform examples

ChatGPT said:

Sure—let's walk through **waveform examples** that clearly show the difference between VHDL **signal** ( $\leq$ ) and **variable** ( $:=$ ) assignments.

---

## Example 1: Signal assignment ( $\leq$ ) — NON-blocking behavior

**Code:**

```
signal a, b : integer := 1;

process
begin
    a <= b;
    b <= a;
    wait;
end process;
```

**What simulator does:**

At time 0:

1. Read  $a=1, b=1$
2. Schedule updates (NOT applied immediately)

So:

- $a \leq b$  schedules  $a = 1$
- $b \leq a$  schedules  $b = 1$

**Waveform timing:**

Time	a	b	
0	1	1	(old values read)
δ	1	1	(updates applied in delta cycle)

Both read the old values first.  
Exactly like Verilog non-blocking  $\leq$ .

## **Example 2: Variable assignment (:=) — BLOCKING behavior**

### **Code:**

```
process
  variable x, y : integer := 1;
begin
  x := y;
  y := x;
  wait;
end process;
```

### **What simulator does:**

- First line updates x immediately
- Second line sees NEW x

### **Waveform:**

Time	x	y
0	1	1
after x := y	1	1
after y := x	1	1

(but order matters!)

Order matters just like in software execution.

---

## **The cool case where the difference shows dramatically**

### **SIGNAL version:**

```
signal s1, s2: integer := 0;

process
begin
  s1 <= 1;
```

```
s2 <= s1;  
wait;  
end process;
```

### Waveform:

Time 0: s1=0, s2=0  
Updates scheduled:  
s1 <= 1  
s2 <= 0 (old s1)

Time 8:  
s1=1, s2=0

### Result:

```
s1 = 1  
s2 = 0
```

---

## VARIABLE version:

```
process  
  variable v1, v2: integer := 0;  
begin  
  v1 := 1;  
  v2 := v1;  
  wait;  
end process;
```

### Waveform:

Time 0  
v1 := 1  
v2 := 1

### Result:

```
v1 = 1  
v2 = 1
```

### Summary waveform view:

Signals:  
s1 <= 1 gives s1=1, s2=0

Variables:  
v1 := 1 gives v1=1, v2=1

---

## Example with clock edge (very common)

**SIGNAL version (correct for registers):**

```
process(clk)
begin
    if rising_edge(clk) then
        q <= d;
    end if;
end process;
```

Waveform:

```
clk:  _/_\_/_/_/_/_/_/_
d:    0  1  0
q:    x  0  1  0    (updates on rising edge)
```

---

## Variable inside clock is different

```
process(clk)
    variable tmp : integer;
begin
    if rising_edge(clk) then
        tmp := d;
        q <= tmp;
    end if;
end process;
```

Waveform equivalent to:

```
q <= d;
```

because variable updates immediately.

---



## FINAL VISUAL RULE

**Signal (<=):**

```
read old values
schedule update
update after delta cycle
```



## Variable (:=):

update instantly  
next line sees new value

---

--\*--\*--\*--

## Question 3:

Please show how different simulators show delta cycles

ChatGPT said:

Great question—this is something many beginners never get to see clearly. Different VHDL simulators **don't disagree on delta cycles**, but they **visualize them differently**. I'll show you how the popular ones display it.

---

## First: why delta cycles matter

Consider:

```
signal a, b : std_logic := '0';

process
begin
    a <= '1';
    b <= a;
    wait;
end process;
```

During simulation:

1. Both assignments **read old values**.
2. Updates occur in **a later delta cycle**.

Expected result:

```
a = '1'
b = '0'
```

---



## Key concept

Events happen in this order:

```
Time 0
  Delta 0: process triggers, read signals
  Delta 1: scheduled updates applied
```

So a signal may change **at the same time**, but in a different delta.

---

## Now let's show different simulators

---

### 1. ModelSim / QuestaSim

ModelSim actually shows delta cycles explicitly in its event log:

```
# 0 ns
#   a <= '1'
#   b <= '0'
# ** Delta #1 update **
#   a = '1'
#   b = '0'
```

In the waveform, both updates appear at **the same time**, but you can click the cursor and see multiple events at 0 ns:

Time		a	b
0ns	d0	0	0
0ns	d1	1	0

And the event list looks like:

```
0 ns    delta 0
0 ns    delta 1
```

---

### 2. GHDL

GHDL prints deltas when run with `--disp-tree` or `--wave`:

Source: <https://chatgpt.com/c/692bab6a-fda0-832e-ac36-b4e5730f08a9>

```
0 fs delta 0: process executes
0 fs delta 1: signal update
```

And in GTKWave waveform viewer:

You see an extra marker when the value updated without time advancing:

```
0
|
|----value changes but time stays at 0----
```

GTKWave shows a zig-zag marker at same time.

---

### 3. Xilinx Vivado Simulator

Vivado does NOT show delta numbers directly, but you will see:

- multiple events at the same timestamp in the log
- signals change in the same time step

Example:

```
Time 0 ps: a <= 1
Time 0 ps: b <= 0
```

But if you enable `Log All Events`, Vivado shows:

```
Time 0 ps (delta cycle)
Time 0 ps (delta cycle)
```

Waveform looks like ModelSim—two events at the same timestamp.

---

### 4. Aldec Riviera-PRO / Active-HDL

These simulators show deltas explicitly in the event list:

```
0 ns 50
0 ns 51
```

Example event trace:

```
0ns(0): process wakeup
```

Source: <https://chatgpt.com/c/692bab6a-fda0-832e-ac36-b4e5730f08a9>

```
0ns(1): signal update
```

Waveform view often shows tiny hash marks for delta transitions.

---

## 5. Cadence Xcelium

Xcelium has the cleanest textual version:

```
0:0    a<=1
0:1    b<=0
```

They actually print delta numbers as:

```
(time):(delta)
```

---

## Important: ALL simulators behave the same

The VHDL standard defines delta cycles precisely.

The difference is only:

- **ModelSim/Quest**a: shows deltas in transcript
  - **GHDL**: shows in event trace
  - **Vivado**: hides deltas by default
  - **Aldec/Riviera**: shows (delta #)
- 

## Visual summary

```
Time = 0
  Δ0 process runs
  Δ1 signal updates happen
Time = 10ns
  Δ0 process runs
  Δ1 scheduled updates
```

Most waveforms flatten deltas into the same timestamp:

0ns: a and b update, even though delta differs

---

--\*--\*--\*--

## Question 4:

Please show how deltas explain race conditions

ChatGPT said:

Great—this is **the #1 place where delta cycles cause real bugs**.  
Once you understand this, a lot of VHDL timing behavior suddenly makes sense.

---

## The classic race condition example

Two processes, both sensitive to the same signal:

```
signal a, b : integer := 0;
```

```
p1: process(a)
begin
    b <= a + 1;
end process;
```

```
p2: process(b)
begin
    a <= b + 1;
end process;
```

Looks harmless, right?

---

## What you might expect in hardware

You might think:

a -> b -> a -> b -> ...

or maybe stable.

But in simulation, **delta cycles decide who wins**.

---



## Let's simulate step-by-step

**Initial:**

$a=0, b=0$

**Assume p1 gets triggered first.**

**$\Delta 0$ :**

p1 runs:  $b \leq a+1 = 1$  (SCHEDULED)

No update yet!

**$\Delta 1$ :**

b updates to 1.

This triggers p2:

**$\Delta 1$ :**

p2 runs:  $a \leq b+1 = 2$  (SCHEDULED)

**$\Delta 2$ :**

a updates to 2  $\rightarrow$  triggers p1

**$\Delta 2$ :**

p1 runs:  $b \leq a+1 = 3$

**$\Delta 3$ :**

b updates, triggers p2

**$\Delta 3$ :**

$a \leq 4$

---

## **Result: infinite oscillation AT TIME 0**

Waveform (conceptual):

Time	$\Delta$	a	b
0	0	0	0
	1	0	1
	2	2	1
	3	2	3
	4	4	3
	5	4	5
	6	6	5
	...		

No real time advances.

If a simulator didn't have a maximum delta count?  
It would never stop.

Most simulators abort with:

Fatal: delta cycle limit reached.

---

## **Why did this happen?**

Because signals update **after the delta**, this happens:

```
process reads OLD values
process writes NEW values
update happens later
triggering other processes
```

So ordering between processes is based on delta scheduling.

---

## **Can this happen in real hardware?**

Not like that. In real hardware:

- all combinational logic settles in parallel
- no sequential feedback unless you add registers

The example is **pure combinational feedback**, which is illegal hardware.

Simulation reveals the mistake.

---

## ✓ Fix with a register

```
process(clk)
begin
    if rising_edge(clk) then
        b <= a + 1;
        a <= b + 1;
    end if;
end process;
```

Now waveform is clean:

```
Time 0: a=0
Time 1: b=1
Time 2: a=2
Time 3: b=3
...
```

Delta cycles disappear because updates happen only on clock edges.

---

## ✓ Fix with ordering inside one process

```
process(a)
begin
    b <= a + 1;
    a <= b + 1; -- reads old b
end process;
```

Safe because a single process runs in one delta.

---



## Key lesson

**Race conditions happen when:**

- multiple processes write signals that trigger each other



- signal updates use **old values**
- delta cycle scheduling defines order

---

## ✓ Engineering rule

If two processes read/write the same signals, beware of racing deltas.

---

\_\_\*\_\_\*\_\_\*\_\_

## Question 5:

Please show a real FPGA example where this bug happened

ChatGPT said:

Thought for 10s

Nice — I'll show **two real, documented FPGA cases** where delta-cycle / combinational-feedback timing in VHDL caused *simulation-vs-hardware* or *unexpected in-system* bugs, explain exactly why the delta cycles produced the misbehavior, and give concrete fixes you can apply immediately.

I'll cite the original reports and an authoritative delta-cycle explainer so you can read the source material. [Adaptive Support+2Reddit+2](#)

---

## Example 1 — “Worked in simulation, behaved oddly on Nexys2 FPGA”

**Source:** forum report of an old design that simulated fine but produced random/odd outputs on a Digilent Nexys2 board. [Adaptive Support](#)

### Problem (typical code pattern)

Two or more processes form **combinational feedback** (no registers) so simulation schedules a chain of signal updates through delta cycles — sometimes the simulator happily settles, but the synthesized logic in the FPGA becomes a real combinational loop or a latch/oscillator and behaves unpredictably in hardware.

Minimal problematic pattern:

```
-- process p1
p1: process(x, y)
begin
    y <= x + 1;    -- signal assigned (scheduled)
end process;

-- process p2
p2: process(y)
begin
    x <= y + 1;    -- signal assigned (scheduled)
end process;
```

### What happened in the field

- In simulation, the tool might produce a repeating delta-cycle trace (or hit a simulator delta limit), or appear stable because of initialization assumptions.
- On the FPGA, synthesis tries to implement the logic; the combinational feedback becomes a real hardware loop (or optimized away oddly) causing **random outputs, oscillation, or stuck conditions**. The forum report described outputs that were “random” on hardware despite expected simulated behavior. [Adaptive Support](#)

## Why delta cycles matter here

VHDL signal ( $\leq$ ) updates are **scheduled** and take effect in a later delta. That scheduling can create zero-time chains ( $\Delta 1, \Delta 2, \Delta 3, \dots$ ) in simulation that don't correspond to how real gates settle with propagation delays — so simulation and synthesis may disagree about stable logic. See a plain explanation of delta cycles. [VHDLwhiz](#)

## Fixes

1. **Insert registers (clock the feedback)** so updates occur on clock edges — this removes purely combinational loops.
2. **Merge logic into a single process** using variables if the behavior is intended to be sequential in a single evaluation.
3. **Refactor to break feedback** — add explicit state machine registers or handshake signals.

Fixed (register) example:

```
process (clk)
begin
    if rising_edge(clk) then
        y_reg <= x_reg + 1;
        x_reg <= y_reg + 1;
    end if;
end process;
```

---

## Example 2 — “Gated-clock / clock-tree delta delay causing unbalanced clocks”

**Source:** field reports & community posts describing gated clocks appearing delayed by a delta cycle and causing edge detectors or clocked logic to fire in a different order (reddit and StackOverflow threads). [Reddit+1](#)

## Problem (typical pattern)

A gated clock (or a clock derived inside VHDL using signal logic) ends up being toggled or gated using a signal assignment that introduces an extra delta. Two clocked domains that should

be aligned now have a **delta offset** so the order of execution between processes changes in the simulator — producing missed edges or glitches in simulation and, sometimes, subtle mismatches after synthesis.

Minimal problematic pattern:

```
gated_clk <= clk and enable; -- signal update may be scheduled
-- process using gated_clk
process(gated_clk)
begin
    if rising_edge(gated_clk) then
        -- ...
    end if;
end process;
```

## Field symptom

Designers saw that an edge detector or sequential logic driven by the gated clock produced glitches or missed events in simulation; ordering between parent clock domain and gated clock domain became inconsistent. In some cases designers had to add dummy assignments or restructure the gating to avoid the delta offset. [Reddit+1](#)

## Why delta cycles matter here

The gated clock's generation via signal assignments can be resolved in a later delta than other processes, so even at the *same timestamp* the simulator runs some processes earlier than others ( $\Delta 0$  vs  $\Delta 1$ ). That changes perceived timing ordering. Real hardware uses physical clock gating or careful CDC, so the VHDL behavioral construction mismatches the synthesized implementation.

## Fixes

1. **Use clock-enable (preferred)** instead of synthesizable combinational gating: keep the root clock and sample enable inside the clocked process:

```
process(clk)
begin
    if rising_edge(clk) then
        if enable = '1' then
            gated_action <= data;
        end if;
    end if;
end process;
```

2. **If using gated clocks is necessary**, ensure you use vendor-provided clock gating cells or special IP that synthesizes to proper clock-gates.
3. **Avoid creating clocks with combinational signal assignments** in VHDL; the delta scheduling can change the order of execution in simulation.

# Short concrete reproduction steps you can run locally

(so you can *see* the delta-driven mismatch)

1. Create a minimal two-process VHDL file with the combinational feedback example above.
2. Simulate with GHDL or ModelSim and watch the event log — you'll see repeated delta updates (or "delta cycle limit" errors). GHDL/ModelSim show delta events in the transcript; GTKWave will show changes at the same timestamp. [VHDLwhiz+1](#)
3. Synthesize and load to an FPGA — the logic will implement as combinational feedback (or tools will warn about combinational loop) and hardware behavior will differ. Synthesis/warning pages (Vivado/Quartus DRC) will often report combinational loop warnings. [Adaptive Support+1](#)

---

## TL;DR — how delta cycles explain the bug

- **VHDL signals** update **after** the process (scheduled into a later delta).
- Multiple processes feeding each other via signals can create chains of scheduled updates that run through many deltas ( $\Delta_0 \rightarrow \Delta_1 \rightarrow \Delta_2 \rightarrow \dots$ ). Simulators model this faithfully. [VHDLwhiz](#)
- **In hardware** there are real gate delays and the synthesizer may implement the logic as a physical loop or optimize differently — often resulting in different behavior than the zero-time delta chain in sim. This mismatch produces the field bugs reported. [Adaptive Support+1](#)

---

--\*--\*--\*--

## Question 6:

Can you give a tiny testbench + step-by-step modelsim commands and an ASCII waveform showing the delta cycles

ChatGPT said:

Absolutely. Here is a **minimal reproducible race/delta-cycle example** that you can paste into ModelSim/Questa and run immediately.



## DUT that causes a delta-cycle oscillation

**file:** dut.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity dut is
end entity;

architecture rtl of dut is
    signal a, b : std_logic := '0';
begin

    p1: process(a)
    begin
        b <= not a;
    end process;

    p2: process(b)
    begin
        a <= not b;
    end process;

end architecture;
```

This is the classic circular dependency:

a -> b -> a

which produces delta-cycle updates forever (no real time advances).



## Testbench

## file: `tb.vhd`

```
library ieee;
use ieee.std_logic_1164.all;

entity tb is
end entity;

architecture sim of tb is
    component dut is end component;
begin

    uut: dut;

    -- Stop the simulation after some time
    process
    begin
        wait for 10 ns;
        assert false report "Simulation finished" severity failure;
    end process;

end architecture;
```

---

## What happens in simulation

At time 0, p1 and p2 trigger repeatedly:

```
0 ns Δ0: a=0 b=0
0 ns Δ1: a=1 b=0
0 ns Δ2: a=1 b=1
0 ns Δ3: a=0 b=1
0 ns Δ4: a=0 b=0
...
```

No time ever moves forward — only delta cycles.

---



## Running it in ModelSim step-by-step

From the ModelSim console:

```
vlib work
vcom dut.vhd
vcom tb.vhd
vsim tb
run 0
```

Source: <https://chatgpt.com/c/692bab6a-fda0-832e-ac36-b4e5730f08a9>

You should immediately see something like:

```
# ** Warning: (vsim-3601) Iteration limit reached at time 0 ns.
```

If your ModelSim is configured to stop after too many deltas.

---

## If you want to SEE the deltas happen:

### Step 1: add waveform

```
add wave *  
run 0
```

### Step 2: now step through delta cycles manually:

```
run -delta 1  
run -delta 1  
run -delta 1  
run -delta 1
```

Every step updates either a or b again without advancing time.

---

## ModelSim transcript example

You will typically see:

```
# 0 ns: Delta 0: a='0', b='0'  
# 0 ns: Delta 1: a='1', b='0'  
# 0 ns: Delta 2: a='1', b='1'  
# 0 ns: Delta 3: a='0', b='1'  
# 0 ns: Delta 4: a='0', b='0'  
...
```

Some versions show the iteration number instead of “delta”.

---





## ASCII waveform of deltas

Time	$\Delta$	a	b
0	0	0	0
	1	1	0
	2	1	1
	3	0	1
	4	0	0
	5	1	0
	6	1	1
	...		

Notice:

- timestamp never changes
- values keep toggling
- delta increases

This is the smoking-gun symptom of a combinational loop + signal updates.

---



## Why this happens

Because VHDL signal assignments (`<=`) schedule updates into **later delta cycles**, each process reacts to the other's update.

Simulation:

```
p1 runs → schedules b update
p2 runs → schedules a update
repeat forever
```

Real hardware:

- becomes a combinational loop = illegal/bad design
-

## **Fix (turn into hardware)**

Just clock the signal:

```
if rising_edge(clk) then
  b <= not a;
  a <= not b;
end if;
```

Now no deltas race, no oscillation.

---

\_\_\*\_\_\*\_\_\*\_\_

## Question 7:

can you show a version that looks safe but races

ChatGPT said:

Absolutely—this is the **most dangerous kind of delta-cycle race** because the code *looks perfectly reasonable and hardware-like*, but still races in simulation due to **signal scheduling order and delta semantics**.

---

### Version that *looks safe* but still races

We only change one detail: both processes use the **same clock**.

```
signal a, b : std_logic := '0';

p1 : process(clk)
begin
    if rising_edge(clk) then
        a <= not b;
    end if;
end process;

p2 : process(clk)
begin
    if rising_edge(clk) then
        b <= a;
    end if;
end process;
```

At first glance, this looks like a normal sequential design.

But in simulation, this is still a race.

---

## Why? Because both $a$ and $b$ read old values on the same delta

Let's assume:

```
a = 0
b = 1
```

Clock rises.

**$\Delta 0$ :**

- p1 executes: `a <= not b` schedules `a = 0`
- p2 executes: `b <= a` schedules `b = 0`

But neither updates yet!

**$\Delta 1$ :**

signal update phase:

- `a = 0`
- `b = 0`

Next clock rises...

**$\Delta 0$ :**

- p1 reads `b = 0`, schedules `a = 1`
- p2 reads `a = 0`, schedules `b = 0`

**$\Delta 1$  updates:**

- `a = 1`
- `b = 0`

Next clock rise...

...and you get this repeating pattern:

---

## ASCII waveform

```
clk:  _/^-\_/_/^-\_/_/^-\_/_
a:    0  0  1  1  0  ...
b:    1  0  0  0  1  ...
      ^ reads old values each cycle
```

Each register update uses the other signal's **previous** value because both use `<=`.

Even though both are clocked processes, there is a race in the sense of **which previous value wins**.

This difference is caused by delta ordering inside the same clock tick.

---

## Why this is dangerous

Beginners often expect:

a and b update in sequence

But the real semantics are:

both processes execute  
they both read old values  
updates scheduled for later

So the two registers are not functionally identical to a single process.

---

## ✓ Corrected version

Combine the processes, or use a temporary variable:

```
process(clk)
    variable tmp : std_logic;
begin
    if rising_edge(clk) then
        tmp := not b;
        b   <= a;
        a   <= tmp;
    end if;
end process;
```

Now there is no ambiguity:

- variable assignment happens immediately
  - you control evaluation order
  - no cross-process delta races
-

## Even more subtle version

Many people try this thinking it fixes it:

```
a <= not b after 0 ns;
```

Still races (just adds another delta).

---

## Takeaway rule

**Two clocked processes updating each other's signals is a race unless:**

- They are registered in the same process, or
- Controlled using variables, or
- Broken by deterministic pipeline staging.

Even with the same clock, **multiple clocked processes can race in the delta domain.**

---

--\*--\*--\*--

## Question 8:

Can you please show the version where the race happens only in simulation but not after synthesis

ChatGPT said:

Absolutely—this is one of those classic “works in post-synthesis timing sim or on FPGA, but races in RTL simulation” traps. Below is a **realistic minimal VHDL example** that looks clean and synthesizes fine, and a real race only appears in RTL simulation because of delta-cycle ordering.

---

## ✓ Scenario: two processes communicate via signals → race in RTL, but not after synthesis

**FILE: race.vhd**

```
library ieee;
use ieee.std_logic_1164.all;

entity race is
    port(
        clk : in  std_logic;
        a   : in  std_logic;
        y   : out std_logic
    );
end entity;

architecture rtl of race is
    signal flag : std_logic := '0';
    signal reg  : std_logic := '0';
begin

    -- process A drives flag
    proc_a : process(clk)
    begin
        if rising_edge(clk) then
            if a = '1' then
                flag <= '1';
            else
                flag <= '0';
            end if;
        end if;
    end process;

    -- process B reads flag
    proc_b : process(flag)
    begin
        reg <= flag;
    end process;

end architecture;
```

```
-- process B uses flag
proc_b : process(clk)
begin
    if rising_edge(clk) then
        if flag = '1' then      -- <-- this read is the problem!
            reg <= '1';
        else
            reg <= '0';
        end if;
    end if;
end process;

y <= reg;
end architecture;
```



## What we *think* happens

Both see the same rising edge → flag is updated → reg uses the new value.

But in **RTL simulation**:

Process A schedules `flag <= new_value` for the **end of the delta cycle**  
Process B reads the **old value**.

So depending on delta ordering in your simulator:

- ModelSim gives `flag_old` to proc B
- Questa, Riviera or GHDL may schedule differently
- Synthesis will infer **real registers**, so both behave synchronously → bug disappears



## Why it works after synthesis

These two processes become **two flip-flops**:

FF1 (flag) → FF2 (reg)  
(clk aligned)

In real hardware on FPGA, both sample on the same rising edge, and reg sees the correct, updated value **one clock later**, deterministically.

No deltas in silicon.



## Demonstration testbench that reproduces the race

### FILE: `tb.vhd`

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb is end;

architecture sim of tb is
    signal clk : std_logic := '0';
    signal a    : std_logic := '0';
    signal y    : std_logic;
begin

    uut : entity work.race
        port map (clk => clk, a => a, y => y);

    clk <= not clk after 5 ns;

    stim : process
    begin
        wait for 8 ns;
        a <= '1';
        wait for 20 ns;
        a <= '0';
        wait;
    end process;
end architecture;
```

---

## ASCII timing showing the race in RTL (ModelSim)

```
time   5ns       10ns       15ns       20ns
      ↑         ↑         ↑
      CLK rising edges
```

Process scheduling order (simplified):

```
delta0:  clk rises
delta1:  proc_a evaluates a=1 → schedules flag<=1
delta2:  proc_b reads old flag=0 → reg<=0
delta3:  signal updates flag=1
```

Result in RTL:

```
-----  
CLK      ┌───┐ ┌───┐ ┌───┐  
flag     0   0   1   1  
reg       0   0   0   1 ← one extra cycle delay here (race)  
y         0   0   0   1  
-----
```

So reg misses the update on the first rising edge.



## After synthesis (real FPGA behaviour)

All edge-triggered registers sample at the same time → no delta cycles.

```
CLK      ┌───┐ ┌───┐ ┌───┐  
flag     0   1   1  
reg       0   1   1 (correct behavior)
```

This is the classic:

RTL simulation does not model synchronous flops the same way hardware does.



## Why this version looks safe but races

- All assignments are clocked
- No combinational loops
- No asynchronous drivers
- No waveform glitches
- No sensitivity list bugs

Yet RTL is wrong.



## Correct (portable-safe) style

Register all cross-process signals:

### **Solution 1: combine into one process**

```
process (clk)
begin
    if rising_edge(clk) then
        flag <= a;
        reg  <= flag;
    end if;
end process;
```

### **Solution 2: synchronize explicitly**

Don't fan-out clocked signals across processes unless intentionally pipelined.

---

---

--\*--\*--\*--

## Question 9:

Please show the real fix used in FPGA industry

ChatGPT said:

Great question. In professional FPGA design flows there **are several industry-standard fixes** that avoid this simulation-only race 100% reliably. I'll show the ones that real teams use, why they work, and which ones are most recommended today.



### First, the root cause (why tools disagree)

You had two clocked processes communicating directly:

```
proc_A writes flag
proc_B reads flag
```

In simulation the update to `flag` is scheduled after a delta delay, so `proc_B` may see the old value.

In hardware there are no delta cycles—just clock edges—so it behaves differently.

So the goal is always the same:

**Eliminate simulation ordering dependencies.**

### ✓ Fix #1 (MOST common in industry): use ONE clocked process

Combining both flops into the same sequential block guarantees ordering is simulation-portable and synthesis-portable:

```
process(clk)
begin
    if rising_edge(clk) then
        flag <= a;
        reg <= flag;
    end if;
end process;
```

Why it works:

- VHDL simulators evaluate statements top-down
- Synthesis maps to two flop stages (identical to the good hardware case)
- No delta races, ever

This is by far the **most accepted fix** in aerospace, medical, and telecom FPGA coding standards.

---

## ✓ Fix #2: Use signals only in one direction across clocked logic

Industry practice:

- A sequential signal should be written in **one** process only
- Another sequential process should not react to it in the same cycle

So instead of:

```
proc1 writes S
proc2 reads S same cycle  <-- BAD
```

You change so that proc2 reads it only next cycle.  
That makes RTL match silicon.

This is mandated in major coding standards:

- Xilinx UltraFast
  - Intel Quartus HDL Style Guide
  - DO-254 / FAA-EASA guidelines
  - NASA MAPLD style guides
  - RMM (Reuse Methodology Manual)
  - DASC-UG-9 FPGA Safety Standard
- 

## ✓ Fix #3: Explicit pipeline (real FIFO style)

```
process (clk)
begin
    if rising_edge(clk) then
        flag_r <= a;
        reg_r  <= flag_r;
```

```
    end if;  
end process;
```

This is actually how synthesis will infer your two original flops anyway.

So you explicitly write it that way to avoid the delta-cycle illusion.

---

## ✓ **Fix #4: Use variables inside a single clocked process**

Variables update immediately in simulation, which is exactly how hardware behaves between internal registers:

```
process(clk)  
    variable flag_v : std_logic;  
begin  
    if rising_edge(clk) then  
        flag_v := a;    -- immediate update  
        reg    <= flag_v;  
    end if;  
end process;
```

This is another extremely common industry pattern.

---

## ✓ **Fix #5: Use protected / controlled interface instead of naked signals**

Real-world systems use:

- handshaking (valid/ready)
- registered outputs
- synchronizers
- scoreboards
- CDC modules

Example:

```
if flag_rising_edge = '1' then ...
```

i.e., generate safe events instead of sampling raw signals.

---

## ✓ **Fix #6: Apply a design rule (industry mantra)**

You will see this in manuals from Xilinx, Intel, Microchip, BAE, Thales, ESA:

**1 process per register is fine, but never have two processes that use each other's register outputs in the same cycle.**

Or the more blunt version used at some companies:

**No combinational dependencies across clocked processes.**

These rules are so common that lint tools check them:

- Siemens Questa Lint
  - Blue Pearl
  - Spyglass
  - AscentLint
- 

## **Why not rely on newer simulators?**

Because hardware is the truth.

The fix isn't simulator-compatibility.

The fix is expressing hardware unambiguously.

---



## Summary of industry-proven fixes

Fix	Status
Single clocked process	<b>Most common</b>
Pipeline registers explicitly written	Widely recommended
Single writer per signal	Standard practice
Handshake architecture	Large designs
Variables in sequential logic	Very common
RTL lint rules	Industry standard

And now the key takeaway:

In silicon, behavior is deterministic.

In RTL simulation, races only appear if the code describes something ambiguous.

So industrial practice removes the ambiguity.

\_\_\*\_\_\*\_\_\*\_\_



## Question 10:

please show DO-254 compliant version

ChatGPT said:

Thought for a couple of seconds

## DO-254–style (compliance-oriented) VHDL rewrite

Below is a **DO-254–compliant style** rewrite of the earlier `race` example. It follows common DO-254 guidance and industrial coding rules used in airborne systems:

- single-clock synchronous processes (no cross-process same-cycle dependencies)
- single writer per register/signal where practical
- explicit pipeline / staging of data — no expectation that one register updates another in the same cycle within a separate process
- synchronous reset (clear, deterministic initialization)
- self-checking assertions for simulation-time checks (severity note: assertions in flight code are for verification only; remove or convert per project policy for final load)
- clear signal naming, documentation comments, and a verification checklist included below

---

### File: `do254_race.vhd`

```
-- do254_race.vhd
-- DO-254 style: deterministic synchronous design, single-writer rules,
explicit pipeline
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity do254_race is
  port (
    clk    : in  std_logic;
    reset_n : in std_logic;  -- active-low synchronous reset (per project
policy)
    a_in   : in  std_logic;
    y_out  : out std_logic
  );
end entity;

architecture do254_arch of do254_race is
```

```
-- Stage registers (explicit pipeline)
signal flag_r      : std_logic := '0'; -- registered version of a_in
signal reg_r       : std_logic := '0'; -- final registered output

-- Optional: monitor signal for assertions (verification only)
signal sim_check_trigger : std_logic := '0';

begin

-----
-
-- Synchronous sequential pipeline: single process to describe both flops
-- This guarantees deterministic simulation semantics and matches
synthesized
-- hardware (no delta-cycle ambiguity).
-----
-
pipeline_proc : process(clk)
begin
    if rising_edge(clk) then
        if reset_n = '0' then
            -- Synchronous reset (deterministic, recommended for avionics)
            flag_r <= '0';
            reg_r  <= '0';
        else
            -- Explicit pipeline: capture input, then capture pipeline stage
            flag_r <= a_in;
            reg_r  <= flag_r;
        end if;
    end if;
end process pipeline_proc;

-- Output assignment (single driver for reg_r)
y_out <= reg_r;

-----
-
-- Verification assertions (simulation-only / verification harness).
-- DO-254 guidance: use assertions for requirements verification,
structural
-- checks, and sanity during test. Remove or gate them if required by cert
plan.
-----
-
sim_checks : process(clk)
begin
    if rising_edge(clk) then
        -- Example: check that flag propagates to reg after one clock
        assert not (flag_r = '1' and reg_r = '0')
            report "Flag is set but reg did not capture previous flag value
(simulation check)"
            severity note;
        -- Example: detect metastability risk / CDC misuses (project-specific)
    end if;
end process sim_checks;

end architecture do254_arch;
```

## File: do254\_tb.vhd (minimal verification stimulus)

```
-- do254_tb.vhd -- simple testbench for quick RTL simulation
library ieee;
use ieee.std_logic_1164.all;

entity do254_tb is
end entity;

architecture sim of do254_tb is
    signal clk      : std_logic := '0';
    signal reset_n : std_logic := '0';
    signal a_in     : std_logic := '0';
    signal y_out    : std_logic;
begin

    uut: entity work.do254_race
        port map (
            clk      => clk,
            reset_n  => reset_n,
            a_in     => a_in,
            y_out    => y_out
        );

    -- 100 MHz-like clock (10 ns period)
    clk <= not clk after 5 ns;

    stim: process
    begin
        -- apply synchronous reset for a couple cycles
        reset_n <= '0';
        wait for 20 ns;
        reset_n <= '1';

        -- stimulus: toggle a_in and observe pipeline behavior
        wait for 10 ns;
        a_in <= '1';
        wait for 20 ns;
        a_in <= '0';
        wait for 30 ns;

        -- finish
        assert false report "End of simulation" severity failure;
    end process;

end architecture sim;
```

---

## Why this meets DO-254 expectations (summary)

1. **Deterministic behavior** — single clocked process updates both pipeline stages top-to-bottom; simulation order is identical to synthesis result.
  2. **No same-cycle cross-process dependencies** — avoids delta-cycle races that appear in RTL simulation.
  3. **Synchronous reset** — preferred for deterministic initialization and certification evidence (document reset behavior).
  4. **Single-writer/single-driver** — `reg_r/flag_r` are updated in one process; outputs have a single source.
  5. **Self-checking instrumentation** — assertions provide early detection for requirements verification; these should be described in the verification plan and may be removed or gated in flight code per cert policy.
  6. **Clear naming & comments** — supports traceability to requirements (DO-254 requires traceability between requirements, design, and verification artifacts).
- 

## DO-254 verification checklist (practical steps you should perform)

Use this list to build your DO-254 verification evidence package:

1. **Requirements traceability** — link the VHDL module and signals to system-level requirements in the requirements database.
  2. **Coding standard & review** — run style/lint checks (e.g., AscentLint / CDC lint rules) and perform peer code reviews; record review artifacts.
  3. **Static checks** — lint for single-writer, unused signals, inferred latches, clock-gating usage. Document warnings and fixes.
  4. **Synthesis comparison** — synthesize for target FPGA; capture synthesis report and show the inferred registers match the coding intent.
  5. **Netlist vs RTL simulation** — run post-synthesis timing simulation (SDF back-annotation) and compare behavior to RTL sim for functional equivalence; capture diffs.
  6. **Timing constraints & STA** — create and verify SDC constraints, run static timing analysis, and show timing closure under worst-case corners.
  7. **Hardware bring-up tests** — run targeted hardware tests that exercise the same scenarios; capture logs and waveforms.
  8. **Fault injection / robustness tests** — where applicable, run tests that simulate reset, power-on, and error conditions.
  9. **Verification evidence** — collect testbenches, coverage results (functional and structural), assertion logs, simulation transcripts, and checklist artifacts.
  10. **Configuration management** — store all sources, scripts, and tool versions in revision control and record tool versions used for simulation/synthesis.
-

## Practical notes & project-policy details

- DO-254 projects often require *documented* justification for coding choices (e.g., why synchronous reset, why single-process pipeline). Put these in your design rationale.
- Assertions used for verification must be identified as verification-only and handled according to your project's DAL (Design Assurance Level) plan.
- Lint and formal checks (model checking where useful) are commonly required as part of the DO-254 evidence package.
- Hardware bring-up evidence should include oscilloscope captures or logic analyzer traces when applicable.

---

--\*--\*--\*--