VHDL Coding Guidelines

Table of Contents

1 Introduction

The following coding standards are suggested for VHDL synthesizable code projects. following these standards in general for all designs would be considered a best practice. Those guidelines are adopted from Reference 1.

Each coding standard listed hereafter falls under one of four categories. These categories are arbitrarily named, but each category serves a distinct purpose as described. The categories are

- Coding Practices
- Clock Domain Crossings
- Safe Synthesis
- Code Reviews

Each coding standard has a default severity level of Error, Warning, or Note, which provides a measure of the worst case impact a standard violation can have on safety. In general, errors should always be corrected, warnings should usually be corrected but may have documented and justified exceptions, and notes should simply be examined to ensure there is no impact on safe design operation.

2 Coding Practices (CP)

This category of standards ensures that a coding style supporting safety-critical and good digital design practices are used. Each rule that follows is given a coding practice (CP) number for ease of reference

2.1 Avoid Incorrect VHDL Type Usage (CP1)

Check for the incorrect use of types, incompatible bounds, and/or constraints.
While these types of issues are usually caught during compilation, it is beneficial to correct the problem as early as possible.

Default severity: Error

Example:

```
ENTITY top IS port (rf2fe_vec: IN dword);
ARCHITECTURE rtl OF top IS
signal tx_index: std_logic_vector (14 DOWNTO 0);
….
tx_index <= rf2fe_vec (f_txindex_hi DOWNTO f_txindex_lo);
-- Type error at ‘rf2fe_vec’. Needed type ‘std_logic_vector’
END rtl;
```

2.2 Avoid Duplicate Signal Assignments (CP2)

The same signal should not be assigned a value more than once within the same statement region.

Default severity: Warning

Example:

```
same_signal <= '0'; -- Default assignment
IF (reset = '1') THEN
        same_signal <= '1'; -- This is NOT a duplicate assignment.
END IF;
IF (reset = '1') THEN
        -- Default Reset Values
        beep <= '0';
        -- more lines
        beep <= '1'; -- Duplicate assignment maybe a mistake
ELSIF (falling_edge(clk)) THEN
```

2.3 Avoid Hard-Coded Numeric Values (CP3)

For design IP reuse and portability ease, hard-coded numeric values should not be used. Constants or generics should be used and documented within the design. This will greatly reduce the probability of a design error from creeping into the design code as it is being ported to a new application.

Default severity: Warning

Example:

```
PACKAGE dcc_pkg is
------------------------------------
-- constant declarations
------------------------------------
CONSTANT CPU_ADR_WIDTH      : INTEGER := 16; -- cpu address bus width
CONSTANT CPU_DATA_WIDTH     : INTEGER := 32; -- cpu data bus width
. . .
ENTITY dual_clock_cache_struct is
PORT(
cpu_addr        : IN std_logic_vector( 15 DOWNTO 0 ); -- violation
cpu_di          : IN std_logic_vector(CPU_DATA_WI DTH-1 DOWNTO 0); --32-bit
cpu_clk         : IN std_logic;
```

2.4 Avoid Hard-Coded Vector Assignment (CP4)

For vector reset assignments; do not use hard-coded values.

The reset assignment should be done in a way that is independent of the size of the vector. This limits the impact of changing vector sizes and enhances design portability ease.

Default severity: Note

Example:

```
WHEN OTHERS =>
        clk_div_en <= '0';
        xmitdt_en <= '0';
        ser_if_select <= "00000000"; -- Violation
        ser_in_select <= (OTHERS => ‘0’); -- This is preferred
```

2.5 Ensure Consistent FSM State Encoding Style (CP5)

A design should employ a consistent state encoding s tyle for Finite State Machines (FSM). FSM state types should not be hard-coded, unless unavoidable.

Inconsistent FSM encoding style may interfere with the design’s FSM error detection and recovery scheme for the operating environment. Enumerated state types make HDL code more readable generally.

Note that while this check of the HDL code is important, the synthesis tool must also be set up properly to implement the FSM state encoding consistently in the targeted hardware. The synthesis tool control can be achieved through a pragma in the code or via an external control file. Consult your synthesis tool vendor for more information.

Default severity: Error

Example:

```
TYPE cpu_sm_state_type IS (
        IDLE,           -- reset & default state
        START_OP,
        WRITE_DATA,
        DO_READ,
        WAITMEM,
        STALL_WAIT,
        DO_RD           -- Note, FSM states are encoded as enumerated type
);
P_CPU_SM_NEXT_STATE : PROCESS( . . . )
BEGIN
CASE current_state_r is
        WHEN IDLE =>
        . . .
        WHEN START_OP =>
        . . .
        WHEN “0011” => -- Violation, inconsistent state encoding
        . . .
        END CASE;
END PROCESS P_CPU_SM_NEXT_STATE;
```

2.6 Ensure Safe FSM Transitions (CP6)

- An FSM should have a defined reset state.
- All unused (illegal or undefined) states should transition to a defined state, whereupon this error condition can be processed accordingly.
- There should be no unreachable states (i.e., those without any incoming transitions) and dead-end states (i.e., those without any outgoing transitions) in an FSM.

Default severity: Error

Example:

```
TYPE fsm_state_type IS (
        IDLE,                   -- reset & default state
        START_OP,
        WRITE_DATA,
        DO_READ,
        WAITMEM,                -- wait for memory response
        STALL_WAIT,             -- cpu stall
        DO_RD
        -- AIS                  -- Commented out AIS state will cause violation
);
. . .
CASE current_state IS
        WHEN IDLE =>
                IF (rd_req='1' AND pre='0') THEN
                        next_state <= DO_RD;
        WHEN DO_READ =>                         -- Violation, no incoming transition
                next_state <= DO_RD;
        WHEN DO_RD =>
                IF (pre='1') THEN
                        status <= WAITMEM;      -- Violation, no outgoing transition
        WHEN OTHERS =>                          -- Others, including error states
                next_state <= AIS;              -- transition to the AIS state
                                                -- Violation, AIS is not a defined state
        END CASE;
```

2.7 Avoid Mismatching Ranges (CP7)

Bit widths on both sides of an assignment, comparison, or association should match. Mismatching range in an assignment, comparison, or association might result in logic overflow errors.

Default severity: Warning

Example

```
ENTITY nonStatRange IS
PORT (
        in1 : IN std_logic_vector(31 DOWNTO 0);
        in2 : IN std_logic_vector(31 DOWNTO 0);
        sel1 : IN integer range 0 TO 15;
        sel2 : IN integer range 0 TO 15;
        out1 : OUT std_logic_vector(31 DOWNTO 0));
END;
ARCHITECTURE arch_nonStatRange OF nonStatRange is
BEGIN
        sm_proc: PROCESS(in1, in2)
                variable local_var1 : std_logic_vector (1 DOWNTO 0);
                variable local_var2 : std_logic_vector (1 DOWNTO 0);
        BEGIN
                local_var1 := in1 (sel1 + 1 DOWNTO sel1); --This is not
                --a violation, since the range is actually static although
                --the left and right indices are not.
                local_var2 := in2 (sel1 + sel2 DOWNTO sel1); -- Violation
        END PROCESS;
        END;
```

2.8 Ensure Complete Sensitivity List (CP8)

The sensitivity list should only contain the signals needed by the process. The sensitivity list of a VHDL process should only include the required signals and does not include any unnecessary signals. The unused signals will slow down simulation. Missing signals potentially lead to mismatching simulation versus synthesized hardware function. Existence of unused signals should be justified.

Default severity: Warning

Example:

```
RCV_CLOCKED_PROC : PROCESS (
        clk,
        -- rst,                 --Violation, signal needed in sensitivity list
        rcv_bit_cnt_cld         --Violation, signal unnecessary
                                --will cause process to be trigger more than needed.
)
BEGIN
        IF (rst = '0') THEN
                rcv_current_state <= waiting;
        ELSIF (falling_edge(clk)) THEN
                rcv_current_state <= rcv_next_state;
                CASE rcv_current_state IS
                WHEN waiting =>
                        rcv_bit_cnt_cld <= "000";
                        IF (sin='0') THEN
                        rcv_bit_cnt_cld <= "001";
                    END IF;
                WHEN incr_count2 =>
                    IF (sample='1' AND rcv_bit_cnt_cld /= "111") THEN
                        rcv_bit_cnt_cld <= unsigned(rcv_bit_cnt_cld) + 1;
                    END IF;
                WHEN OTHERS =>
                    NULL;
                END CASE;
            END IF;
        END PROCESS RCV_CLOCKED_PROC;
```

2.9 Ensure Proper Sub-Program Body (CP9)

Each sub-program must:

- have only one exit point
- have no recursion
- access only local variables/signals

Check for all these conditions in procedures, functions, and tasks. In some cases, if the code is intentionally designed this way (such as a recursive sub-program), its justification should be documented.

Default severity: Warning

Example:

```
FUNCTION logicop (in1, in2 : IN bit_vector; c1, c2 : IN bit) RETURN
        bit_vector IS
BEGIN
        IF(c1 = '1') THEN
                RETURN (in1 OR in2);                    -- Violation, multiple exit points
        ELSIF(c2 = '1') AND (adr_mode = ‘1’) THEN
                RETURN (in1 AND in2);                   -- Violation
        END IF;
END FUNCTION;
```

2.10 Assign Value Before Using (CP10)

Every object (e.g., signal, variable, por t) should be assigned a value before using it.
When objects are used before being defined, a latch may be inferred during synthesis, which is most likely unintentional functional behavior for the design.

Default severity: Warning

Example:

```
ENTITY fifo_bk_pressure IS
PORT(
        -- Port Declarations
        clk_ck2 : IN std_logic;                  -- GLOBAL: downstream clock
        rst_ck2_n : IN std_logic;                -- GLOBAL: downstream reset(N)
        cntr_ck1_i : IN std_logic_vector(FIFO_CNT R-1 DOWNTO 0); -- 9-bit
        -- Cut-and-paste error, should be cntr_ck2_i, results in violation
. . .
);
ARCHITECTURE rtl OF fifo_bk_pressure is
------------------------------------
-- register definitions
------------------------------------
signal full_threshold_r : fifo_cntr_type; -- 9-bit data type
. . .
threshold_proc: PROCESS(cntr_ck2_i, full_threshold_r)
BEGIN
        assert_bk_pressure_s <= ‘0’;
        . . .
        ELSIF (cntr_ck2_i = full_threshold_r – CDC_DELAY) THEN
                --VIOLATION “cntr_ck2_i” should be assigned before being read
                assert_bk_pressure_s <= ‘1’;
        END IF;
END PROCESS threshold_proc;
```

2.11 Avoid Unconnected Input Ports (CP11)

All input ports should be driven by a port, signal or constant value. Input ports and bidirectional ports should be connected and not left floating.

Unconnected input ports can cause the associated circuit blocks to exhibit nondeterministic functional behavior in hardware, varying with changes in voltage-rails and operating conditions.

Default severity: Error

Example:

```
component usable_vhd
PORT (
        in1 : IN std_logic ;
        in2 : IN std_logic_vector(MEM_ADR_ WIDTH-1 DOWNTO 0);
        out1 : OUT std_logic_vector(MEM_ADR_WIDTH-1 DOWNTO 0);
        out2 : OUT std_logic_vector(MEM_ADR_WIDTH-1 DOWNTO 0)
);
. . .
U_bad: usable_vhd PORT MAP(
        in1 => d,
        out1 => q,
        out2 => q
);
-- VIOLATION. Unit bound to instance “U_bad” has an
-- unused input port “in2”, that should be
-- actively driven by another signal or fixed to ‘0’ or ‘1’
```

2.12 Avoid Unconnected Output Ports (CP12)

Design output ports should be connected. Output ports should usually be connected to an internal signal. However, in some cases, such as with built-in self test (BIST) output signals used for debug, a floating output is acceptable. In these cases, the violation should be justified and documented.

Default severity: Note

Example:

```
component usable_vhd
PORT (
        in1 : IN std_logic ;
        in2 : IN std_logic_vector(MEM_ADR_ WIDTH-1 DOWNTO 0);
        out1 : OUT std_logic_vector(MEM_ADR_WIDTH-1 DOWNTO 0);
        out2 : OUT std_logic_vector(MEM_ADR_WIDTH-1 DOWNTO 0)
);
...
U_bad: usable_vhd PORT MAP (
        in1 => d,
        out1 => FLOAT,
        out2 => q
);
-- VIOLATION. Unit bound to instance “U_bad” has a
-- unconnected output port “out1”
-- unused output port “out1”, that should be
-- actively driven by another signal or by fixed ‘0’ or ‘1’
```

2.13 Declare Objects Before Use (CP13)

Objects should be declared before use. All objects used must be declared. Note: While this sort of error will be caught downstream in compilation, it is useful to check for it as early as possible as it is a very common mistake that results in misinterpreted code.

Default severity: Error

Example:

```
ENTITY top IS GENERIC (Speed: SpeedType := typical;….);
                        -- ‘SpeedType’ is an unknown type
        PORT(
        . . .
        );
```

2.14 Avoid Unused Declarations (CP14)

All declared objects should be used. Check for objects that have been declared, but are never used (i.e., read from or assigned to). Unused declared objects are considered dead code. Dead code can be detrimental when a design is reused. The dead code could be inadvertently activated during the code base port.

Default severity: Warning

Example:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY unuseddecl IS
PORT (
        in1 : IN std_logic_vector (3 DOWNTO 0);   -- Violation
                                                  -- in1(3 DOWNTO 1)not used in RTL code
        out1,                                     -- Violation out1 not used in RTL code below,
        out2 : OUT std_logic_vector(3 DOWNTO 0) -- Violation
        -- out2(2 DOWNTO 0) not used in RTL code
);
END;
ARCHITECTURE unuseddecl_rtl OF unuseddecl IS
SIGNAL temp1, temp2 : std_logic;
BEGIN
        temp1 <= in1(0);            -- Violation Signal 'temp1' is never used
        out2(3) <= temp2;           -- Violation Signal 'temp2' is never assigned
END;
```

3 Clock Domain Crossing (CDC)

This set of standards addresses potential hazards with designs containing multiple clock zones and asynchronous clock zone transitions.

3.1 Analyze Multiple Asynchronous Clocks (CDC1)

Any time a design has multiple asynchronous clocks, or if internally generated clocks are allowed, a thorough clock domain crossing (CDC) analysis should be done.

Improper clock domain crossings result in metastability or indeterminate circuit operation, which can have serious adverse effects on a device’s operation. This design guidance needs to be mentioned, even though clock domain crossing issues and analysis is beyond the scope of typical HDL linting tools and beyond the scope of this document.

As part of the design review process, all digital designs should be reviewed for potential clock domain crossing boundaries, and, when found, analyzed to verify that they are properly addressed by a synchronizer circuit. This challenging design methodology should be identified due to the extreme difficulties associated with isolating and debugging the intermittent CDC-error-induced functional behavior at the hardware level. It is often impossible to consistently duplicate these CDC-error induced behaviors in hardware for design debug. Making this situation worse, circuit elements are affected by the operating conditions, such as loading, transistor junction temperature, voltage rails and ground bounce. In a worst-case scenario, a CDC error might not exhibit itself during normal testing conditions, but it will occur during infield (i.e., in-flight) operating conditions given the right set of environmental factors for the targeted system application.

4 Safe Synthesis (SS)

The following standards are checked to ensure a proper netlist is created by the synthesis tool.

4.1 Avoid Implied Logic (SS1)

Do not allow coding that implies feedthroughs, delay chains, and internal tri-state drivers.

Certain coding styles that are dependent on implied synthesis constructs can be dangerous. This implied logic might prevent the design code base from being synthesized in a consistent manner across different device technologies. This sort of implied logic includes feedthroughs, delay chains, and internal tri-state drivers. Delay chains are two or more consecutive components with a single fan-in and single fan-out, such as inverters. In some cases, such as tri-state assignments and small width counters, this could be allowed, and the applicant should document and justify the warning in these cases.

Default severity: Warning

Example:

```
SIGNAL mode : std_logic;                          -- Internal Tri-state Control
SIGNAL tri_bus : std_logic_vector (1 DOWNTO 0);   -- Internal signal (not top level port)
…
Tristate_Control: PROCESS (mode)
BEGIN
        IF (mode = '0') THEN
                tri_bus <= "0Z"; -- Do not allow internal tristates
        ELSE
                tri_bus <= "Z0"; -- Do not allow internal tristates
        END IF;
END PROCESS Tristate_Control;
```

Example:

```
ENTITY feed_through_ea IS
PORT(
        a_i : IN std_logic;
        av_i : IN std_logic_vector (10 DOWNTO 0);
        x_o : OUT std_logic
);
END feed_through_ea;
ARCHITECTURE rtl OF feed_through_ea IS
BEGIN
        x_o <= a_i;         -- Violation, feed-through from input port “a_i” to output port “x_o”
```

Example:

```
ENTITY delay_ea IS
GENERIC (ADR_WIDTH : INTEGER := 8);
PORT(
        adr_i           : IN      std_logic_vector(ADR_WID TH-1 DOWNTO 0);
        av_i            : IN      std_logic_vector(10 DOWNTO 0);
        adrx_o          : OUT     std_logic_vector(ADR_WIDTH-1 DOWNTO 0)
);
END delay_ea;
ARCHITECTURE rtl OF delay_ea is
signal adr_inv1_s : std_logic_vector(ADR_WIDTH-1 DOWNTO 0);
signal adr_inv2_s : std_logic_vector(ADR_WIDTH-1 DOWNTO 0);
BEGIN
        adr_inv1_s <= NOT(adr_i);
        adr_inv2_s <= NOT(adr_inv1_s);
        adrx_o <= adr_inv2_s; -- Violation, delay chain from input port “adr_i” to output port “adrx_o”
```

4.2 Ensure Proper Case Statement Specification (SS2)

Case statements should:

- Be complete
- Never have duplicate/overlapping statements
- Never have unreachable case items
- Always include the “when others” clause

Case statements need to define operations for all enumerations of the case variable. Incomplete case statements will not synthesize in a predictable manner. Thus, when this logic is driven by a non-predefined input pattern (e.g., during ground bounce condition or SEU), its functional behavior will be dependent on the synthesis and place-and-route tools’ optimization algorithm settings, the release version, and even the computing environment used to execute the downstream tool runs.

Default severity: Error

Example:

```
CASE addr IS
        WHEN "000" =>
                clk_div_en <= '1';
        WHEN "001" =>
                clk_div_en <= '1';
        WHEN "000" =>                   -- Duplicate/overlapping case specification
                clk_div_en <= '1';      -- Incomplete case specification
        WHEN "10X" =>                   -- Not reachable case specification
                xmitdt_en <= '1';
                ser_if_select <= addr(1 DOWNTO 0);
        WHEN "110" =>
                ser_if_select <= addr(1 DOWNTO 0);
        WHEN "111" =>
                clr_int_en <= '1';      -- Missing WHEN OTHERS clause
END CASE;
```

4.3 Avoid Combinational Feedback (SS3)

Do not allow reading and assigning to the same signal in the sa me combinational process. Such combinational feedback paths cause race conditions, making the design's functional behavior unpredictable. This design check could also be referred to as "asynchronous feedback loops."

Default severity: Error

Example:

```
fred_s <= gated_in_s OR pulse_r;        -- Violation combinational feedback
                                        -- loop, at fred_s
P_GATED_IN : PROCESS(
        en_i,
        fred_s,
        pulse_r)
BEGIN
        gated_in_s <= fred_s AND en_i; -- Violation, async. feedback loop
        IF (en_i = '0') THEN
                gated_in_s <= NOT(fred_s); -- Violation, async. feedback loop
        ELSIF (pulse_r <= '1') THEN
                gated_in_s <= '0';
        END IF;
END PROCESS P_GATED_IN;
```

4.4 Avoid Latch Inference (SS4)

The HDL coding style should avoid inference of latches.

Careful consideration should be given to the creation of latches in a design. In many cases latches may be introduced unintentionally, due to coding errors. The HDL coding style should avoid inference of latches. While latches might simulate properly, the synthesized hardware behavior may not match functional simulation.

To avoid this problem, a good HDL coding practice is to write every IF-statement ending with an ELSE to avoid unnecessary latches in the design, as a result of synthesis inference of the IF-statement functional behavior. This does not apply to the IF-statement for the clock. A non-used ELSE-statement should be declared with a NULL.

If a project decides not to allow any asynchronous design, this should be considered an Error. However, if it is acceptable in certain cases, the severity should be considered Warning, and the applicant should document and justify each case.

Default severity: Warning

Example:

```
library ieee;
use ieee.std_logic_1164.all;
ENTITY vhdlatch IS
PORT (
        in1, in2, in3, in4  : IN     std_logic;
        out1                : OUT    std_logic;
        out2                : OUT    std_logic_vector(3 DOWNTO 0));
END;
ARCHITECTURE arch OF vhdlatch IS
BEGIN
        PROCESS (in1, in2, in3, in4) -- Violation
        BEGIN
                IF( in4 = '0') THEN
                        out2(3) <= in1;
                        out2(0) <= in2;
                ELSE
                        out2 <= (others => in3);
                END IF;
        END PROCESS;
END;
```

4.5 Avoid Multiple Waveforms (SS5)

Only one waveform should exist on the right-hand side of a signal assignment.
A waveform consists of an assignment value expression and an optional assignment delay expression. Multiple waveforms are non-synthesizable. With multiple waveforms, synthesized hardware behavior will not match simulation.

Default severity: Error

Example:

```
nRW <= '1' AFTER clk_prd, '0' AFTER 8 * clk_prd;
        -- Violation, Only first waveform element used for synthesis
```

4.6 Avoid Multiple Drivers (SS6)

The same signal/variable should be assigned in only one sequential block. A signal/shared variable should not have multiple, simultaneously active drivers. Multiply-driven signals can synthesize as “bucking drivers” in the hardware implementation and exhibit latent in-field failures of these circuits, which may become permanent.

Default severity: Error

Example:

```
library ieee;
use ieee.std_logic_1164.all;
ENTITY top IS
PORT(
        a1, a2 : IN std_logic_vector(7 DOWNTO 0);
        a3 : OUT std_logic_vector(7 DOWNTO 0));
END;
ARCHITECTURE rtl OF top IS
BEGIN
        operate_proc: PROCESS(a1)
        BEGIN
                a3 <= a1 and "10101000"; -- Violation
        END PROCESS;
        test_proc: PROCESS(a2)
        BEGIN
                a3 <= a2; -- Associated Violation
        END PROCESS;
END rtl;
```

4.7 Avoid Uninitialized VHDL Deferred Constants (SS7)

Ensure all VHDL deferred constants are initialized.

A constant can be declared in a package without an initialization. The corresponding package body should contain the initialization. Constants defined using this style are called deferred constants. It is important to ensure that VHDL deferred constants do actually get initialized. When VHDL deferred constants are not initialized, they may not be synthesizable.

Default severity: Warning

Example:

```
PACKAGE trafficPackage IS
        CONSTANT MaxTimerVal: integer
        –- Violation. Deferred constant ‘MaxTimerVal’ without initial
        -- value may not be synthesizable
END trafficPackage;
```

4.8 Avoid Clock Used as Data (SS8)

Clock signals should not be used in a logic path that drives the data input of a register.

Clock signals used as data can result in potential race conditions. This standard would also be violated in the case where a clock signal happens to be a primary output of the design (that is, output of the top design unit), since it can potentially be used as data outside the design.

While synthesis engines should catch this issue, it may only be shown as a "warning" in the synthesis environment, and therefore could easily be overlooked. Thus, the recommended severity is error. This ensures the issue is caught and addressed.

Default severity: Error

Example:

```
P_GATED_IN : PROCESS(in1, mclk)
BEGIN
        gated_in_s <= '0';
        IF (in1 = TRANSITION) and (mclk = '0') THEN        -- Associated Violation
                gated_in_s <= '1';                          -- See below
        END IF;
END PROCESS P_GATED_IN;
P_PULSE_FF : PROCESS(mclk, rst_n)          -- Violation clock used as data
BEGIN                                      -- Race condition can occur here
        IF (rst_n = '0') THEN
                pulse_r <= '0';
        ELSIF rising_edge(mclk) THEN
                pulse_r <= gated_in_s;
        END IF;
END PROCESS P_PULSE_FF;
```

4.9 Avoid Shared Clock and Reset Signal (SS9)

The same signal should not be used as both a clock and reset signal.

Shared clock and reset functions for the same signal will cause race conditions in the design. In general, clock signals should be coded in HDL in a manner to help the downstream tools synthesize and map them onto the dedicated clock routing resources. Similarly, the reset signals should be coded in HDL in a clear manner to insure that they are mapped onto dedicated reset routing resources. This will mitigate potential timing violations due to different routing implementations and design code base ports targeting a different device.

Default severity: Error

Example:

```
reset_n <= mclk AND en_i;                  -- reset_n signal has embedded clock
P_FRED_R : PROCESS(mclk, reset_n)
```

```
BEGIN
    IF (reset_n = '0') THEN             -- Violation shared clock & reset signal
        fred_r <= '0';
    ELSIF rising_edge(mclk) THEN
        fred_r <= in1(DIR_BIT);
    END IF;
END PROCESS P_FRED_R;
```

4.10 Avoid Gated Clocks (SS10)

Data signals should not be used in a logic path that drives the clock input of a register.

Gated clocks can cause clock skews and are sensitive to glitches, due to propagation delay and SET (Single Event Transient) effects, which can lead to incorrect data being registered. Coding should be checked for the usage of gated clocks throughout the design. Disallowing clock gating is good safety-critical design best practice for synchronous digital designs. A design based on synchronous enabled registers should be used instead of gated clocks.

If the project's standards determine it is never OK to use clock gating, then the severity should be set to Error. However, the suggested severity is generally Warning because in some cases (e.g., battery-power devices), clock gating is intentionally designed in to save power. When allowed, each implementation and justification must be documented thoroughly. Careful consideration of potential adverse effects when halting and starting up the associated circuit block(s) must be reviewed.

Additionally, special consideration for the targeted FPGA device technology must be taken into account. Clock gating designs for FPGA should not be allowed if the targeted FPGA device does not contain special purpose-built clock gating circuitry in silicon.

Default severity: Warning

Example:

```
clk_s <= mclk AND en_i;                 -- Gating mclk as clk_s
P_PULSE_FF : PROCESS(clk_s, rst_n)      -- Violation gated clock
BEGIN
    IF (rst_n = '0') THEN
        pulse_r <= '0';
    ELSIF rising_edge(clk_s) THEN
        pulse_r <= in1(DIR_BIT);
    END IF;
END PROCESS P_PULSE_FF;
```

4.11 Avoid Internally Generated Clocks (SS11)

Internally generated clocks should be avoided.

Clocks, clock-trees, and signals crossing clock-domains need careful consideration and handling. If internally generated clocks are allowed in a design, clock-domain crossing (CDC) analysis should be run to detect metastable design errors.

If allowed, each implementation must be justified and thoroughly documented. In FPGA targeted designs, the potential for CDC and propagation delay variance induced errors can be mitigated by requiring these clock signals to be generated with the FPGA's special purpose-built clock management circuitry, global clock drivers and routing.

Default severity: Warning

Example:

```
ARCHITECTURE rtl OF top IS
BEGIN
U_and: gate port map (a => clk_master, b => enable,
                      c => clk_c); --Isolated Clock Generator instance
synch_PROC: PROCESS (clk_c, rst_master)
BEGIN
      IF (rst_master = '0') THEN
            …
      ELSIF (rising_edge(clk_c)) THEN
            -– VIOLATION: Internally generated clock signal “clk_c” used to drive
            synchronous block “synch_PROC”, due to default rule configuration is
            Disallowed. Should the project team (or company) wish to allow
            isolation at top-level, they could change the setting or severity of
            this rule and it would not violate the example above.
```

4.12 Avoid Internally Generated Resets (SS12)

Unless they are properly isolated, internally-generated resets should be avoided.

Internally generated resets should only be allowed throughout the design if they are at the top-level of the design hierarchy and properly isolated. For synchronous digital designs, it is considered best practice to use asynchronous reset signals, which are synchronously released. This avoids race condition problems when the reset signal is de-asserted too close to the active edge of the clock, violating set-up or hold time.

In FPGA targeted designs, internally generated reset propagation delay variance, due to routing differences, can induce timing violations. This can be mitigated by generating the reset signals with the FPGA's special purpose-built reset management circuitry, global reset drivers and routing.

Default severity: Warning

Example:

```
ENTITY reset_module IS
PORT(
        clk: IN std_logic;
        rst: IN std_logic;
        in1: IN std_logic;
        out1:OUT std_logic);
END reset_module;
ARCHITECTURE rtl OF reset_module IS
SIGNAL rst_int: std_logic;
BEGIN
        PROCESS(clk) -- Violation, rst_int
        BEGIN -- is internally generated
                IF rising_edge(clk) THEN
                        IF (rst = '1') THEN
                                rst_int <= '0';
                        ELSE
                                rst_int <= in1;
                        END IF;
                END IF;
        END PROCESS;
        PROCESS(clk)
        BEGIN
                IF rising_edge(clk) THEN
                        IF (rst_int = '1') THEN -- Associated Violation
                                out1 <= '0';
                        ELSE
                                out1 <= in1;
                        END IF;
                END IF;
        END PROCESS;
END rtl;
```

4.13 Avoid Mixed Polarity Reset (SS13)

The same reset signal should not be used with mixed styles or polarities.
Reset signals should be used consistently throughout the entire design. This is important for a safety-critical program from a design IP reuse and comprehension aspect. Applying a reset signal inconsistently can cause unintended circuit behaviors during the reset signal’s assertion and de-assertion when the designer does not fully comprehend the interactions between active circuit blocks and the reset blocks. If this is intentionally implemented in the design, its justification should be documented.

Unintended design behavior can occur due to potential race conditions.

Default severity: Warning

Example:

```
ARCHITECTURE rtl OF top IS
BEGIN
    proc1: PROCESS(clk_master, clk_n, rst_master)
    BEGIN
        IF rising_edge(clk_master) THEN
            IF (rst_master = '1') THEN -- Violation, inconsistent
                q <= '0'; -- reset polarities & style
            ELSE
                q <= d1;
            END IF;
        END IF;
        IF (rst_master = '0') THEN -- Violation, inconsistent
            q <= '0'; -- reset polarities & style
        ELSIF (falling_edge(clk_n)) THEN
            q <= d2;
        END IF;
    END PROCESS;
END rtl;
```

4.14 Avoid Unresettable Registers (SS14)

All registers should have a reset control.
All registers in the design should have an explicit reset of a specified type. A reset signal is used to help initialize a device into a predetermined condition. This system initiated error recovery command is used as the last resort recovery mechanism for a non-responsive device.

There are occasional situations where the registers are not implemented with a reset control intentionally (e.g., synchronizer flops and scan-chains). In these cases, the applicant should document and justify each exception. Note that multi-dimensional signals/register declarations modeled as memories should be exempt from this check.

Default severity: Warning

Example:

```
PROCESS (clk, reset) –- Violation for out3, missing reset control
BEGIN
    IF(reset = '1') THEN
        out2 <= (others => '0');
    ELSIF(rising_edge(clk)) THEN
        out2 <= in2;
        out3 <= in1;
    END IF;
END PROCESS;
```

4.15 Avoid Asynchronous Reset Release (SS15)

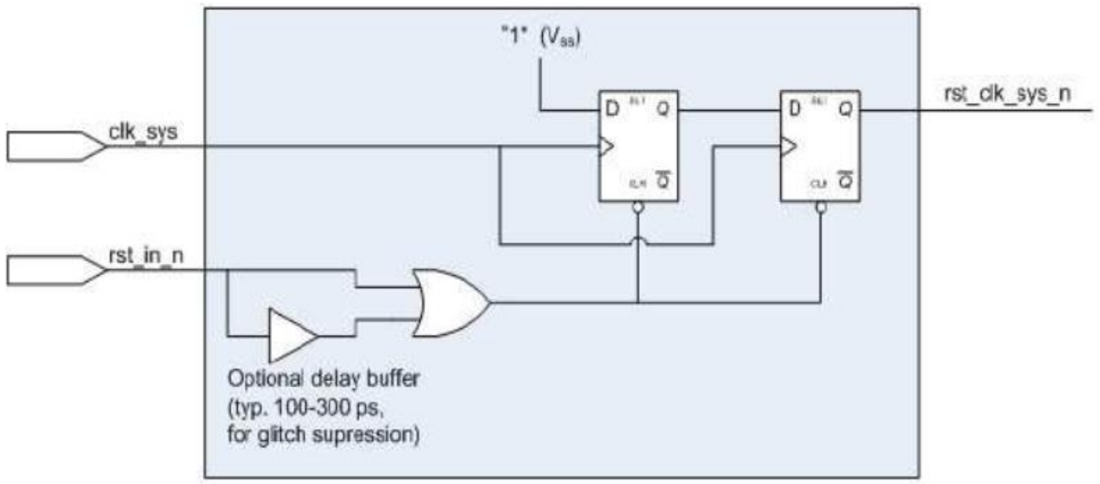
Reset signals should have a synchronous release.

For synchronous digital designs, it is considered best practice to generate reset control as asynchronous assertion and synchronous de-assertion signal to avoid problems when the reset signal is de-asserted during the active edge of the clock.

Default severity: Error

Example:

Note that the following figure demonstrates a correct on-chip reset scheme as described in the preceding text.



4.16 Avoid Initialization Assignments (SS16)

Do not use register initialization assignments.

It is best practice to use an explicit reset to set a register to a value as opposed to using an initialization assignment in the port/signal declaration. It is desirable to allow uninitialized signals to simulate as unknown, to detect and evaluate the X propagation through the design. This may reflect actual design issues in-hardware.

Default severity: Error

Example:

```
library ieee;
use ieee.std_logic_1164.all;
PROCESS (clk, reset) –- Violation for out3, missing reset control
BEGIN
        IF(reset = '1') THEN
                out2 <= (others => '0');
        ELSIF(rising_edge clk)) THEN
                out2 <= in2;
```

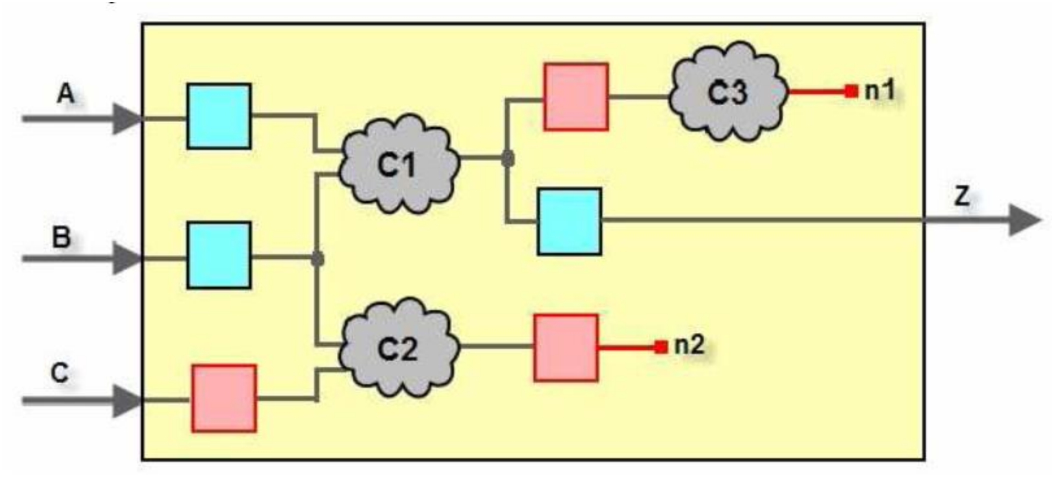
```
                    out3 <= in1;
            END IF;
    END PROCESS;
```

4.17 Avoid Undriven and Unused Logic (SS17)

- Every register and latch must be used and driven in the design.
- Registers and latches affecting only unused logic must be examined.

Unused registers, latches, and other logic are simply dead code. Dead code can be detrimental when a design is reused, and the dead code is inadvertently activated during the code base port. Dead code effects on safety-critical design assurance is not redactable nor insured to be consistent across different synthesis tools, release versions, and even computing environments.

Default severity: Error
Example:



As seen from this figure, n1 and n2 are not used in the design and hence constitute unused logic violations. Moreover, the registers in red are only feeding nets that are unused, and consequently those registers are unused as well. However, the other (blue) registers are used since they drive output Z and hence, they should be allowed, even though they also drive unused logic n1 and n2.

4.18 Ensure Register Controllability (SS18)

Each register should be controllable from its inputs.
All registers/latches (state bits) must be controlled from their inputs. A state bit might suffer from a dead-end value, which means that the bit drives a constant unless an asynchronous reset is applied. A state bit might also suffer from a stuck-at-fault, which

means that the register always drives a constant value, irrespective of any changes to the inputs, including resets. These situations can lead to inconsistent synthesis results.

Default severity: Warning

Example:

```
flag_s <= (OTHERS => ’0’);
. . .
P_CMD_R : PROCESS(rst_n, mclk)
BEGIN
        IF (rst_n = ’0’) THEN
                cmd_r <= (OTHERS => ’0’);
        ELSIF RISING_EDGE(mclk) THEN
                cmd_r <= request_in AND flag_s;  -- violation, stuck-at-0 fault
        END IF;
END PROCESS P_CMD_R;
```

4.19 Avoid Snake Paths (SS19)

Combinational paths should not exceed a maximum allowable logic depth.
It is difficult for timing optimizers to optimize the critical path of a design, if it spans multiple levels of hierarchy. Long combinational paths, also called snake paths, can pose synthesis problems. The project team should determine the maximum levels of permissible user hierarchy level that a path can span. Any path longer than this should be reported as a violation and should be broken at an appropriate point by inserting registers.

Default severity: Warning

4.20 Ensure Nesting Limits (SS20)

Conditional branching constructs should have a maximum nesting depth.
Deeply nested conditional branching logic will tend to synthesize with longer propagation delay depths, in addition to being more complex to debug and fix when an error has been found. The project teams should specify, and then check for, a maximum limit for nesting depth for conditional branching constructs. Controlling the nesting depth and format improves readability and reduces complexity, which is desirable to insure that the code base will more likely implement a fully verifiable design that can also be modified quickly for design reuse and debugging. All of these aspects are important in a fail-safe design. The nesting limit can be set at the discretion of the project team, taking into account logic propagation delays’ negative effect on the design’s timing requirements. Violations should be examined to see if they are acceptable or require re-design.

If the design requires the use of a logic implementation exceeding the maximum allowable nesting limit, then this violation and its impact study should be documented.

Default severity: Warning

Example:

```
FLIP_FLOP: PROCESS(rst,clk)
BEGIN
IF rst = '1' THEN
        qout <= '0';
        out_one <= '0';
        out_two <= '0';
        out_three <= '0';
ELSIF RISING_EDGE(clk) THEN
        IF in_one = '1' THEN
                out_one <= in_one;
                IF in_two = '1'THEN
                        out_two <= in_two;
                        IF in_three = '1' THEN -- Violation if set to 3, as 4th level
                                out_three <= in_three;
        ...
```

4.21 Ensure Consistent Vector Order (SS21)

Use the same multi-bit vector order consistently throughout the design.
In order to promote code comprehension and reduce potential interconnection errors, the multi-bit vector order should be consistent throughout the design. The vector range should consistently use either an ascending or descending order. Note that violations often occur when a design uses reused code or IP. But each of these violations should be examined and justification/assurance should be given that the issue is appropriately addressed in the design.

Default severity: Warning

Example:

```
Bus_ascending : IN std_logic_vector (7 DOWNTO 0);
Bus_decending : IN std_logic_vector (0 T0 7); -- Violation if Descending order enabled
```

5 Design Reviews (DR)

The following standards are checked to make design reviews and code comprehension easier.

5.1 Use Statement Labels (DR1)

Statements should have labels.

Case Statements, Processes, Always Blocks, and Initial Blocks should have labels to improve clarity for design code reviews and tracing during simulation. Labels should be added to the End constructs to assist with following the scope of the code.

Default severity: Note

Example:

```
ARCHITECTURE flow OF top IS
BEGIN
        -- Architecture concurrent statements
        data_out <= div_data WHEN clk_div_en = '1' ELSE ser_if_data;
        -- Violation. Concurrent statements should be labeled
END flow;
```

5.2 Avoid Mixed Case Naming for Differentiation (DR2)

Names should not be differentiated by case alone.

Note that this is different than mixed case naming for a single object. Mixed case naming is sometimes preferred for clarity/readability. However, when design code uses the same name for two different objects, with the only difference being character case, this should not be allowed. Violating this coding standard will cause confusion regarding the design's intention. It is a bad coding practice with respect to design comprehension for a safety-critical application.

Default severity: Warning

Example:

```
ENTITY top IS
        PORT (nrw: OUT std_logic );
END top;
ARCHITECTURE flow OF top
BEGIN
        Nrw <= ‘0’;
        -- Violation. Do not allow mixing of case identifier “nrw”
        -- Identifiers “nrw” and “Nrw” are differentiated by case only
END
```

5.3 Ensure Unique Name Spaces (DR3)

The same name should not be used for different types of identifiers.

The same name is not used in different name spaces. For example, the same name should not be used to identify a signal in one part of the design but also used as a process label elsewhere. Port names having the same names as connected signals can be excluded.

At best, violating this standard will cause confusion regarding the design's intention. At worst, the design could pass synthesis without any violation but its in-hardware functionally will not follow the HDL code's intention.

Default severity: Error

Example:

```
ARCHITECTURE rtl OF top IS
SIGNAL i : std_logic;
-- Violation. Do not use identifier “i” since it exists in
-- another namespace within the same design
PROCEDURE my_and (in1 : std_logic; in2 : std_logic; out1 : OUT
std_logic) is
        VARIABLE i : std_logic;    -- Identifier name “i” already used in another namespace
BEGIN
                i := '1';
                out1 <= in1 AND in2;
END PROCEDURE;
```

5.4 Use Separate Declaration Style (DR4)

Each declaration should be placed on a separ ate line.
Having declarations on separate lines dramatically improves code reviews and readability. Thus declarations should be formatted with each on a separate line or, in the case of ports, with each group of ports of the same mode (input, output etc) on a separate line.

Default severity: Note

Example:

```
ARCHITECTURE spec OF status_registers IS
        SIGNAL xmitting_r, done_xmitting_r : std_logic; -- declaration
        -- Violation. Multiple signals declared in one line,
        -- declarations should be on separate lines.
END spec;
```

5.5 Use Separate Statement Style (DR5)

Each statement should be placed on a separate line.
Having statements on separate lines ensures readability of the code. Declarations are excluded unless there is a declaration on the same line as another statement. Note that the “begin” keyword can be removed from this check.

Default severity: Note

Example:

```
IF (en_i = '1') THEN
```

```
        x1_s <= NOT(a1_i); x2_s <= a1_i;  -- Violation, multiple statements
ELSE
        x1s <= '0'; x2_s <= '0';          -- Violation, multiple statements
END IF;
```

5.6 Ensure Consistent Indentation (DR6)

Code should be consistently indented.

In order to promote readability of code between different editors, code should be consistently indented in terms of spaces and the number of indentation steps.

Default severity: Note

Example:

```
FLOP_FLIP: PROCESS(rst,clk) –- Consistently formatted
BEGIN
        IF rst = '1' THEN
                tout_one <= '0';
                tout_two <= '0';
                tout_three <= '0';
        ELSIF rising_edge(clk) THEN
                IF in_one = '1' THEN
                        tout_one <= in_one;
                        IF in_two = '1' THEN
                                tout_two <= in_two;
                                IF in_three = '1' THEN
                                        tout_three <= in_three;
                                ENDIF;
                        ENDIF;
                ENDIF;
        ENDIF;
```

5.7 Avoid Using Tabs (DR7)

Tabs should not be used.

Tabs should not be used, as they can result in problems when moving source code from one editing environment to another.

Default severity: Warning

5.8 Avoid Large Design Files (DR8)

Designs should be partitioned, and files should be of limited size.

In order to promote code comprehension, a design file should be of a limited size to promote better design partitioning. This will help force the design structure into a more hierarchical partition away from a large flat code structure. The acceptable size should be at the discretion of the project team.

Default severity: Warning

5.9 Ensure Consistent Signal Names Across Hierarchy (DR9)

Signals and busses should have consistent names when they span the design hierarchy.

In order to promote code comprehension and reduce potential interconnection errors, a signal/bus should have a constant name when it spans across hierarchical levels.

Default severity: Warning

5.10 Ensure Consistent File Header (DR10)

Ensure a consistent file header.

In order to promote code comprehension, a consistent file header comment style should be enforced throughout the design project. Important information such as the code’s author, organization, creation and modification dates, legal briefing, statement of proprietary information, etc should be included if deemed appropriate. A summary of the code’s intention should be placed in the file header as well.

Default severity: Warning

5.11 Ensure Sufficient Comment Density (DR11)

Code should be sufficiently documented via inline comments.
In order to promote code comprehension, a minimum design code in-line documentation (i.e., commenting) should be enforced. The amount of design code inline commenting should allow a different designer to be able to understand the design well enough to be able to modify it for a different project in a reasonable amount of time.

Default severity: Warning

5.12 Ensure Proper Placement of Comments (DR12)

Comments should be placed in appropriate places to aid understanding.
Enforce the placement of comments near (preferably above) code statements and code blocks to aid design code functionality understanding. This design check helps to enforce stricter degrees of design code documentation/commenting for more complex language constructs usage.
Default severity: Note

5.13 Ensure Company Specific Naming Standards (DR13)

Each company or project should establish and enforce its own naming standards.

These standards will vary from company to company, or even project to project, and therefore cannot be explicitly included in a generic set of DO-254 coding standards. However, they should be considered and included in each company's HDL coding standards. The sorts of things to consider include:

- Having the component have the same name as the associated entity
- Ensuring name association between formal and actual generics, ports or parameters
- Enforcing specific filename matching with associated entity

- Enforcing specific object type naming convention, with a prefix or postfix appended to the object name. Choose only one of these two methods (prefix vs. postfix labels) and consistently apply it throughout the entire design. Consideration should be given to naming conventions for clocks, resets, signals, constants, variables, registers, FSM State Variables, generics, labels etc. For example:
    - signals use "_s"
    - registers use "_r"
    - constants use "_c"
    - processes use "_p"
    - off-chip inputs use "_I"
    - on-chip inputs use "_i"
    - off-chip outputs use "_O"
    - on-chip outputs use "_o"
    - etc.

Default severity: Note

6 Reference

1. Best Practice VHDL Coding Standards for DO-254 Programs, Michelle Lange, 9/13/10