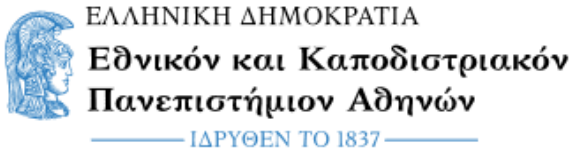


ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΔΠΜΣ Space Technologies, Applications and seRvices (STAR)

M806 Space Data Systems

Best Practice VHDL Coding for Aerospace Systems

Ακαδημαϊκό Έτος 2023-2024

Νεκτάριος Κρανίτης

DO-254 and DO-178C Standards

- DO-254 and DO-178C are standards developed by Radio Technical Commission for Aeronautics (RTCA) and adopted by regulatory bodies i.e. the Federal Aviation Administration (FAA) and European Aviation Safety Agency (EASA) for Aerospace industry
- **DO-254**, "**Design Assurance Guidance for Airborne Electronic Hardware (AEH)"**
  - Provides a structured and standardized approach to the development, verification, and validation of electronic HW used in aerospace systems
  - Covers electronic HW components, such as ICs, PCBs, and other electronic equipment, and defines a set of Design Assurance Levels (DALs) that must be met for each component based on level of safety criticality.
  - Compliance with DO-254 is required for achieving certification of electronic HW used in airborne systems
- **DO-178C**, "**Software Considerations in Airborne Systems and Equipment Certification**"
  - Provides a structured and standardized approach to the development, verification, and validation of SW used in airborne systems
  - Covers SW components, including OSs, application software, and SW tools, and defines five levels of criticality, called Software Levels (SWLs), that must be met for each component
  - Compliance with DO-178C is required for achieving certification of SW used in airborne systems.
- In summary, **DO-254 focuses on electronic HW**, while **DO-178C focuses on SW**, and both standards are critical for ensuring the safety and reliability of airborne systems

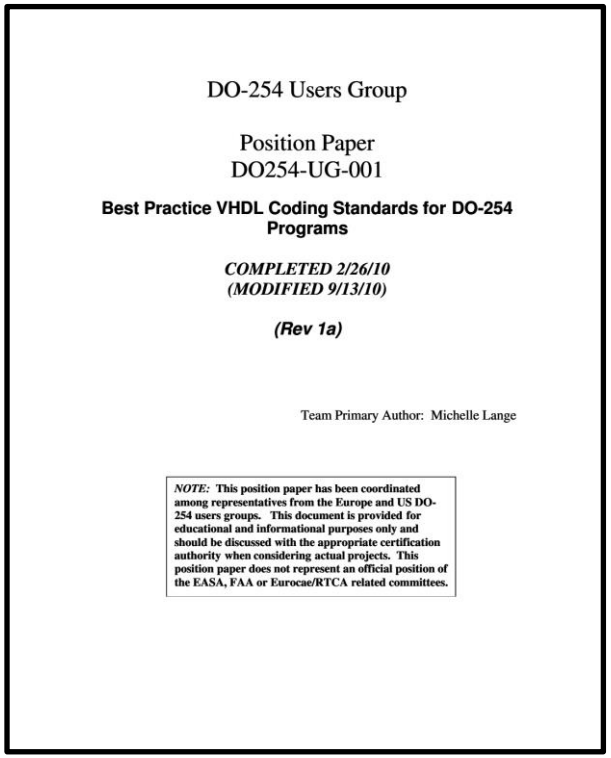
VHDL coding for DO-254 and DO-178C Compliance

- VHDL design with DAL A, B and C must establish conformance to VHDL coding standards
  - No official list from FAA or EASA
  - Guidance & list of rules proposed by DO-254 UG (2010)
- RTCA/DO-254 provides design assurance guidance for airborne electronic HW
  - Ensure that AEH works reliably as specified, avoiding faulty operation and potential functional hazards
  - Discusses need for “Design Standards”
- FAA Order 8110.105 section 6-2a clarified that HDL coding standards should be defined and checked when it stated:
  - *“To prevent potentially unsafe attributes of HDLs from leading to unsafe features of the components, we must expect that, if they use an HDL, applicants define the coding standards for this language consistent with the system safety objectives, and establish conformance to those standards by HDL code reviews.”*

DAL	Failure Condition	Resulting Conditions
Level A	Catastrophic	Failure may result in deaths and loss of the aircraft
Level B	Hazardous	Failure creates a major negative impact on safety or performance or reduces the aircraft crew’s ability to operate the aircraft. This can result in serious or fatal injuries.
Level C	Major	Failure causes significant reduction of the safety margin or significant increase in the aircraft crew workload. Passenger discomfort or minor injuries can result.
Level D	Minor	Failure slightly reduces the margin of safety or causes slight increase in aircraft crew workload. Results can include passenger inconvenience or changes to a routine flight plan.
Level E	No Effect	Failure causes no impact or effect on safety, crew workload, or operation of the aircraft.

DO-254 UG Positioning Paper

- Provide a list of generally accepted **VHDL design best practice coding guidelines** that should be considered for a fail-safe design, including DO-254 programs
- These coding guidelines should NOT be viewed as what must be done in a DO- 254 program
  - What must be done is always the decision of the applicant in conjunction with the certification authority
  - However, if project team is looking for a good foundational set of checks to assess the HDL design quality for their DO-254 program, this document provides that foundation



DO-254 Users Group

Position Paper
DO254-UG-001

Best Practice VHDL Coding Standards for DO-254 Programs

COMPLETED 2/26/10
(MODIFIED 9/13/10)

(Rev 1a)

Team Primary Author: Michelle Lange

NOTE: **This position paper has been coordinated among representatives from the Europe and US DO-254 users groups. This document is provided for educational and informational purposes only and should be discussed with the appropriate certification authority when considering actual projects. This position paper does not represent an official position of the EASA, FAA or Eurocae/RTCA related committees.**

DO-254 UG Positioning Paper

- Following 4 Rule Categories Were Proposed
  - Coding Practices [14 Rules CP1 – CP14]
    - This set of rules ensures that a coding style supporting safety-critical and good digital design practices are used
  - Safe Synthesis [21 Rules SS1-SS21]
    - This set of rules ensure that a proper netlist is created by the synthesis tool.
  - Code Reviews [13 rules DR1 – DR13]
    - This set of rules are checked to ease design reviews & code comprehension
  - Clock Domain Crossings [1 Rule CDC1]
    - This rule addresses potential hazards with designs containing multiple clock zones and asynchronous clock zone transitions

DO-254 Category: Coding Practices (CP)

- This category 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

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_WIDTH-1 DOWNTO 0); --32-bit
    cpu_clk  : IN std_logic;
```

DO-254 Category : Coding Practices (CP)

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
```

DO-254 Category : Coding Practices (CP)

5. **Ensure Consistent FSM State Encoding Style (CP5)**
   a. *A design should employ a consistent state encoding style for Finite State Machines (FSM).*
   b.FSM state types should not be hard-coded, unless unavoidable.

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;
```

DO-254 Category : Coding Practices (CP)

6. **Ensure Safe FSM Transitions (CP6)**
   a. *An FSM should have a defined reset state.*
   b. All unused (illegal or undefined) states should transition to a defined state, whereupon this error condition can be processed accordingly.
   c. 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;
```

DO-254 Category : Coding Practices (CP)

10. **Assign Value Before Using (CP10)**
Every object (e.g., signal, variable, port) 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_CNTR-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;
```

DO-254 Category : Clock Domain Crossing (CDC)

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

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 affects 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.

DO-254 Category : Safe Synthesis (SS)

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

1. **Avoid Implied Logic (SS1)**
   Do not allow coding that implies feed-throughs, delay chains, and internal tri-state drivers.
   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”
```

DO-254 Category : Safe Synthesis (SS)

2. **Ensure Proper Case Statement Specification (SS2)**
   Case statements should:
   a. Be complete
   b. Never have duplicate/overlapping statements
   c. Never have unreachable case items
   d. Always include the "when others" clause

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;
```

DO-254 Category : Safe Synthesis (SS)

4. **Avoid Latch Inference (SS4)**
   The HDL coding style should avoid inference of latches.

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;
```

DO-254 Category : Safe Synthesis (SS)

7. **Avoid Uninitialized VHDL Deferred Constants (SS7)**
   Ensure all VHDL deferred constants are initialized.

Default severity: Warning

Example:

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

DO-254 Category : Safe Synthesis (SS)

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.

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;
```

DO-254 Category : Safe Synthesis (SS)

9. **Avoid Shared Clock and Reset Signal (SS9)**
   The same signal should not be used as both a clock and reset signal.

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;
```

DO-254 Category : Safe Synthesis (SS)

10. **Avoid Gated Clocks (SS10)**

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

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;
```

DO-254 Category : Safe Synthesis (SS)

13. **Avoid Mixed Polarity Reset (SS13)**

The same reset signal should not be used with mixed styles or polarities.

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;
```

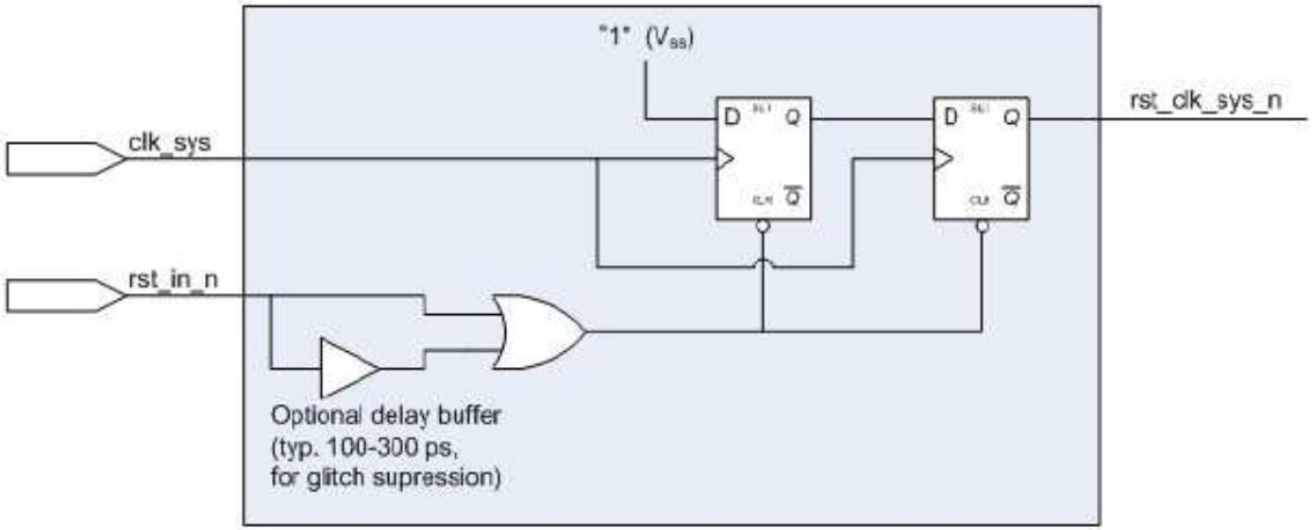
DO-254 Category : Safe Synthesis (SS)

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.



DO-254 Category : Safe Synthesis (SS)

20. **Ensure Nesting Limits (SS20)**

Conditional branching constructs should have a maximum nesting depth.

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;
..
```

DO-254 Category : Safe Synthesis (SS)

21. **Ensure Consistent Vector Order (SS21)**
Use the same multi-bit vector order consistently throughout 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
```

DO-254 Category : Design Reviews (DR)

2. **Avoid Mixed Case Naming for Differentiation (DR2)**
   Names should not be differentiated by case alone.

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
```

DO-254 Category : Design Reviews (DR)

4. **Use Separate Declaration Style (DR4)**
   Each declaration should be placed 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;
```

DO-254 Category : Design Reviews (DR)

5. **Use Separate Statement Style (DR5)**
   Each statement should be placed on a separate line.

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;
```

DO-254 Category : Design Reviews (DR)

6. **Ensure Consistent Indentation (DR6)**
   Code should be consistently indented.

   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;
```

DO-254 Category : Design Reviews (DR)

7. **Avoid Using Tabs (DR7)**
   Tabs should not be used.

Default severity: Warning

10. **Ensure Consistent File Header (DR10)**
    Ensure a consistent file header.

Default severity: Warning

11. **Ensure Sufficient Comment Density (DR11)**
    Code should be sufficiently documented via inline comments.

Default severity: Warning

DO-254 Category : Design Reviews (DR)

13. **Ensure Company Specific Naming Standards (DR13)**
Each company or project should establish and enforce its own naming standards.

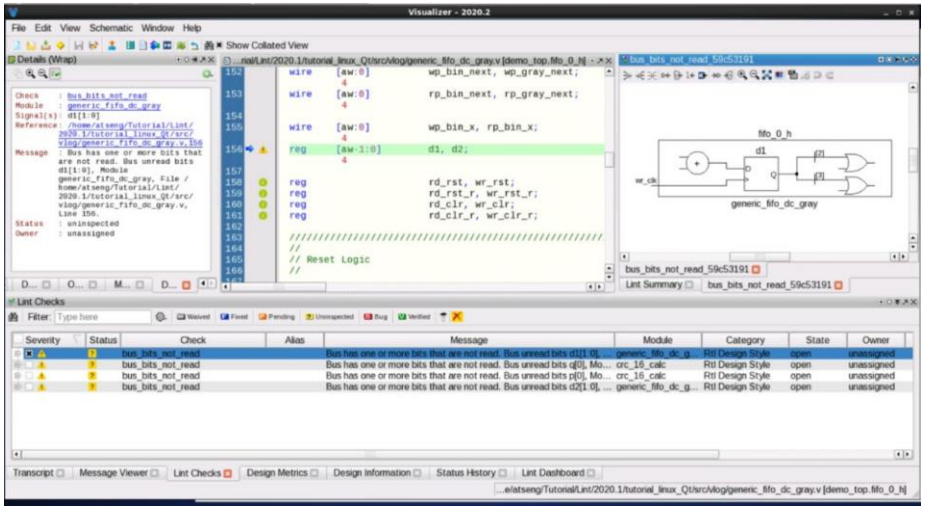
These standards will vary from company to company, or even project to project, and therefore cannot be explicitly included in a generic set of DO-254 coding standards. However, they should be considered and included in each company's HDL coding standards. The sorts of things to consider include:

- 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 through out the entire design. Consideration should be give to naming conventions for clocks, resets, signals, constants, variables, registers, FSM State Variables, generics, labels etc. For example:
    a. signals use "_s"
    b. registers use "_r"
    c. constants use "_c"
    d. processes use "_p"
    e. off-chip inputs use "_I"
    f. on-chip inputs use "_i"
    g. off-chip outputs use "_O"
    h. on-chip outputs use "_o"
    i. etc.

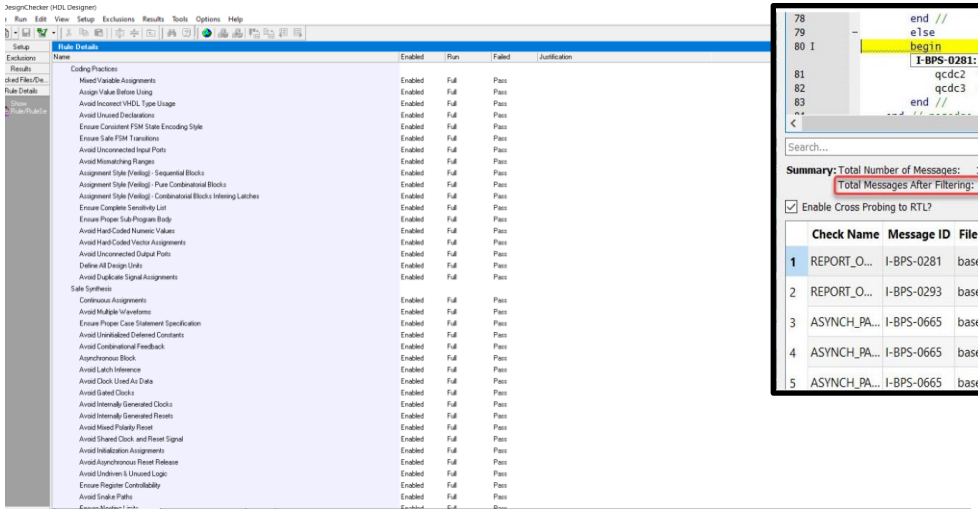
Default severity: Note

Automated Rule Checking

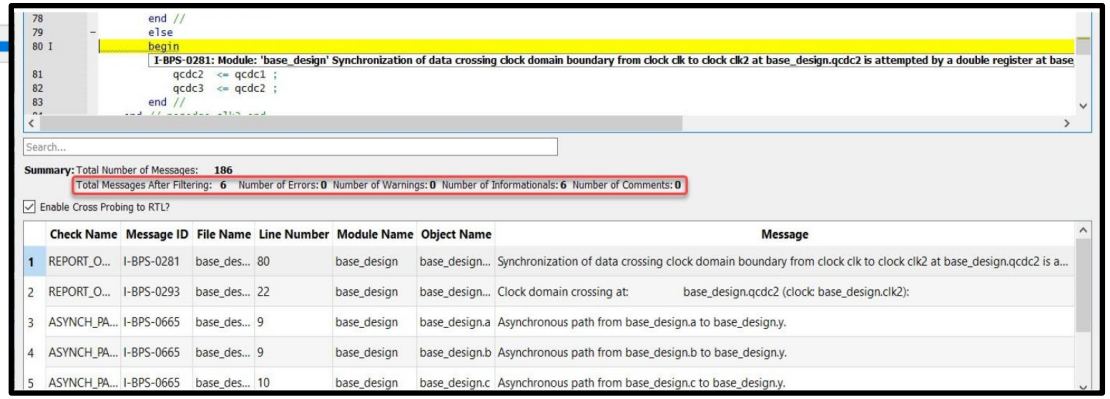
- Design reviews: can be done manually
- Automated approach (called **linting**):
  - guarantees a more consistent VHDL code quality assessment
  - has the added benefit of promoting regular VHDL design checking steps throughout the design development process, as opposed to waiting for gating design reviews where issues can be overwhelming and more costly to address



SIEMENS Questa Lint



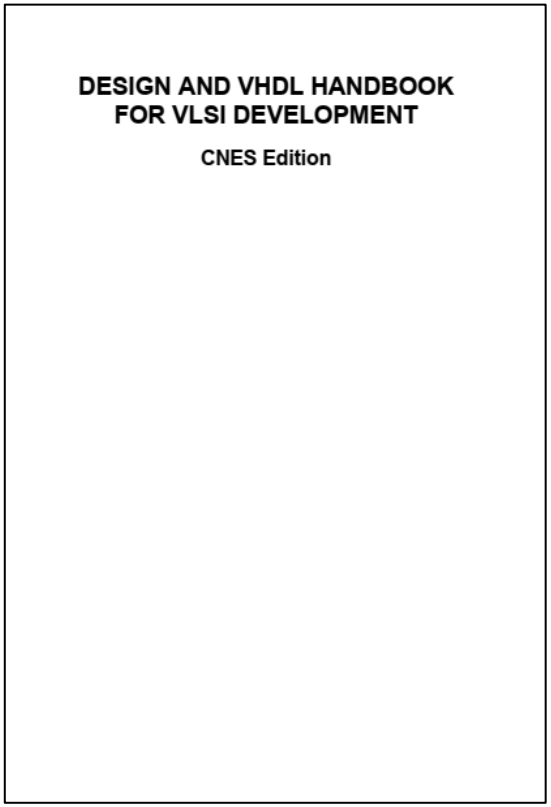
Siemens HDL Designer



BluePearl

CNES Rules

- Originally developed by CNES for internal projects
- To improve the way VHDL code is written and thus reducing the time spent for code review
- Useful for other companies that want to share common VHDL rules between them and their subcontractors
- Handbook is divided into two chapters:
  - "**Standard Rules**“: includes general rules or recommendations that are common between all companies working with VHDL. These rules share a general agreement between CNES and partners initially involved. Addition and changes to these rules have to be agreed by everyone.
  - "**Custom Rules**“: includes company specific rules that are adapted/refined from standard rules or completely new. These custom rules allow third party companies to create their own version of the VHDL Handbook.



CNES Rules

Table of Contents

CNES Standard rules: Examples

STD_00300 : Name of reset signal

STD_00300	Name of reset signal	Major
Revision	7 / 2020-04-23	
Status / Engine	Implemented / ZamiaCad	
Classification	VLSI / Formatting / Naming	
Application Field	General	
Parent Rule	STD_00100	
Description	The reset signal name includes "rst", "reset" or "clr".	

- **Detailed Description:**

  A signal is considered as a "RESET" whenever it is used inside a clocked-process to initialize signals value to a known state (most of the time zero) or mapped on a IP reset input.
  If several reset signals are used, each reset is identified with a different name.

- **Rationale:**

  The reset signal is critical. This signal needs to be easily found through the design.

- **Good Example:**

  Extracted from STD_00300_good.vhd

```
    i_Reset_n : in  std_logic;        -- Reset signal
```

CNES Standard rules: Examples

STD_00400 : Label for process

STD_00400	Label for process	Minor
Revision	5 / 2020-04-23	
Status / Engine	Implemented / ZamiaCad	
Classification	VLSI / Formatting / Naming	
Application Field	General	
Parent Rule	STD_00100	
Description	Processes are identified by a label.	

- **Detailed Description:**

  No additional information.

- **Rationale:**

  Labels improve readability of simulations, VHDL source code and synthesis logs.

- **Good Example:**

  Extracted from STD_00400_good.vhd

```
-- LABELLED PROCESS
P_FlipFlop : process(i_Clock, i_Reset_n)
begin
   if (i_Reset_n = '0') then
      Q <= '0';
   elsif (rising_edge(i_Clock)) then
         Q <= i_D;
   end if;
end process;
```

- **Bad Example:**

  Extracted from STD_00400_bad.vhd

```
-- UNLABELLED PROCESS
process(i_Clock, i_Reset_n)
begin
   if (i_Reset_n = '0') then
      Q <= '0';
   elsif (rising_edge(i_Clock)) then
         Q <= i_D;
   end if;
end process;
```

CNES Standard rules: Examples

STD_01600 : Entity port sort

STD_01600	Entity port sort
Revision	6 / 2020-04-23
Status / Engine	Validated / None
Classification	VLSI / Formatting / FileStructure
Application Field	General
Parent Rule	STD_01500
Description	Entity ports are organized by interface.

- **Detailed Description:**

  Entity ports are grouped by external interfaces.

  Within an interface group, ports could then be sorted by direction (input, output, bidirectional).

- **Rationale:**

  Ports grouped by external interfaces improves readability.

- **Good Example:**

  Extracted from STD_01600_good.vhd

```
entity STD_01600_good is
   -- We sort port by interfaces, with special ports first
   port (
      -- Special signals:
      i_Clock   : in  std_logic;        -- Clock input
      i_Reset_n : in  std_logic;        -- Reset input
      -- First Mux:
      i_A1      : in  std_logic;        -- first input
      i_B1      : in  std_logic;        -- second input
      i_Sel1    : in  std_logic;        -- select input
      o_Q1      : out std_logic;        -- Mux output
      -- Second Mux:
      i_A2      : in  std_logic;        -- first input
      i_B2      : in  std_logic;        -- second input
      i_Sel2    : in  std_logic;        -- select input
      o_Q2      : out std_logic         -- Mux output
      );
end STD_01600_good;
```

- **Bad Example:**

  Extracted from STD_01600_bad.vhd

```
entity STD_01600_bad is
   -- We sort port by name
   port (
      i_A1      : in  std_logic;        -- First Mux, first input
      i_A2      : in  std_logic;        -- Second Mux, first input
      i_B1      : in  std_logic;        -- First Mux, second input
      i_B2      : in  std_logic;        -- Second Mux, second input
      i_Sel1    : in  std_logic;        -- First Mux, selector input
      i_Sel2    : in  std_logic;        -- Second Mux, selector input
      i_Clock   : in  std_logic;        -- Clock input
      i_Reset_n : in  std_logic;        -- Reset input
      o_Q1      : out std_logic;        -- First module output
      o_Q2      : out std_logic         -- Second module output
      );
end STD_01600_bad;
```

CNES Standard rules: Examples

STD_01700 : Entity special ports

STD_01700	Entity special ports	Minor
Classification	VLSI / Formatting / FileStructure	
Application Field	General	
Parent Rule	STD_01500	
Description	Special ports are the first group of an entity.	

- **Detailed Description:**

  Special input ports like clock(s), reset and global enable are the first to be written in an entity.

- **Rationale:**

  These special signals are important for the understanding of the module functionalities. Thus gathering them at the beginning of an entity improves readability.

- **Good Example:**

  Extracted from STD_01700_good.vhd

```
entity STD_01700_good is
   port (
      i_Clock   : in  std_logic;        -- Clock signal
      i_Reset_n : in  std_logic;        -- Reset signal
      i_D       : in  std_logic;        -- D Flip-Flop input signal
      o_Q       : out std_logic         -- D Flip-Flop output signal
      );
end STD_01700_good;
```

- **Bad Example:**

  Extracted from STD_01700_bad.vhd

```
entity STD_01700_bad is
   port (
      o_Q       : out std_logic;        -- D Flip-Flop output signal
      i_D       : in  std_logic;        -- D Flip-Flop input signal
      i_Clock   : in  std_logic;        -- Clock signal
      i_Reset_n : in  std_logic         -- Reset signal
      );
end STD_01700_bad;
```

CNES Standard rules: Examples

STD_03600 : Reset sensitive level

STD_03600	Reset sensitive level	Major
Revision	6 / 2020-04-23	
Status / Engine	Implemented / ZamiaCad	
Classification	FPGA / Design / Reset	
Application Field	General	
Parent Rule	N/A	
Description	Every synchronous process uses the same reset activation level.	

- **Detailed Description:**

  No additional information.

- **Rationale:**

  In a FPGA, the reset signal is usually a high fan-out signal routed using a dedicated global signal routing track. Usin both levels of the reset signal to asynchronously reset the flip-flops of the design results in the synthesis of the res signal itself and its inverted version, which leads to the usage of 2 global dedicated routing tracks versus a single on with one of the reset signal passing through an inverter in the FPGA fabric rendering the reset recovery timings closu harder to meet for the FPGA EDA tools.

Good Example:

Extracted from STD_03600_good.vhd

```
architecture Behavioral of STD_03600_good is
   signal D_r1 : std_logic;             -- D signal registered 1 time
   signal D_r2 : std_logic;             -- D signal registered 2 times
   signal D_re : std_logic;             -- Module output
begin
   P_First_Register : process(i_Reset_n, i_Clock)
   begin
      if (i_Reset_n = '0') then
         D_r1 <= '0';
      elsif (rising_edge(i_Clock)) then
            D_r1 <= i_D;
      end if;
   end process;

   P_Second_Register : process(i_Reset_n, i_Clock)
   begin
      if (i_Reset_n = '0') then
         D_r2 <= '0';
         D_re <= '0';
      elsif (rising_edge(i_Clock)) then
            D_r2 <= D_r1;
            D_re <= D_r1 and not D_r2;
      end if;
   end process;

   o_Q <= D_re;
end Behavioral;
```

Bad Example:

Extracted from STD_03600_bad.vhd

```
architecture Behavioral of STD_03600_bad is
   signal D_r1  : std_logic;            -- D signal registered 1 time
   signal D_r2  : std_logic;            -- D signal registered 2 times
   signal D_re  : std_logic;            -- Module output
   signal Reset : std_logic;            -- Reset signal (active high)
begin
   P_First_Register : process(i_Reset_n, i_Clock)
   begin
      if (i_Reset_n = '0') then
         D_r1 <= '0';
      elsif (rising_edge(i_Clock)) then
            D_r1 <= i_D;
      end if;
   end process;

   Reset <= not i_Reset_n;

   P_Second_Register : process(Reset, i_Clock)
   begin
      if (Reset = '1') then
         D_r2 <= '0';
         D_re <= '0';
      elsif (rising_edge(i_Clock)) then
            D_r2 <= D_r1;
            D_re <= D_r1 and not D_r2;
      end if;
   end process;

   o_Q <= D_re;
```

CNES Standard rules: Examples

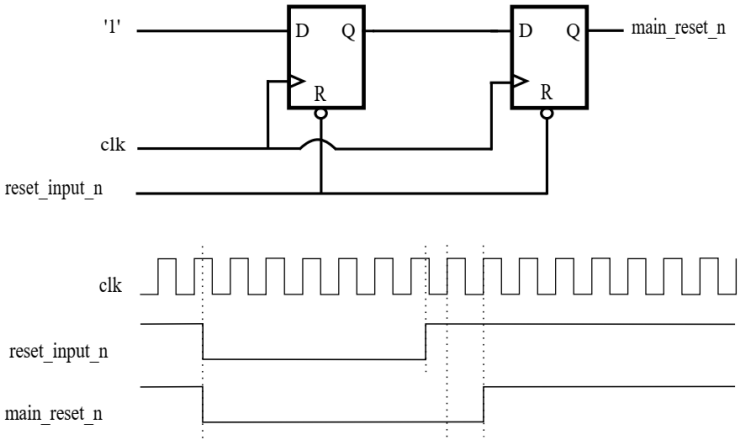
STD_03700 : Reset assertion and deassertion

STD_03700	Reset assertion and deassertion	Major
Revision	7 / 2020-04-23	
Status / Engine	Implemented / ZamiaCad	
Classification	VLSI / Design / Reset	
Application Field	General	
Parent Rule	N/A	
Description	Internal reset is asserted asynchronously and deasserted synchronously.	

- **Detailed Description:**

  If several clock domains are used then several reset signals are created to be deasserted synchronously with each targeted clock domain.

- **Figure:**



Good Example:

Extracted from STD_03700_good.vhd

```
entity STD_03700_good is
   port (
      i_Clock         : in  std_logic;  -- Clock signal
      i_Reset_Input_n : in  std_logic;  -- Reset input
      o_Main_Reset_n  : out std_logic   -- Global reset signal active low
      );
end STD_03700_good;

architecture Behavioral of STD_03700_good is
   signal Main_Reset_n   : std_logic;   -- Internal signal between FlipFlops
   signal Main_Reset_n_r : std_logic;   -- Assertion block output
begin
   P_Reset_Assert : process(i_Reset_Input_n, i_Clock)
   begin
      if (i_Reset_Input_n = '0') then
         Main_Reset_n   <= '0';         -- Output reset signal is active low
         Main_Reset_n_r <= '0';
      elsif rising_edge(i_Clock) then
            Main_Reset_n   <= '1';  -- Reset is deasserted. Since it is active low, the inactive
 value is 1
           Main_Reset_n_r <= Main_Reset_n;
      end if;
   end process;

   o_Main_Reset_n <= Main_Reset_n_r;
end Behavioral;
```

Rationale:

Synchronous design uses the principle that all registers in a same clock domain leave the reset state at the same time. Asynchronous assertion ensures that the design could be reset even if the input clock is not yet functional. Synchronous deassertion ensures that the component startup sequence is reproducible and that the clock is ready and stable before the deassertion of the reset inside the component. Doing so, if there is a glitch on the external reset, it will produce an internal reset that is active at least one clock period and guarantees a correct reset of the internal logic.

CNES Standard rules: Examples

STD_03900 : State machine type definition

STD_03900	State machine type definition	Major
Revision	5 / 2020-04-23	
Status / Engine	Implemented / Yosys-ghdl	
Classification	VLSI / Design / StateMachine	
Application Field	General	
Parent Rule	N/A	
Description	FSM states are encoded using enumerated type.	

- **Detailed Description:**

  Other type of state machine definition like vectors or integer are forbidden.

- **Rationale:**

  Enumerated type to encode FSM states allows readability and reuse.

- **Good Example:**

  Extracted from STD_03900_good.vhd

```
architecture Behavioral of STD_03900_good is
   constant c_Length : std_logic_vector(3 downto 0) := (others => '1'); -- How long we should count
    type t_state       is (init, loading, enabled, finished);                -- Enumerated type
 for state encoding
   signal sm_State   : t_state;                                        -- State signal
    signal Raz         : std_logic;                                                -- Load the
 length value and initialize the counter
   signal Enable     : std_logic;                                      -- Counter enable signal
   signal Length     : std_logic_vector(3 downto 0);                  -- Counter length for counting
   signal End_Count  : std_logic;                                      -- End signal of counter
begin
   -- A simple counter with loading length and enable signal
   Counter:Counter
   port map (
      i_Clock     => i_Clock,
      i_Reset_n   => i_Reset_n,
      i_Raz       => Raz,
      i_Enable    => Enable,
      i_Length    => Length,
      o_Done      => End_Count
   );

    -- FSM process controlling the counter. Start or stop it in function of the input (i_Start
 & i_Stop),
   -- load the length value, and wait for it to finish
   P_FSM:process(i_Reset_n, i_Clock)
   begin
      if (i_Reset_n='0') then
         sm_State <= init;
         Raz      <= '0';
         Enable   <= '0';
         Count_Length <= (others=>'0');
      elsif (rising_edge(i_Clock)) then
            case sm_State is
               when init =>
               -- Set the length value
                  Length <= c_Length;
                  sm_State <= loading;
               when loading =>
               -- Load the counter and initialize it
                  Raz <= '1';
                  sm_State <= enabled;
               when enabled =>
               -- Start or stop counting depending on inputs until it finishes
                  Raz <= '0';
                  if (End_Count='0') then
                  -- The counter has not finished, wait
                     Enable <= i_Start xor not i_Stop;
                     sm_State  <= Enabled;
                  else
                  -- The counter has finished, nothing else to do
                     Enable <= '0';
                     sm_State <= finished;
                  end if;
               when others =>
                  sm_State <= init;
            end case;
      end if;
   end process;
end Behavioral;
```

- **Bad Example:**

  Extracted from STD_03900_bad.vhd

```
architecture Behavioral of STD_03900_bad is
   constant c_Length : std_logic_vector(3 downto 0) := (others => '1');  -- How long we should count
   signal sm_State   : std_logic_vector(3 downto 0);  -- State signal
   signal Raz        : std_logic;  -- Load the length value and initialize the counter
   signal Enable     : std_logic;       -- Counter enable signal
   signal Length     : std_logic_vector(3 downto 0);  -- Counter length for counting
   signal End_Count  : std_logic;       -- End signal of counter
begin
   -- A simple counter with loading length and enable signal
   Counter : Counter
      port map (
         i_Clock   => i_Clock,
         i_Reset_n => i_Reset_n,
         i_Raz     => Raz,
         i_Enable  => Enable,
         i_Length  => Length,
         o_Done    => End_Count
         );

    -- FSM process controlling the counter. Start or stop it in function of the input (i_Start
 & i_Stop),
   -- load the length value, and wait for it to finish
   P_FSM : process(i_Reset_n, i_Clock)
   begin
      if (i_Reset_n = '0') then
         sm_State <= "0001";
         Raz      <= '0';
         Enable   <= '0';
         Count_Length <= (others=>'0');
      elsif (rising_edge(i_Clock)) then
            case sm_State is
               when "0001" =>
                  -- Set the length value
                  Length   <= c_Length;
                  sm_State <= "0010";
               when "0010" =>
```

CNES Standard rules: Examples

STD_04000 : State machine case enumeration completion

STD_04000	State machine case enumeration completion	Major
Revision	6 / 2021-10-14	
Status / Engine	Implemented / Yosys-ghdl	
Classification	VLSI / Design / StateMachine	
Application Field	General	
Parent Rule	N/A	
Description	VHDL code addresses all the defined states of the state machine.	

- **Detailed Description:**

  When all cases statement are not explicitly addressed in the VHDL code, an extra "when others" case will be added. "when others" instruction handles the default condition when none of the previous case statements are met.

- **Rationale:**

  State completion ensures deterministic behaviour between simulation and final design.

CNES Standard rules: Examples

STD 04000 : State machine case enumeration completion

Good Example:

Extracted from STD_04000_good.vhd

```
architecture Behavioral of STD_04000_good is
  constant c_Length   : std_logic_vector(3 downto 0) := (others => '1');  -- How long we should count
   type t_state is (init, loading, enabled, finished);  -- Enumerated type for state encoding
   signal sm_State     : t_state;       -- State signal
   signal Raz          : std_logic;  -- Load the length value and initialize the counter
   signal Enable       : std_logic;     -- Counter enable signal
   signal Count_Length : std_logic_vector(3 downto 0);  -- Counter length for counting
   signal End_Count    : std_logic;     -- End signal of counter
begin
   -- A simple counter with loading length and enable signal
   Counter : Counter
      port map (
         i_Clock   => i_Clock,
         i_Reset_n => i_Reset_n,
         i_Raz     => Raz,
         i_Enable  => Enable,
         i_Length  => Count_Length,
         o_Done    => End_Count
         );

    -- FSM process controlling the counter. Start or stop it in function of the input (i_Start
 & i_Stop),
   -- load the length value, and wait for it to finish
   P_FSM : process(i_Reset_n, i_Clock)
   begin
      if (i_Reset_n = '0') then
         sm_State <= init;
         Raz      <= '0';
         Enable   <= '0';
         Count_Length <= (others=>'0');
      elsif (rising_edge(i_Clock)) then
            case sm_State is
               when init =>
                  -- Set the length value
                  Count_Length <= c_Length;
                  sm_State     <= loading;
               when loading =>
                  -- Load the counter and initialize it
                  Raz      <= '1';
                  sm_State <= enabled;
               when enabled =>
                  -- Start or stop counting depending on inputs until it finishes
                  Raz <= '0';
                  if (End_Count = '0') then
                     -- The counter has not finished, wait
                     Enable   <= i_Start xor not i_Stop;
                     sm_State <= Enabled;
                  else
                     -- The counter has finished, nothing else to do
                     Enable   <= '0';
                     sm_State <= finished;
                  end if;
               when others =>
                  sm_State <= init;
            end case;
      end if;
   end process;
end Behavioral;
```

- **Bad Example:**

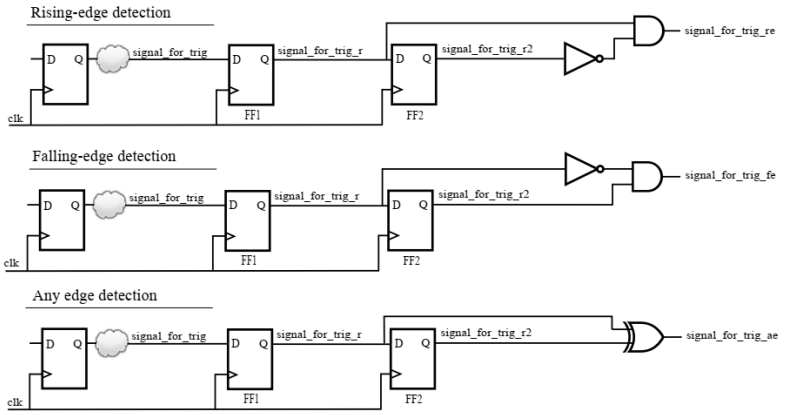
Extracted from STD_04000_bad.vhd

```
architecture Behavioral of STD_04000_bad is
  constant c_Length   : std_logic_vector(3 downto 0) := (others => '1');  -- How long we should count
   type t_state is (init, loading, enabled, finished);  -- Enumerated type for state encoding
   signal sm_State     : t_state;       -- State signal
   signal Raz          : std_logic;  -- Load the length value and initialize the counter
   signal Enable       : std_logic;     -- Counter enable signal
   signal Count_Length : std_logic_vector(3 downto 0);  -- Counter length for counting
   signal End_Count    : std_logic;     -- End signal of counter
begin
   -- A simple counter with loading length and enable signal
   Counter : Counter
      port map (
         i_Clock   => i_Clock,
         i_Reset_n => i_Reset_n,
         i_Raz     => Raz,
         i_Enable  => Enable,
         i_Length  => Count_Length,
         o_Done    => End_Count
         );

    -- FSM process controlling the counter. Start or stop it in function of the input (i_Start
 & i_Stop),
   -- load the length value, and wait for it to finish
   P_FSM : process(i_Reset_n, i_Clock)
   begin
      if (i_Reset_n = '0') then
         sm_State <= init;
         Raz      <= '0';
         Enable   <= '0';
         Count_Length <= (others=>'0');
      elsif (rising_edge(i_Clock)) then
            case sm_State is
               when init =>
                  -- Set the length value
                  Count_Length <= c_Length;
                  sm_State     <= loading;
               when loading =>
                  -- Load the counter and initialize it
                  Raz      <= '1';
                  sm_State <= enabled;
               when enabled =>
                  -- Start or stop counting depending on inputs until it finishes
                  Raz <= '0';
                  if (End_Count = '0') then
                     -- The counter has not finished, wait
                     Enable   <= i_Start xor not i_Stop;
                     sm_State <= Enabled;
                  else
                     -- The counter has finished, nothing else to do
                     Enable   <= '0';
                     sm_State <= finished;
                  end if;

                  --*** MISSING finished state of the FSM ***--

            end case;
      end if;
   end process;
end Behavioral;
```

CNES Standard rules: Examples

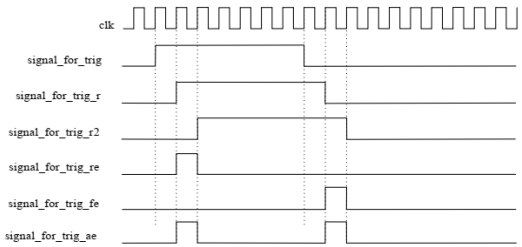
STD_04900 : Edge detection best practice

STD_04900	Edge detection best practice	Major
Revision	6 / 2020-04-23	
Status / Engine	Validated / None	
Classification	VLSI / Design / Synchronous	
Application Field	General	
Parent Rule	N/A	
Description	Synchronous mechanisms are used for signal edge detection.	



- **Detailed Description:**

A specific mechanism is used in order to detect rising or falling edge input signal.
This mechanism involves a real design clock, at least one D Flip-Flop to delay the signal, and combinational gate(s) to select the edge.



- **Rationale:**

Flip-Flops clock input is dedicated to a clock signal. Thus, using it as a way to detect a signal edge (by using rising_edge(...) or 'event attribute) may lead to the creation of a new clock domain for each signal edge detection implemented in the design: this is not the purpose.

Good Example:

Extracted from STD_04900_good.vhd

```
   -- Registration process to be able to detect edges of signal A
   P_Registration : process(i_Reset_n, i_Clock)
   begin
      if (i_Reset_n = '0') then
         A_r1 <= '0';
         A_r2 <= '0';
      elsif (rising_edge(i_Clock)) then
            A_r1 <= i_A;
            A_r2 <= A_r1;
      end if;
   end process;

-- Assign the outputs of the module:
   o_A_re <= A_r1 and not A_r2;
   o_A_fe <= not A_r1 and A_r2;
   o_A_ae <= A_r1 xor A_r2;
end Behavioral;
```

CNES Standard rules: Examples

STD_05200 : Output signal registration

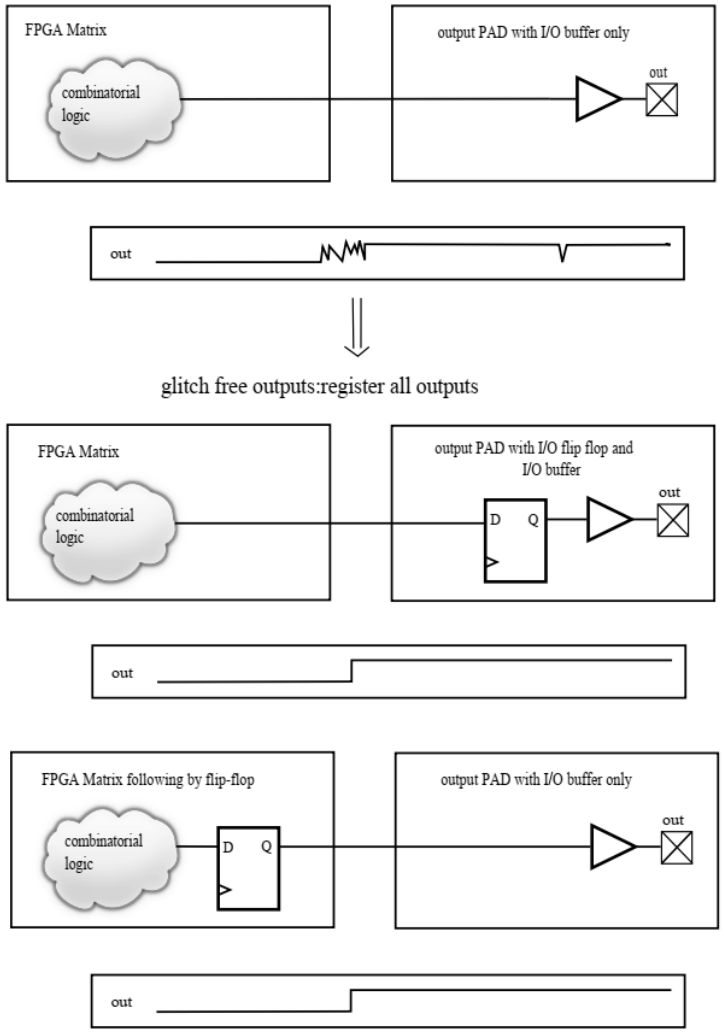
STD_05200	Output signal registration	Minor
Revision	6 / 2020-04-23	
Status / Engine	Implemented / Yosys-ghdl	
Classification	VLSI / Design / Synchronous	
Application Field	General	
Parent Rule	N/A	
Description	All outputs signal from a top level entity are registered.	

- **Detailed Description:**

Combinational outputs at top level are forbidden. Those outputs belong to the timing domain in which they are generated at top level. Whenever it is possible, use I/O blocks register instead of internal register for top level outputs. Unless specified and approved, a combinational signal is never used directly as a top level output.

- **Rationale:**

All outputs of an integrated circuit are coming from output registers whenever possible and from regular registers when not possible. The clock used is the same as the one used in the signal source clock domain. Doing so suppresses all glitches on the outputs whenever a signal level change occurs and enables control of the clocks to outputs delays of the circuit so that the time borrowed by any signal to propagate from its respective launching clock edge to its assigned device output is controlled and less than a given maximum allowed time. With controlled clocks to outputs delays, enough PCB propagation time and inputs to clocks delays is left for those outputs to be captured using the same clock in a remote device.



CNES Custom rules: Examples

CNE_00500 : Convention for signal naming

CNE_00500	Convention for signal naming	Minor
Revision	5 / 2020-04-23	
Status / Engine	Validated / None	
Classification	VLSI / Formatting / Naming	
Application Field	General	
Parent Rule	STD_00100	
Description	Each word that composes a signal name are clearly identified with an underscore.	

- **Detailed Description:**

  In order to separate words in signal name the following convention is applied: Name_Of_The_Signal.
  The separation by uppercase (NameOfTheSignal) is not used.

CNE_00600 : Convention for constant naming

CNE_00600	Convention for constant naming	Minor
Revision	5 / 2020-04-23	
Status / Engine	Validated / None	
Classification	VLSI / Formatting / Naming	
Application Field	General	
Parent Rule	STD_00100	
Description	Each word that composes a constant name are clearly identified with an underscore.	

- **Detailed Description:**

  In order to separate words in a constant name the following convention is applied: Name_Of_The_Constant.
  The separation by uppercase (NameOfTheConstant) is not used.

CNES Custom rules: Examples

CNE_00700 : Convention for process naming

CNE_00700	Convention for process naming	Minor
Revision	5 / 2020-04-23	
Status / Engine	Validated / None	
Classification	VLSI / Formatting / Naming	
Application Field	General	
Parent Rule	STD_00100	
Description	Each word that composes a process name are clearly identified with an underscore.	

- **Detailed Description:**

  In order to separate words in a process name the following convention is applied: Name_Of_The_Process.
  The separation by uppercase (NameOfTheProcess) is not used.

CNE_00800 : Convention for generic ports

CNE_00800	Convention for generic ports	Minor
Revision	5 / 2020-04-23	
Status / Engine	Validated / None	
Classification	VLSI / Formatting / Naming	
Application Field	General	
Parent Rule	STD_00100	
Description	Each words that composes a generic port name are clearly identified with an underscore.	

- **Detailed Description:**

  In order to separate words in a generic port name the following convention is applied: Name_Of_The_Generic.
  The separation by uppercase (NameOfTheGeneric) is not used.

CNES Custom rules: Examples

CNE_01000 : Identification of variable name

CNE_01000	Identification of variable name	Minor
Revision	5 / 2020-04-23	
Status / Engine	Implemented / ZamiaCad	
Classification	VLSI / Formatting / Naming	
Application Field	General	
Parent Rule	STD_00100	
Description	The name of a variable use "v_" prefix.	

- **Detailed Description:**

  No additional information.

- **Rationale:**

  When an unique naming convention is applied to the whole source files from the design, then the resulting code is homogenized which increases readability. With this convention, designer will be able to track synthesis of variable and especially identify if a variable created some unwanted flip-flops.

CNES Custom rules: Examples

CNE_01900 : Identification of registered signals

CNE_01900	Identification of registered signals	Minor
Status / Engine	Validated / None	
Classification	VLSI / Formatting / Naming	
Application Field	General	
Parent Rule	STD_00100	
Description	The suffix of a signal that is a registration of another one is: "_r".	

- **Detailed Description:**

The signal source is also included inside the signal name.
Thus, a signal that is clock delayed of a signal named My_Signal is My_Signal_r.
If a small number of registration of a same signal is used (less or equal to 3), SIGNAL_r can become SIGNAL_rx where x is the number of registration stage.
If a significant number of registration of a same signal is used, use an array for registration level instead of different signals.

Good Example:

Extracted from CNE_01900_good.vhd

```
entity CNE_01900_good is
   port  (
      i_Reset_n   : in std_logic;   -- Reset signal
      i_Clock     : in std_logic;   -- Clock signal
      i_D    : in std_logic;   -- Signal on which detect edges
      o_D_re : out std_logic   -- Rising edge of My_Sig
   );
end CNE_01900_good;

architecture Behavioral of CNE_01900_good is
   signal D_r1  : std_logic; -- i_D registered 1 time
   signal D_r2  : std_logic; -- i_D registered 2 times
begin
   -- Rising edge detection process
   P_detection: process(i_Reset_n, i_Clock)
   begin
      if (i_Reset_n='0') then
         D_r1 <= '0';
         D_r2 <= '0';
      elsif (rising_edge(i_Clock)) then
            D_r1 <= i_D;
            D_r2 <= D_r1;
      end if;
   end process;

   o_D_re <= D_r1 and not D_r2;
end Behavioral;
```

CNES Custom rules: Examples

CNE_04800 : Finite State Machine two processes based

CNE_04800	Finite State Machine two processes based	Note
Revision	6 / 2020-04-23	
Status / Engine	Validated / None	
Classification	VLSI / Design / StateMachine	
Application Field	General	
Parent Rule	CNE_04600	
Description	FSM coding style use the two processes method.	

- **Detailed Description:**

FSM coding style use one synchronous process for state registration and one asynchronous process for states and outputs assertion.

Good Example:

Extracted from CNE_04800_good.vhd

```
architecture Behavioral of CNE_04800_good is
  constant c_Length    : std_logic_vector(3 downto 0) := (others => '1'); -- How long we should count
    type t_state            is (init, loading, enabled, finished);                -- Enumerated
 type for state encoding
   signal sm_State      : t_state;                                          -- State signal
   signal sm_Next_State : t_state;                                          -- Next state
    signal Raz          : std_logic;                                                -- Load the
 length value and initialize the counter
   signal Enable     : std_logic;                                       -- Counter enable signal
  signal Length     : std_logic_vector(3 downto 0);                  -- Counter length for counting
   signal End_Count  : std_logic;                                       -- End signal of counter
begin
   Counter:pkg_Counter
   port map (
      i_Clock     => i_Clock,
      i_Reset_n   => i_Reset_n,
      i_Raz       => Raz,
      i_Enable    => Enable,
      i_Length    => Length,
      o_Done      => End_Count
   );

    -- FSM process controlling the counter. Start or stop it in function of the input (i_Start
 & i_Stop),
   -- load the length value, and wait for it to finish

   -- Process registration
   P_FSM_State_Reg:process(i_Reset_n, i_Clock)
   begin
      if (i_Reset_n='0') then
         sm_State <= init;
      elsif (rising_edge(i_Clock)) then
            sm_State <= sm_Next_State;
      end if;
   end process;

   -- Outputs assertion
   P_FSM_Output:process(sm_State, i_Start, i_Stop, End_Count)
   begin
      Raz <= '0';
      Enable <= '0';
      Length <= c_Length;  -- Set the length value

      case sm_State is
         when init =>
            sm_Next_State <= loading;
         when loading =>
            -- Load the counter and initialize it
            Raz <= '1';
            sm_Next_State <= enabled;
         when enabled =>
            -- Start or stop counting depending on inputs until it finishes
            if (End_Count='0') then
               Enable <= i_Start xor not i_Stop;
               sm_Next_State <= enabled;
            else
               sm_Next_State <= finished;
            end if;
         when others =>
            sm_Next_State <= init;
      end case;
   end process;
end Behavioral;
```

CNES Custom rules: Examples

CNE_01100 : Identification of ports direction inside entity port name

CNE_01100	Identification of ports direction inside entity port name	Minor
Revision	6 / 2020-04-23	
Status / Engine	Validated / None	
Classification	VLSI / Formatting / Naming	
Application Field	General	
Parent Rule	STD_00100	
Description	Entity port name uses prefix to determine the port direction.	

- **Detailed Description:**

  Prefixes are:
  "i_" for input port,
  "o_" for output port,
  "b_" for bidirectional port.

- **Rationale:**

  Indicating the port direction inside the port name improves readability.

- **Good Example:**

  Extracted from CNE_01100_good.vhd

```
entity CNE_01100_good is
   port  (
      i_Clock     : in std_logic;    -- Clock signal
      i_Reset_n   : in std_logic;    -- Reset signal
      i_D         : in std_logic;    -- D Flip-Flop input signal
      o_Q         : out std_logic    -- D Flip-Flop output signal
   );
end CNE_01100_good;
```

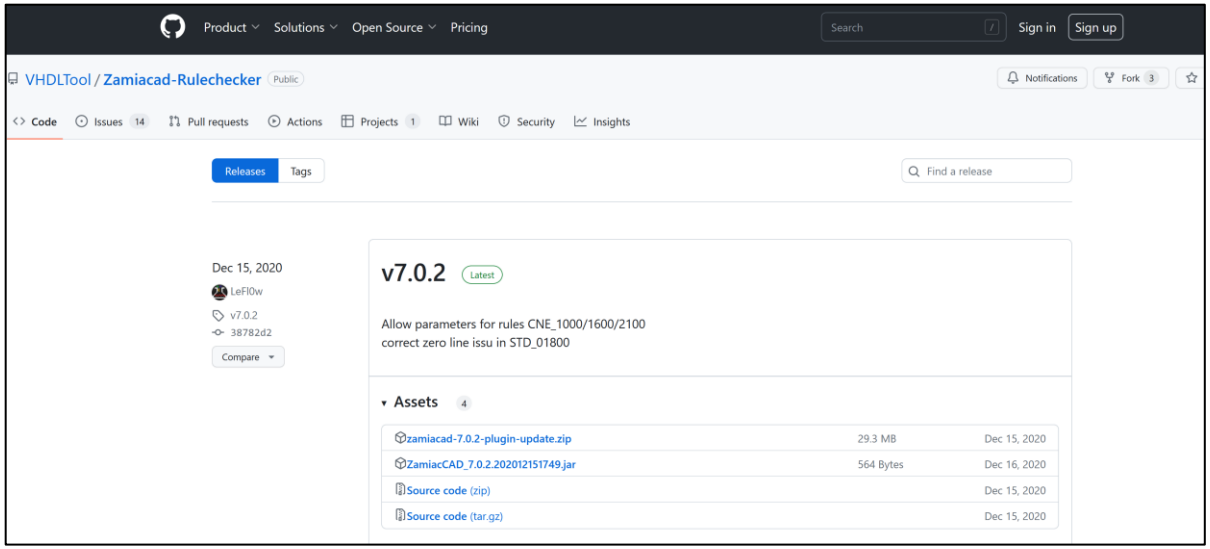
Automated Rule Checking

VHDLTool/**Zamiacad-Rulechecker**

Zamiacad plugin rulechecker (https://sourceforge.net/p/zamiacad/zamia-eclipse-plugin/ci/master/tree/) eclipse plugin clone including the rulechecker addon (need zamiacad code to work)

11 Contributors | 14 Issues | 3 Stars | 3 Forks

https://github.com/VHDLTool/Zamiacad-Rulechecker

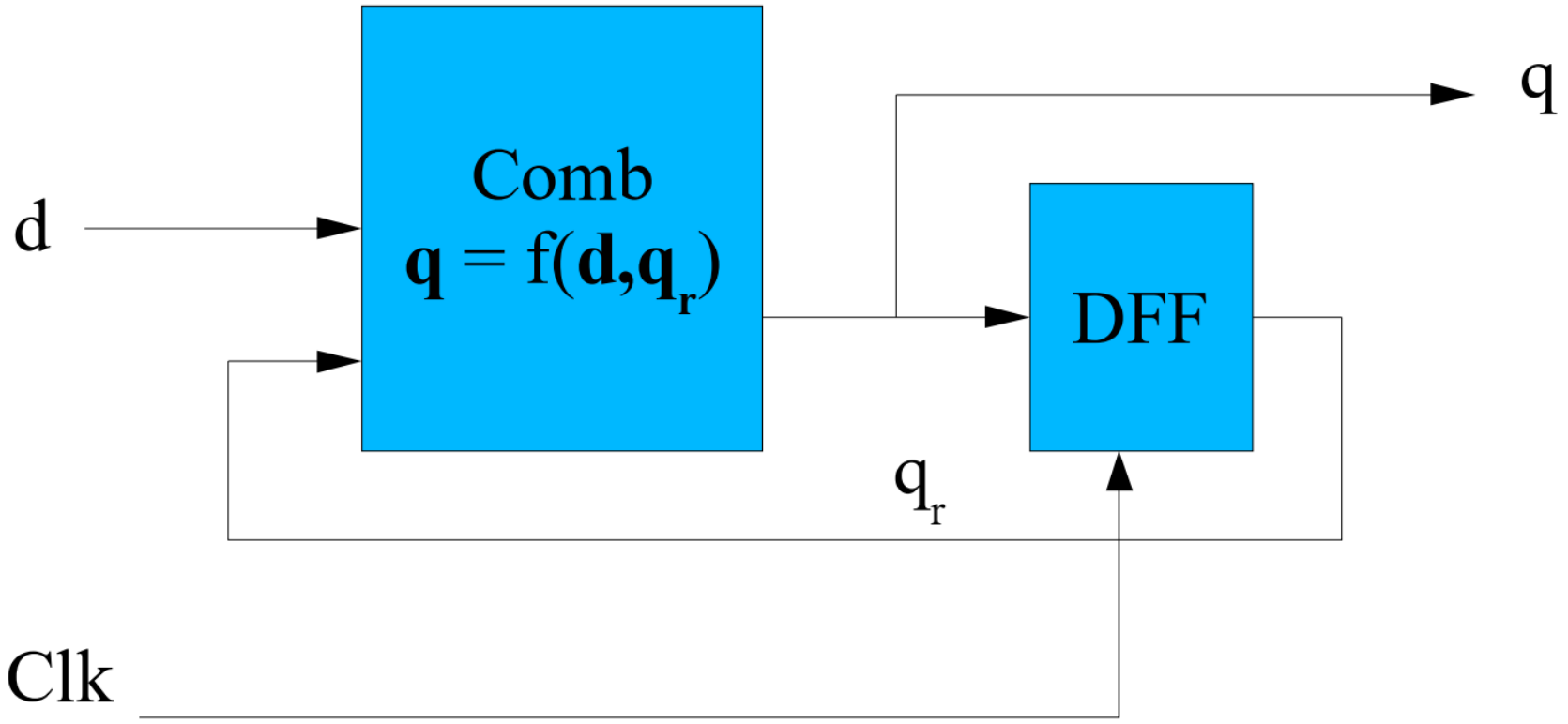


Traditional VHDL design

- Features
  - Many concurrent statements
  - Many signals
  - Few and small process statements
  - No unified signal naming convention
  - Coding is done at low RTL level:
    - Assignments with logical expressions
    - Only simple array data structures are used
- Problems
  - Slow execution due to many signals and processes
  - Dataflow coding difficult to understand
  - Algorithm difficult to understand
  - No distinction between sequential and combinational signals
  - Difficult to identify related signals
  - Large port declarations in entity headers

Abstract of synchronous digital system

- Two separate parts
  - A combinational part
  - A sequential part



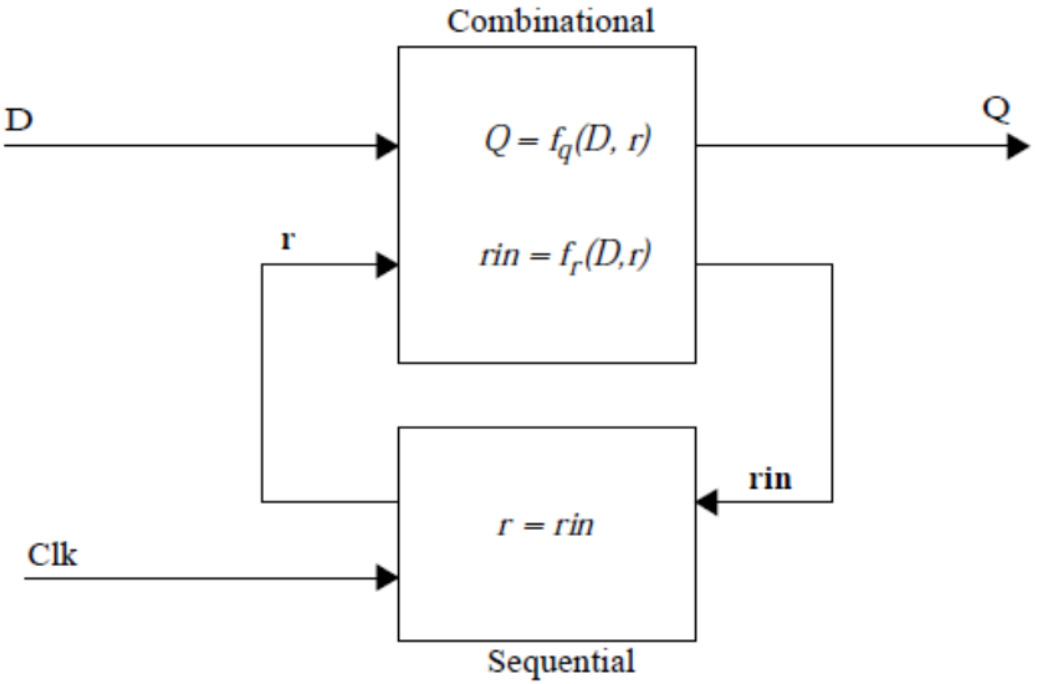
Two-process method (J. Gaisler)

- Two local signals are declared:
  - register-in (*rin*)
  - register-out (*rout*)
- The full algorithm (*q = f(d,r)*) is performed in combinational process
- Comb. process is sensitive to all input ports and register outputs *r*
- The sequential process is only sensitive to the clock
- Record types are used extensively
  - Record in each module defines all the registers
  - Records in Packages define port I/O
  - **Don't have to fix every instantiation when a port map changes, just update the record and everything still works!**
- Few signals, mostly in/out/state records
- Variables
- Functions / procedures are used extensively

Two-process VHDL entity

- From “A Structured VHDL Design Method” by Jiri Gaisler

- Only two processes combinational (asynchronous) and sequential (registers) are used
- Complete algorithm can be coded sequential in the combinational process that outputs the internal state and sequential process registers the internal state and handles reset



* http://www.gaisler.com/doc/vhdl2proc.pdf for a full write-up of the benefits of this style.

Two-process method: data types

- The local signals *r* and *rin* are of composite type (record) and include all registered values
- All outputs are grouped into one entity-specific record type, declared in a global interface package
- Input ports are of output record types from other entities
- A local variable of the registered type is declared in the combinational processes to hold newly calculated values
- Additional variables of any type can be declared in the combinational process to hold temporary values

Example

```
use work.interface.all;

entity irqctrl is port (
  clk   : in std_logic;
  rst   : in std_logic;
  sysif : in sysif_type;
  irqo  : out irqctrl_type);
end;

architecture rtl of irqctrl is

  type reg_type is record
    irq    : std_logic;
    pend   : std_logic_vector(0 to 7);
    mask   : std_logic_vector(0 to 7);
  end record;

  signal r, rin : reg_type;

begin

  comb : process (sysif, r)
    variable v : reg_type;
  begin
    v := r; v.irq := '0';
    for i in r.pend'range loop
      v.pend := r.pend(i) or
         (sysif.irq(i) and r.mask(i));
      v.irq := v.irq or r.pend(i);
    end loop;
    rin <= v;
    irqo.irq <= r.irq;
  end process;

  reg : process (clk)
  begin
    if rising_edge(clk) then
      r <= rin;
    end if;
  end process;

end architecture;
```

Two-process method: using records

- Useful to group related signals
- Nested records further improves readability
- Directly synthesizable
- Element name might be difficult to find in synthesized netlist

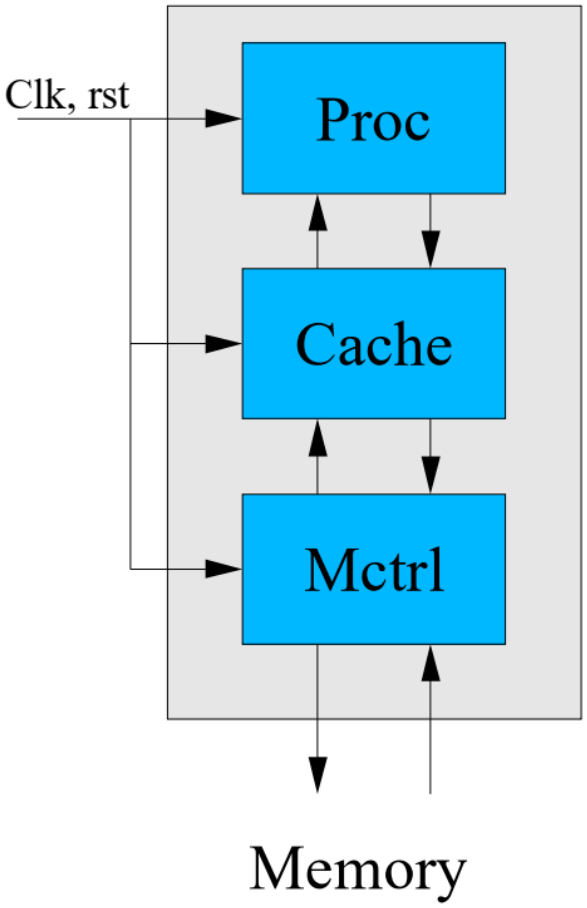
```
type reg1_type is record
  f1 : std_logic_vector(0 to 7);
  f2 : std_logic_vector(0 to 7);
  f3 : std_logic_vector(0 to 7);
end record;

type reg2_type is record
  x1 : std_logic_vector(0 to 3);
  x2 : std_logic_vector(0 to 3);
  x3 : std_logic_vector(0 to 3);
end record;

type reg_type is record
  reg1 : reg1_type;
  reg2 : reg2_type;
end record;

variable v : regtype;

v.reg1.f3 := "0011001100";
```

```
type mul32_in_type is record
  op1             : std_logic_vector(32 downto 0); -- operand 1
  op2             : std_logic_vector(32 downto 0); -- operand 2
  flush           : std_logic;
  signed          : std_logic;
  start           : std_logic;
  mac             : std_logic;
  acc             : std_logic_vector(39 downto 0);
  --y               : std_logic_vector(7 downto 0); -- Y (MSB MAC register)
  --asr18           : std_logic_vector(31 downto 0); -- LSB MAC register
end record;

type mul32_out_type is record
  ready           : std_logic;
  nready          : std_logic;
  icc             : std_logic_vector(3 downto 0); -- ICC
  result          : std_logic_vector(63 downto 0); -- mul result
end record;
```

```
entity mul32 is
generic (
    tech    : integer := 0;
    multype : integer range 0 to 3 := 0;
    pipe    : integer range 0 to 1 := 0;
    mac     : integer range 0 to 1 := 0;
    arch    : integer range 0 to 3 := 0;
    scantest: integer := 0
);
port (
    rst     : in  std_ulogic;
    clk     : in  std_ulogic;
    holdn   : in  std_ulogic;
    muli    : in  mul32_in_type;
    mulo    : out mul32_out_type;
    testen  : in  std_ulogic := '0';
    testrst : in  std_ulogic := '1'
);
end;
```

Hierarchical design

- Grouping of signals makes code readable and shows the direction of the dataflow




```
use work.interface.all;

entity cpu is port (
  clk     : in std_logic;
  rst     : in std_logic;
  mem_in  : in mem_in_type;
  mem_out : out mem_out_type);
end;

architecture rtl of cpu is
  signal cache_out  : cache_type;
  signal proc_out   : proc_type;
  signal mctrl_out  : mctrl_type;
begin

  u0 : proc port map
    (clk, rst, cache_out, proc_out);

  u1 : cache port map
    (clk, rst, proc_out, mem_out cache_out);

  u2 : mctrl port map
    (clk, rst, cache_out, mem_in, mctrl_out,
     mem_out);

end architecture;
```

Dataflow vs. two-process comparison

	Two-process method	Dataflow coding
Adding ports	• Add field in interface record type	• Add port in entity declaration • Add port to sensitivity list (input) • Add port in component declaration • Add signal to port map of component • Add definition of signal in parent
Adding registers	• Add field in register record type	• Add two signal declaration (d & q) • Add q-signal in sensitivity list • Add driving signal in comb. process • Add driving statement in seq. process
Debugging	• Put a breakpoint on first line of combination process and step forward • New signal values visible in local variable v	• Analyze how the signal(s) of interest are generated • Put a breakpoint on each process or concurrent statment in the path • New signal value not immediately visible
Tracing	• Trace the r-signal (state) • Automatic propagation of added or deleted record elemenets	• Find all signals that are used to implement registers • Trace all found signals • Re-iterate after each added or deleted signal
	•	•

Coding Style Example – Simple Counter

```
entity Counter is
  generic (
    TPD_G : time := 1 ns); -- Simulated propagation delay
  port (
    clk      : in  sl;
    rst      : in  sl;
    max      : in  slv(7 downto 0);
    count    : out slv(7 downto 0);
    rollover : out sl);
end entity Counter;

architecture rtl of Counter is

  -- Record containing all register elements
  type RegType is record
    count    : slv(7 downto 0);
    rollover : sl;
  end record RegType;

  -- Initial and reset values for all register elements
  constant REG_INIT_C : RegType := (
    count    => (others => '0'),
    rollover => '0');

  -- Output of registers. (The Q output)
  signal r : RegType := REG_INIT_C;

  -- Combinatorial input to registers (The D input)
  signal rin : RegType;

begin

  -- Boilerplate sequential process
  -- Assign rin to r on rising edge of clk to create registers
  seq : process (clk) is begin
    if (rising_edge(clk)) then
      r <= rin after TPD_G;
    end if;
  end process seq;


  -- Main module logic
  -- Generates rin based on r and any module inputs
  comb : process (max, r, rst) is
    variable v : RegType;
  begin
    -- Initialize v with current value of all registers
    v := r;

    -- Set register values for next rising edge
    v.count    := r.count + 1;
    v.rollover := '0';

    -- Override above assignments when max reached
    if (r.count = max) then
      v.count := (others => '0');
      v.rollover := '1';
    end if;

    -- Synchronous reset
    -- Override all above assignments and apply init values
    if (rst = '1') then
      v := REG_INIT_C;
    end if;

    -- Assign final state of local variable to rin signal
    -- Registers will assume these values on next rising edge
    rin <= v;

    -- Assign registered signals to outputs
    count <= r.count;
    rollover <= r.rollover;
  end process comb;

end architecture rtl;
```

*Source: 'SLAC Firmware Standard Library'

Coding Style Example – UART BRG*

```
entity UartBrg is
   generic (
      CLK_FREQ_G   : real    := 125.0E6;  -- Default 125 MHz
      BAUD_RATE_G  : integer := 115200;   -- Default 115.2 kbps
      MULTIPLIER_G : integer := 16);
   port (
      clk       : in  sl;
      rst       : in  sl;
      baudClkEn : out sl);
end entity UartBrg;

architecture rtl of UartBrg is
   constant CLK_DIV_C : integer := integer(CLK_FREQ_G /
real(BAUD_RATE_G * MULTIPLIER_G)) - 1;
   type RegType is record
      count     : integer;
      baudClkEn : sl;
   end record RegType;
   constant REG_INIT_C : RegType := (
      count     => 0,
      baudClkEn => '0');
   signal r   : RegType := REG_INIT_C;
   signal rin : Regtype;
```

```
begin
comb : process (r, rst) is
      variable v : RegType;
   begin
      v := r;
      v.count     := r.count + 1;
      v.baudClkEn := '0';
      if (r.count = CLK_DIV_C) then
         v.count     := 0;
         v.baudClkEn := '1';
      end if;
      if (rst = '1') then
         v := REG_INIT_C;
      end if;
      rin       <= v;
      baudClkEn <= r.baudClkEn;
   end process;
   seq : process (clk) is
   begin
      if (rising_edge(clk)) then
         r <= rin;
      end if;
   end process;
end architecture rtl;
```

*Source: 'SLAC Firmware Standard Library'

Coding Style Example – FSMs

- Simple case-statement implementation
- Maintains current state
- Both combinational and registered output possible

```
architecture rtl of mymodule is

  type state_type is (first, second, last);

  type reg_type is record
    state : state_type;
    drive : std_logic;
  end record;

  signal r, rin : reg_type;

begin
  comb : process(...., r)
  variable v : reg_type;
  begin

    v := r;

    case r.state is
    when first =>
      if cond0 then v.state := second; end if;
    when second =>
      if cond1 then v.state := first;
      elsif cond2 then v.state := last; end if;
    when others =>
      v.drive := '1'; v.state := first;
  end case;

  if reset = '1' then v.state := first; end if;

  modout.cdrive <= v.drive; -- combinational output
  modout.rdrive <= r.drive; -- registered output

  end process;
```

Procedure Example

```
procedure updateCount(
  v   : inout RegType;
  max : in    slv(7 downto 0)) is
begin
  if (v.count = max) then
    v.count    := (others => '0');
    v.rollover := '1';
  else
    v.count    := v.count + 1;
    v.rollover := '0';
  end if;
end procedure;

Comb : process (r, max) is
  variable v := RegType;
begin
  v := r;
  updateCount(v);
  rin <= v;
end process;
```

*Source: 'SLAC Firmware Standard Library'

Packages

- Packages are very useful for sharing constants, typedefs, functions and procedures
- With the “two-process” coding style, functions and procedures become very easy to use
  - Can abstract complex logic to avoid code repetition
  - Procedures used when a logic block has more than one output, or needs to read and update a variable

Two-process method: Benefits

- Sequential coding is well known and understood
- Algorithms easily extracted
- Easy to extend
- Readability = Maintainability
- Fast simulation
- Easier debugging and verification
- No simulation/synthesis discrepancies

Two-process method : Examples

- **Example**: Current NOEL-V integer pipeline
  - 2 processes
    - Combinational, 2200 lines
    - Clocked, 60 lines
    - 53/22 procedures/functions, ~5000 lines
      (not counting generic ones from other files)
  - 17 in port signals
  - 13 out port signals
  - 4 local signals (+12 for disassembler)
- The in/out ports connect to separate modules for: caches, register file, branch prediction, IRQ, debug, mul/div.

*Source: Gaisler

Two-process method : Examples

- **Example**: Current NOEL-V cache controller and MMU
  - 3 processes
    - Combinational, 3500 lines
    - Two clocked, one assignment each (+debug)
    - 10/45 procedures/functions, ~1500 lines (not counting generic ones from other files)
  - 12 in port signals
  - 4 out port signals
  - 4 local signals (+2 for debug)
- The in/out ports connect to: AHB bus, caches, integer pipeline.
- Both LEON5 (Sparc) and NOEL-V (RISC-V)!

*Source: Gaisler

Two-process method : Examples

- **Example**: First half of the execute stage

```
ex_flush    := '0';
if wb_fence_i = '1' or v.wb.flushall = '1' or x_branch = '1' then
  ex_flush  := '1';
end if;
ex_branch_flush     := '0';
if wb_fence_i = '1' or v.wb.flushall = '1' then
  ex_branch_flush   := '1';
end if;
ex_forwarding(...);       -- Lane 0
ex_forwarding(...);       -- Lane 1
branch_unit(...);
jump_ex_forwarding(...);
jump_unit(...);
alu_execute(...);         -- ALU0
alu_execute(...);         -- ALU1
ex_stdata_forwarding(...);
mul_gen(...);
for i in 0 to ISSUEWAYS-1 loop
  ex_xc(i)                := r.e.ctrl(i).xc;
  ex_xc_cause(i)          := r.e.ctrl(i).cause;
  ex_xc_tval(i)           := r.e.ctrl(i).tval;
end loop;
...
```

*Source: Gaisler

Two-process method : Examples

- **Example**: Detail of the execute stage

```
-- Forwarding Lane 1 ------------------------------------------------
ex_forwarding(r,                                    -- in  : Registers
              1,                                    -- in  : Lane 1
              r.e.forw(1),                          -- in  : Forwarded from Memory
              ex_alu_op1(1),                        -- out : Output op1 from Mux
              ex_alu_op2(1)                         -- out : Output op2 from Mux
              );


-- Branch Unit -----------------------------------------------------
branch_unit(ex_alu_op1(1),                          -- in  : Forwarded Op1
            ex_alu_op2(1),                          -- in  : Forwarded Op2
            r.e.ctrl(1).valid,                      -- in  : Enable/Valid Signal
            r.e.ctrl(1).branch.valid,               -- in  : Branch Valid Signal
            r.e.ctrl(1).inst(14 downto 12),         -- in  : Inst funct3
            r.e.ctrl(1).branch.addr,                -- in  : Branch Target Address
            r.e.ctrl(1).branch.naddr,               -- in  : Branch Next Address
            r.e.ctrl(1).branch.taken,               -- in  : Prediction
            r.e.ctrl(1).pc,                         -- in  : PC In
            ex_branch_valid,                        -- out : Branch Valid
            ex_branch_mis,                          -- out : Branch Outcome
            ex_branch_addr,                         -- out : Branch Address
            ex_branch_xc,                           -- out : Branch Exception
            ex_branch_cause,                        -- out : Exception Cause
            ex_branch_tval                          -- out : Exception Value
            );
```

*Source: Gaisler