NOT MEASUREMENT SENSITIVE

NASA TECHNICAL HANDBOOK

NASA-HDBK-4011

Office of the NASA Chief Engineer

Approved: 2022-06-06

VERY HIGH-SPEED INTEGRATED CIRCUIT (VHSIC) HARDWARE DESCRIPTION LANGUAGE (VHDL) STYLE HANDBOOK

Trade names and trademarks are used in this NASA Technical Handbook for identification only. Their usage does not constitute an official endorsement, either expressed or implied, by NASA.

APPROVED FOR PUBLIC RELEASE – DISTRIBUTION IS UNLIMITED

DOCUMENT HISTORY LOG

Status	Document Revision	Change Number	Approval Date	Description
Baseline			2022-06-06	Initial Release

FOREWORD

This NASA Technical Handbook is published by the National Aeronautics and Space Administration (NASA) as a guidance document to provide engineering information; lessons learned; possible options to address technical issues; classification of similar items, materials, or processes; interpretative direction and techniques; and any other type of guidance information that may help the Government or its contractors in the design, construction, selection, management, support, or operation of systems, products, processes, or services.

This NASA Technical Handbook is approved for use by NASA Headquarters and NASA Centers, including Component Facilities and Technical and Service Support Centers. It may also apply to the Jet Propulsion Laboratory (a Federally Funded Research and Development Center), other contractors, recipients of grants, cooperative agreements, or other agreements only to the extent specified or referenced in the applicable contracts, grants, or agreements.

This NASA Technical Handbook establishes a style guide when writing VHDL text.

Requests for information should be submitted via “Feedback” at https://standards.nasa.gov. Requests for changes to this NASA Technical Handbook should be submitted via MSFC Form 4657, Change Request for a NASA Engineering Standard.

Original Signed by Adam West for

Ralph R. Roe, Jr.
NASA Chief Engineer

June 6, 2022

Approval Date

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF APPENDICES

VERY HIGH-SPEED INTEGRATED CIRCUIT (VHSIC) HARDWARE DESCRIPTION LANGUAGE (VHDL) STYLE HANDBOOK

1. SCOPE

1.1 Purpose

Design of modern application-specific integrated circuits, such as gate arrays or field programmable gate arrays (FPGAs), currently is achieved through a text-based approach—not via schematics. The text description is similar in appearance to a programming language such as C or Pascal. The purpose of this NASA Technical Handbook is to aid the design engineer in using good practices to produce quality designs, minimize errors, and produce a description that is maintainable and reviewable. This NASA Technical Handbook covers a hardware description language called very high-speed integrated circuit (VHSIC) hardware description language (VHDL).

VHDL is a large and powerful language. It is also extremely flexible, as there are many ways to describe a design or model. The variety of circuits, situations, and technologies the logic designer encounters are so varied that a one-size-fits-all solution is not practical and will needlessly constrain the design engineer or result in poor guidance. The intent of this NASA Technical Handbook is to cover most situations while giving the design engineer flexibility in the writing of the description. With the powerful VHDL language and very complex application-specific integrated circuits (ASICs) and FPGAs currently available, flexibility is needed in this design domain.

Because having no guidelines or rigidly enforcing rules in various domains hinders the goal of correct, maintainable, and reviewable descriptions, this NASA Technical Handbook takes a middle-ground approach, summarized by Brooks (1975) as follows:

> At first sight, the idea of any rules or principles being superimposed on the creative mind seems more likely to hinder than to help, but this is quite untrue in practice. Disciplined thinking focuses inspiration rather than blinkers it.

1.2 Applicability

This NASA Technical Handbook is applicable to digital designs that are described by VHDL text. The principles that form the basis of this NASA Technical Handbook are applicable to other hardware description languages such as Verilog™.

This NASA Technical Handbook is approved for use by NASA Headquarters and NASA Centers, including Component Facilities and Technical and Service Support Centers. It may also apply to the Jet Propulsion Laboratory (a Federally Funded Research and Development Center), other contractors, recipients of grants, cooperative agreements, or other agreements only to the extent specified or referenced in the applicable contracts, grants, or agreements.

This NASA Technical Handbook, or portions thereof, may be referenced in contract, program, and other Agency documents for guidance.

2. APPLICABLE DOCUMENTS

2.1 General

None.

References are provided in Appendix A.

2.2 Order of Precedence

2.2.1 The guidance established in this NASA Technical Handbook does not supersede or waive existing guidance found in other Agency documentation.

2.2.2 Conflicts between this NASA Technical Handbook and other documents are to be resolved by the delegated Technical Authority.

3. ACRONYMS, ABBREVIATIONS AND DEFINITIONS

3.1 Acronyms and Abbreviations

ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
CDD	Coding and Design Document
CPU	Central Processing Unit
DSP	Digital Signal Processing
ESA	European Space Agency
FFRDC	Federally Funded Research and Development Center
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine

GRC	Glenn Research Center
HDBK	Handbook
HDL	Hardware Description Language
I/O	Input/Output
ID	Identification
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
ISE®	Integrated Software Environment
MIL	Military
MSFC	Marshall Space Flight Center
mux	Multiplexer
NASA	National Aeronautics and Space Administration
RAM	Random Access Memory
RTL	Register Transfer Level
SET	Single-Event Transient
SEU	Single-Event Upset
STD	Standard
TMR	Triple Modular Redundancy
V	Volt
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit

3.2 Definitions

None.

4. TECHNICAL GUIDANCE

4.1 General Style Guidelines

4.1.1 Design and write the description properly, describing the hardware that is to be implemented. Hardware description languages such as VHDL and Verilog™, while superficially appear similar to software languages, are definitely not software languages. The statements are not translated into code that runs on a central processing unit (CPU). Do not treat the design as "software," leaving it up to the logic synthesizer to do your thinking and design work for you. Write your description in a style that matches the target hardware architecture which will lead to predictable results and minimize the chance of a nasty surprise.

4.1.2 Block comments are useful when describing complex behavior, explaining an algorithm, or providing system information to put the VHDL text in context. In-line or "single line comments" (using the "--" notation) reduce line-count for simple annotations. Some hardware description language (HDL) editors (e.g., those in Libero™, Integrated Software Environment (ISE®), Modelsim®, and Synplicity®) can add the "--" comment designator automatically to a range of text.

VHDL-2008, VHDL Language Reference Manual, has “delimited comments” as shown by the following examples:

```
/* A long comment may be written
   on several consecutive lines   */

x := 1; /* Comments /* do not nest */
```

4.1.3 Use labels for all processes, loops, and component instantiations. Wherever possible, qualify all language constructs such as subprograms, package declarations and bodies, entities, architectures, and loop statements (i.e., the identifier associated with the construct should also appear at its end). A simple example (most useful for long blocks of text) follows:

```
component por_sync is port
  ( clk         : in  std_logic;
    por_n       : in  std_logic;
    por_n_sync  : out std_logic   );
  end component por_sync;
```

4.1.4 Explicitly declare components such as input/output (I/O) cells and global buffers. While the logic synthesizer can infer these components, that is not desirable since the design engineer will, after each synthesis run, have to verify that the logic synthesizer used the component that the designer intended to use. Additionally, components instantiated by inference can make the description less maintainable and may increase the time and difficulty of a thorough review.

Consider the following:

a. Explicitly control the I/O cells that are used and manage other “special” components in the design. Verifying inferred components with hundreds of I/O pins, multiple I/O banks with various voltage levels, and multiple I/O standards in modern FPGAs are correct is a labor-intensive and error-prone operation. Explicit definition of the I/O cells also makes the design easier to review and maintain.

(1) Generics can be used in the VHDL source text for targeting multiple device technologies, aiding in re-use.

b. Set the particulars of a specific I/O cell in the back-end design tools. For example, the VHDL design would contain the following for a 1.8-V driver:

```
u_mcd_clk: OUTBUF_LVCMOS18 port map (D   => mcd_clk,
                                     Pad => xmcd_clk);
```

while details like the pin number, driver strength, and slew rate for signal integrity are set in the back-end tools.

For FPGAs designed to use large busses, the generate statement can result in a more compact description, with less effort.

c. Disabling the logic synthesizer's I/O insertion capability eliminates the possibility that the logic synthesizer makes an unintended but legal I/O instantiation, but is in fact an inadvertent error in the design (e.g., driven signal that is not used internally to the design, perhaps a typographical error in the signal's name causing the error).

4.1.5 Avoid side effects in procedures and impure functions. Procedures and functions should rely solely on the parameters passed to it. Procedures should only modify its parameters; functions should never modify its parameters. If there are side effects in a design, the VHDL text should be very carefully analyzed and well documented.

4.1.6 The engineer may guide the logic synthesizer in many critical items in the design, such as flip-flop replication, state machine encoding, and fault tolerance (e.g., triple modular redundancy [TMR] cells, safe state machines) as follows:

a. Control the design implementation in the VHDL source text via attributes and not from an external constraint file or a logic synthesizer's graphical user interface when practical. By using attributes, the possibility that the logic synthesizer fails to use an external file (through some error) is eliminated. This also makes the job of engineers modifying or reviewing the design simpler and less error-prone.

b. Ensure an explanation of the design engineer's intent with respect to controlling items such as fault tolerance (e.g., safe state machines) is well commented in the VHDL source file. Attributes can be confusing, as each vendor may have their own attributes for items that are in IEEE 1076.6, VHDL Register Transfer Level (RTL) Synthesis; and many of the attributes are interdependent. Documenting the intent will preserve what the design engineer wanted to do in a form suitable for review by both the design engineer and the independent reviewer.

4.1.7 When describing the design, consider the following for troubleshooting, maintainability, and reviews:

a. Properly organize the textual description in logical, regular, readily understandable components (e.g., multiplexors, arithmetic logic units [ALUs], memory controller).

b. Use indention and align text so the description is neat and the layout of the text shows the structure of the design (e.g., indenting the `then` and `else` clauses and each level of an `if` statement; vertically align fields to improve readability), as shown below:

```
if rising_edge (clk)
  then if  (por_n = '0')
         then total          <= (others => '0');   -- off module
              total_internal <= (others => '0');   -- in  module
         else if (pong = '1')
                then total          <= std_logic_vector (total_internal);
                     total_internal <=                  adc_data_long;
```

```
                    else total_internal <= total_internal + adc_data_long;
                    end if;
            end if;
    end if;
```

Various tools (e.g., Modelsim®, Synplicity®, ISE®, Libero™) have built-in text editors, and some third-party editors "understand" VHDL syntax. These editors can help with formatting and using templates. It is recommended that these tools be configured to use spaces in place of tabs for indenting, as tab settings vary from editor to editor. VHDL text on the screen may look fine but will be difficult to read when sent to another engineer for review or simulation who may be using an editor with different settings.

c. Use meaningful names (some VHDL text looks like "code"). It is also a good practice to expand abbreviations used in names in the comment where the name is first declared or defined (e.g., signal or variable declaration).

d. Do not use "extended identifiers" (string enclosed between back-slash characters). Extended identifiers are not compatible with all versions of VHDL, spaces and operators reduce readability, and the extended identifier is case-sensitive (the rest of VHDL is case-insensitive).

e. Avoid "tricky" and hard-to-read elements of the VHDL language. For instance, C programmers appear to have as a goal the minimization of the number of characters, and even hold contests for this "style." The goal of a designer using VHDL for spaceborne applications is to (1) describe the design so that it is error free and can be readily proven to be error free, and (2) be maintainable, modifiable, and perhaps reusable.

f. Avoid unusual, seldom-used, and complex elements of VHDL to minimize mistakes and promote readability, reviewability, portability between tool chains, and exposure to any latent defects in any of the software tools.

g. Avoid the latest VHDL feature, unless it significantly enhances the descriptions and, if not supported by a different tool, can be readily changed. While various VHDL standards are issued by IEEE, tool vendors choose which elements of the language they will support and how they will support them.

h. Avoid recent changes to the VHDL standard that weaken type checking. While more type conversions are needed, the strong typing is useful for catching errors early.

4.1.8 Do not use "`wait`" or "`wait for`" in synthesizable designs. Of course, these constructs can be used in test benches. The use of "`wait`" statements in synthesizable VHDL (for those uses that make sense) do not offer an advantage and can be confusing. The use of `rising_edge (clk)` and `falling_edge (clk)` of clock is recommended. An example and explanation are given by Rajan (1998):

```
process begin
  wait until clk = '1';
    if en = '1' then
      q <= d;
      end if;
end process;

...
```

While it appears as though the `wait` statement describes a level-sensitive function, the `wait` statement, in fact, results in an edge-triggered element due to an implied `on` clause. In other words, the line

```
wait until clk = '1';
```

is exactly equivalent to

```
wait on clk until clk = '1';
```

This implies that the process is suspended until there is a change in value of the original signal `clk`, and it does not execute the statements after the `wait` statement unless `clk` is also '1'. This change in value represents a signal transition, which corresponds to a rising edge on the signal `clk`.

It is recommended to avoid these structures because they can rely on implied keywords and appear to have different functionality than what is implicated. `Rising_edge (clk)` and `falling_edge (clk)` are preferred. (See section 4.7 in this NASA Technical Handbook).

4.1.9 To the extent practical, keep the same name through the hierarchy; try to match the name on the board schematic exactly; and be consistent with case use for the identifier (note that some tools processing VHDL designs are case sensitive). This style makes tracing signals throughout the entire design and tool chain easier and less error-prone. If a component is instantiated several times, adding suffixes on the signal names is one technique for making the signals unique while meeting the intent of this NASA Technical Handbook. Note that some board schematic tools allow for characters in net names that are illegal in VHDL such as an asterisk to indicate the net is active-low or +/- for differential pairs. Some also allow bus subscripts to be anywhere in the net name, such as a bus of active-low resets with names like RESET[3]_N. Request the board designer comply with the following VHDL rules for net naming:

a. Begin with an alphabetic character.

b. Contain only alphabetical, numeric, underscore characters, and bus subscript characters (e.g., parentheses).

(1) Some board schematic tools use <> for bus subscripts, so exactly matching names may not be feasible.

c. If bussed, put the bus subscript(s) at the end.

4.1.10 Be liberal with use of parentheses in expressions and carefully align the terms of an expression to eliminate the need to find and follow the VHDL precedence table. This makes the design text less error-prone to write, maintain, and review.

4.1.11 It is acceptable to have multiple statements on a line to promote readability. One example is when multiple assignments need to be made for a case statement as follows:

```
when x"000" => addr <= x"5555"; data <= x"AA";
when x"001" => addr <= x"2AAA"; data <= x"55";
when x"002" => addr <= x"5555"; data <= x"A0";
```

4.1.12 Put only one signal or variable per line in declarations, entity definitions, when instantiating a component, etc., to make it easier and less error-prone to add or delete variables or signals, to change the type of a variable or signal, or to add comments describing the signals, as shown below:

```
--
-- POWER SUPPLY SIGNALS
--

   xFE_AB_PWR_ENA  : out std_logic;  -- Enable analog front end zone AB
   xFE_CD_PWR_ENA  : out std_logic;  -- Enable analog front end zone CD

   xSYNC_V1P2      : out std_logic;  -- Sync signal to 1.2VDC DC-DC convertor
                                     -- (1 MHz)
   xGOOD_V1P2      : in  std_logic;  -- Power good signal from 120 VDC DC-DC
                                     -- convertor

   xSYNC_POWER_GUY : out std_logic;  -- Sync signal to power card
```

4.1.13 Use a unique prefix for names that correspond to signals physically outside the FPGA (e.g., the I/O pad). Several different prefixes can be used for different classes of signals. Keep a legend for the prefixes at the top of the top-level entity or in a separate design document. The following example uses an "x" prefix:

```
U_rama_a18: Outbuf Port Map (D   =>  rama_a(18),
                             Pad => xrama_a18    );
```

Some engineers prefer a style with an underscore after the prefix, such as:

```
x_rama_a18
```

Either of these forms is acceptable.

Note the label attached to the instantiation follows the same convention: "`U_rama_a18`". Match the names; while adding a prefix to indicate the I/O signals likely conflicts with trying to match the corresponding board net names, it is not difficult to visually match the signal and net names.

4.2 Text Readability

4.2.1 Use underscores in names and bit strings to make the VHDL text more readable. Do not use two underscores in a row or use an underscore as the last character in the name.

4.2.2 Do not use tabs; use spaces instead to make the VHDL text readable when opened with a different editor or viewer (as when the design is reviewed), because tabs are environment dependent. Some text or "code" editors can replace, or can be configured to replace, tab characters with spaces.

4.2.3 Use the "_n" naming convention to identify signals or constants that are active low. For example, reset_n is asserted when a logical '0'.

4.2.4 Use the "_p" and "_n" naming conventions to identify positive and negative (respectively) signals of a differential pair (for example, serdes_p and serdes_n).

4.2.5 Document the definitions of registers, instructions, and their codes and parameters, etc., in the VHDL text file, and you can use block comments (see section 4.1.1 in this NASA Technical Handbook) for this purpose.

4.2.6 In general, use named rather than positional association in the component association list for generic and port maps, parameters for procedures and functions, and for aggregates for arrays and records, even when the paired names are the same. This increases the readability of the text and is less error-prone.

4.2.7 Do not leave "commented out" VHDL text in the source file without a good reason. Residual deleted text clutters the file and leads to confusion for people maintaining, modifying, or reviewing the design. If residual text remains in the file, use the "--" commenting style at the beginning of each line, and not the "/* */" delimited comment to avoid confusion for both designer and reviewer. This will help eliminate confusion when an engineer is reading and analyzing the description and the editor used does not color code commented lines. During development, it may be desirable to keep commented out text until the design approach is settled and checked out. There also may be "placeholder" text in the description. Text in the comment should be clearly identified relative to what needs to be removed or cleaned up from the "to do" list. These sections should be flagged for easy location. Special phrases in the comments such as "--FIX ME!!!!" or a long list of asterisks "--**********" are two methods that can work.

4.3 Files

4.3.1 Place each entity-architecture pair in a separate file.

4.3.2 Match the file name and entity name. For example, the entity-architecture pair signal_statistics will be in file signal_statistics.vhd, as shown below to help eliminate confusion:

```
--
-- File name: signal_statistics.vhd
--

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

    use     work.constants_pkg.all;
    use     work.components_pkg.all;


entity signal_statistics is port
      ( clk                 : in  std_logic;
        por_n               : in  std_logic;

        ...
```

4.3.3 Place documentation for each file at the top of the file, including the file name, a brief description of the entity-architecture pair and purpose, a summary of changes for each version, and any limitations to alert a user or reviewer. Putting notes at the bottom of the file increases the chance that the documentation will be missed and perhaps found later (surprise!).

4.3.4 Use a standard prefix or suffix for test benches (e.g., main_tb.vhd or tb_main.vhd) and packages (e.g., constants_pkg.vhd or pkg_constants.vhd). The prefix method groups the packages and test benches together when an alphabetical listing of modules is made. The suffix method groups the test bench along with its corresponding module when an alphabetical listing is made. For a project, be consistent in how the files are labeled. Either method is acceptable and is based on personal preference.

4.4 Packages

4.4.1 Packages are often a good way to keep constants, type information, component definitions, functions, and procedures in a convenient and easy to find location for the following reasons:

a. Saves time in searching for some constant definition in a myriad number of files.

b. Keeps common information (e.g., type and component definitions) in one location so that when there is a change, the change only needs to be made in one location, eliminating any inconsistency errors between multiple files.

c. Helps eliminate “clutter” in the design files, maximizing the engineer’s focus on the particular design architecture.

4.4.2 Use a “pkg_” prefix or a “_pkg” suffix for package names so that the files are readily identified in a listing of files. Either form is acceptable and is based on personal preference.

4.4.3 Dedicate a package to constants and type definitions for the following reasons:

a. Allows all the other entity-architecture pairs to utilize these common elements.

b. Minimizes the chance of error when system constants or types need to be modified; everything is in one place.

If constants in the design need to be changed for certain simulation runs (e.g., a one-second counter for flight that needs to be shortened for certain simulation runs), then do not “bury” constants in individual files and rely on remembering that the constants need to be restored. That is error-prone and has proven to be expensive. Instead, define the constant in a package dedicated to constants with an obvious name to make verification that no test modes are left in the design, trivial. Leave clearly worded comments and annotate the constant with a string of asterisks (“**********”) that will effectively serve as a “red tag item.”

Another method is to use a generic that defaults to the flight value but can be overwritten with the simulation value by the simulator itself. This avoids making changes to the source text and no restores are needed. The generic defaults are always used for synthesis and in simulation any time the generic is not overwritten in a given simulation run. This further reduces the chance of error.

4.4.4 Packages dedicated to component declarations are also very useful. This eliminates clutter in the design files and avoids duplication for components that are used in more than one architecture. A good practice is to have two component declaration packages: one package is for the synthesizable flight design, and the second package is for test benches and models used to emulate components other than the unit under test. If just a single package is used, there will be “extra” declarations needlessly complicating matters and adds work in going through reports.

4.5 Entity Definition

4.5.1 Align the different fields of the entity definition as shown below, making the design easier to understand, particularly when the design is being maintained, reviewed, or ported:

```
entity signal_statistics is port
 ( clk                 : in  std_logic;
   por_n               : in  std_logic;                      -- synchronized reset

   adc_data            : in  std_logic_vector (13 downto 0);

   max_data_continuous : out std_logic_vector (15 downto 0);
   min_data_continuous : out std_logic_vector (15 downto 0);
   clear               : in  std_logic;                      -- reset the
                                                             -- statistics

   max_data_second     : out std_logic_vector (15 downto 0);
   min_data_second     : out std_logic_vector (15 downto 0);
   ping                : in  std_logic;                      -- sample and clear
                                                             -- every micro frame

   total               : out std_logic_vector (47 downto 0);
   sum_of_squares      : out std_logic_vector (63 downto 0);
   pong                : in  std_logic                       -- sample and clear
                                                             -- every major frame
   );
   end entity signal_statistics;
```

Place a label after the end in entity (and component) definitions.

4.5.2 It is acceptable, and often preferable, not to segregate different types of ports (e.g., in, out, inout, etc.), and instead to logically group the ports. A blank line combined with a comment (if necessary) as to what the group represents makes the definition readable and documents it well. This is similar to how symbols were drawn for components when designing with schematics. That method has worked well for the past 50 years and there is no reason to change the philosophy. An example follows:

```
entity mem_controller_512x32x16x8 is port (
       -- WRITE PORT
          clk120                    : in   STD_LOGIC;
          por_n_120                 : in   STD_LOGIC;
          go                        : in   STD_LOGIC;
          write_data                : in   STD_LOGIC_VECTOR (31 downto 0);

       -- READ PORT GENERAL
          clk20                     : in   STD_LOGIC;
          dump_raw_bus_control      : in   STD_LOGIC;

       -- READ PORT TELEMETRY
          read_address_telemetry_inc : in   STD_LOGIC;
          read_data_telemetry       : out  STD_LOGIC_VECTOR ( 7 downto 0);

       -- READ PORT 1553/ENGINEERING
          read_address_engineering  : in   STD_LOGIC_VECTOR ( 9 downto 0);
          read_data_engineering     : out  STD_LOGIC_VECTOR (15 downto 0)      );
       end mem_controller_512x32x16x8;
```

4.5.3 When reasonable, use std_logic or std_logic_vector in the interface definition instead of unsigned, signed, etc. All tools are familiar with std_logic and std_logic_vector, and these interfaces can readily be used with modules described in Verilog™.

The intent is to keep things simple and basic to avoid potential problems or unintended behavior. One engineer had a signal passed between modules of type Boolean. One module was synthesized `TRUE` and `FALSE` one way; another synthesized module had reverse assignments for `TRUE` and `FALSE`.

4.5.4 Record types (a composite type whose values consist of named elements) can be used on the interface if such use has a significant benefit. These record types permit handling related elements of different types as a single object. An advantage of using records to pass information is that it becomes less cumbersome to add or delete signals, or change a signal's type, without needing to update multiple entity definitions. Another advantage is that entity definitions and instantiations are less cumbersome and bulky. The record definition should be placed in the package holding constants and types so a change needs to be made in only a single spot, giving consistency between components. A disadvantage of using records is that care must be taken that all components that use signals of this type are updated. Another disadvantage is that synthesis tools may flatten the record into an array of std_logic_vector and remove the field names from the netlist. This may make timing analysis difficult since all signals have to be referenced according to their offset in the record, which makes the netlist much less readable.

4.5.5 The buffer mode should never appear in the port clause of the model's top-level entity declaration. Use an out port with an internal signal or properly use an `inout` port.

4.5.6 Vertically align comments at the end of the line for I/Os, as needed.

4.5.7 If a VHDL entity-architecture pair is machine generated (e.g., by a macro generator), it is a good practice to reformat the entity definition in accordance with this NASA Technical Handbook so that instantiation of the component is consistent with entity-architecture pairs developed by the designer. This aids in minimizing or eliminating errors and enables an efficient review of the design.

4.6 Component Instantiation

4.6.1 Label the instantiated component with a "`U_`" prefix to the entity name, treating the VHDL component similarly to a component on a circuit board. If there are multiple instantiations of a particular component, use suffixes to make them unique. The style for instantiating a component is similar to the entity definition in section 4.5 of this NASA Technical Handbook.

4.6.2 Use named association to minimize errors when maintaining or porting the description. This makes mistakes easier to catch by the designer and a reviewer, as shown below:

```
U_memory_ram_control_a: memory_ram_control port map (
    clk_10mhz              => clk_10mhz,
    por_n                  => por_10mhz_n,
    r_w_mode               => ram_a_rw_mode,
    memory_mode            => memory_mode,

    -- From science ingest
    sci_write_cmd          => sci_write_cmd_ram_a,   -- Pulse Command
```

4.6.3. Include usage information as a comment embedded in the instantiation text to make the description clearer for the designer, reviewers, maintainers, and porters. See example below:

```
U_mem_controller_512x32x16x8_A: mem_controller_512x32x16x8 port map (

-- READ PORT 1553/ENGINEERING
   read_address => dma_address ( 9 downto 0), -- in   std_logic_vector ( 9 downto 0);
   read_data_   => read_DSP_data_A);          -- out  std_logic_vector (15 downto 0)
```

4.7 Clocks

4.7.1 Use the `rising_edge (clock)` or `falling_edge (clock)` function calls instead of the `clock'event` attribute and a condition for a number of reasons, including the following:

a. The `rising_edge` or `falling_edge` functions are easier to read.

b. While the two formats are similar, they are not identical for clock signals that are std_logic_vector (which is almost always used). Using a function call is preferred for simulation because a function call only detects an edge transition (0 to 1 or 1 to 0) but not a transition from X to 1 or 0 to X, which may not be a valid transition.

4.7.2 It is acceptable to use both rising and falling edges of a clock in your design. While some "coding standards" prohibit such constructions, this clocking technique offers advantages (e.g., lower noise, lower power, optimally allocated timing margins) in certain situations.

4.7.3 There are several acceptable styles for naming clock signals when describing components.

One style uses a generic name such as "clk." When the component is instantiated, the higher level can associate a particular clock with the port, such as "clk_40MHz." This makes the component reusable for different projects with different clocks with no modifications.

Another style uses the actual speed of the clock in the component's entity, such as "clk_40MHz." If used for another project, then the description must be edited.

Either method is fine. Indeed, a mixture of methods is often preferred.

Generic "clk" names are good for simple components such as synchronizers; there is no confusion and there does not need to be multiple components, all with different names.

For more complex components, particularly when multiple engineers are working, explicitly naming the clock with the operating frequency is preferred. The major benefit is elimination of confusion in systems with multiple clocks, minimizing the chance of inadvertent and hard to see clock domain crossing errors.

4.8 Flip-Flops

4.8.1 If the technology being used has hard macros for flip-flops (e.g., many but not all FPGAs), understand all the functionality in the module and the polarity of the signals when writing the VHDL text, as follows:

a. While the logic synthesizer will aid and correct any polarity errors (e.g., some flip-flops asynchronous inputs are active high, others are active low), understanding and exploiting the functionality can make a more efficient and higher speed design (e.g., using the 2:1 multiplexer (mux) on the "nose" of the RTAX-S R-cell).

b. Some technologies will be awkward and bulky if both asynchronous presets and clears are used (i.e., built-in support of asynchronous preset and clear per flip-flop should not be assumed). Other technologies support such operation in hardware, and some technologies may have precedence logic built in. There is no "standard flip-flop."

Using both asynchronous presets and clears is reasonable and natural for certain situations, although often not needed. If the functionality is not clear (e.g., precedence in the hard macro), ensure that comments clearly document the behavior of the description. Comments should also document any technology dependencies, as this style likely will not be portable between FPGA technologies.

c. The style of writing a description of a flip-flop, as noted above, should match the underlying hardware (i.e., the specific FPGA logic elements) and not treat the design task as writing software. A good example of this is the RTG4 architecture, where the use of asynchronous inputs to flip-flops are restricted. When such asynchronous inputs are used, they have to be dealt with through the entire tool chain to ensure proper radiation hardness to single-event transients (SETs) and timing performance. For example, reliance on the CLEAR having precedence over the PRESET while both are asserted in RTAX-S devices will not be portable.

The traditional model of a flip-flop is described as follows:

```
async_proc : process (CLOCK, por_n)
begin
  if (por_n = '0')
    then dbuff0  <= (others => '0');
         DOUT(0) <= '0';
    else if rising_edge(CLOCK)
           then dbuff0(15 downto 0) <= dbuff0(14 downto 0) & DIN(0);
                DOUT(0)             <= dbuff0(15);
           end if;
    end if;
end process async_proc;
```

For devices more amenable to synchronous sets and clears, write the model as follows to enable good use of the hardware:

```
sync_proc : process (CLOCK, por_n)
begin
  if rising_edge(CLOCK)
    then if (por_n = '0')
           then dbuff0  <= (others => '0');
                DOUT(0) <= '0';
           else then dbuff0(15 downto 0) <= dbuff0(14 downto 0) & DIN(0);
                     DOUT(0)             <= dbuff0(15);
           end if;
    end if;
end process sync_proc;
```

For most, but not all, the use of the synchronous reset will not affect system functionality (as many FPGAs need time to start) and will result in increased noise immunity.

d. A style issue is whether to write a description that contains both asynchronous and synchronous resets, or to write two separate descriptions. Both styles are acceptable. Some comments about this issue follow but are not comprehensive, as the technical issues fall outside the scope of this NASA Technical Handbook:

(1) Clearly, a single description makes certain maintenance easier and less error-prone, as changes are made once, in one location. Also, a single description may make reusing the description easier, as there is less work (e.g., changing the name of the component being instantiated). Libraries are also less cluttered.

(2) Making the description “configurable” raises some of the following issues:

A. For an intellectual property (IP) core that has been certified and/or third-party tested (e.g., a MIL-STD-1553B, Digital Time Division Command/Response Multiplex Data Bus IP core that passes certification), certification and testing have to be performed on all variants.

B. Subtle errors may be induced if changing the reset method is controlled by a "switch"; resets will be applied at different times, which may affect functionality (e.g., all resets are not power-on resets).
C. It may be error-prone configuring all components for the appropriate reset in a large design when using a "dual reset" model; each instantiation has to be found and configured properly. If separate models are used, checking is trivial, as only the list of modules need to be examined.

4.8.2 This section intentionally left blank.

4.8.3 Enable and case expressions for flip-flops as follows:

a. Several styles for writing an equation for a flip-flop enable exist. The choice of style is based on the designer's preference, the particular situation, and target hardware architecture. Acceptable methods follow:

(1) One method is to put the logic equation for the enable inside of the if-then-else statement inside of a process. This method has the advantage of having the eye focus on one spot of the VHDL source code and not having to jump back and forth.

(2) Another method is to put the logic equation for the enable outside of the flip-flop process. It can be put in a separate process or in a concurrent statement such as a conditional signal assignment. This method has the advantage of having a signal that can be seen directly in the simulation. This method has the disadvantage of having the logic for the `if-then-else` statement away from the `if-then-else` statement, resulting in a non-linear flow to read the VHDL description.

b. The choice of using a conditional signal assignment statement or a separate process is a matter of personal preference. The conditional signal assignment is "smaller" than a process, but for some engineers it is less intuitive since the "output" in the text precedes the condition which selects it. The process method is bulkier but can use `if-then-else` statements where the condition precedes the "output."

c. Use of either method requires a careful alignment of the terms of the logic equation. Heavy use of parentheses makes the logic equation more readable, more reviewable, maintainable, and less error prone. This technique eliminates the need to rely on memory or look up the VHDL precedence rules and the laborious task of parsing a complex, hard-to-read relation.

d. The case statement expression is similar to the flip-flop enable expression. If the expression is too complex, compute the expression using a variable (preferably) or a signal (which should be avoided in synthesizable descriptions). An example follows:

```
reg1_proc : process (reset_n, clk5mhz)
begin
if (reset_n = '0')
  then mysignal <= '0';
  else if rising_edge (clk5mhz)
        then if ((ASig = '1') and (BSig = '0')) or
                ((CSig = '1') and (DSig = '0'))
               then mysignal <= DSig;
               end if;
        end if;
  end if;
end process reg1_proc;
```

Enable embedded in if-then-else statement in the process describing a flip-flop. Note careful alignment of the logic equation text and use of parentheses to aid in understanding the structure of the logic:

```
enable <= '1' when (ASig = '1') and (BSig = '0') else
          '1' when (CSig = '1') and (DSig = '0') else
          '0';

     reg1_proc : process (reset_n, clk5mhz)
     begin
     if (reset_n = '0')
       then mysignal <= '0';
       else if rising_edge (clk5mhz)
              then if enable= '1'
                     then mysignal <= DSig;
                     end if;
              end if;
       end if;
     end process reg1_proc;
```

Enable external to the process implementing the flip-flop in the process describing a flip-flop. Many engineers prefer to put the enable's logic equation in a separate process and use an `if-then-else` *statement.*

4.9 Processes

4.9.1 Use a descriptive label for each process and use the label in the end process statement.

4.9.2 Try to avoid variables in synthesizable descriptions, in general. Variables in clocked processes can produce unexpected registers. Reading a variable before it is assigned a value implies reading the old value (a flip-flop or register). If a variable is always written prior to being read, no flip-flop or register is implied for that variable.

4.9.3 Avoid latches, in general. Sometimes the transparent latch function is useful, but generally latches in a design are a result of an error in writing the description (always good to check the synthesis reports for warnings and resources, as well as resource usage reports from

the back-end tools). Latches are often inferred because a design does not assign a value under all conditional statements. To avoid having the logic synthesizer inferring a latch, ensure that a value is assigned under all conditions, use a default value appropriately (which also can eliminate clutter), or use an `else` (instead of an `elsif`) for the final branch of the `if-then-else` statements.

4.9.4 Consider that the process' sensitivity list can be replaced with the keyword "`all`" (see VHDL-2008, section 11.3). This can be a convenience, as one does not have to type in the sensitivity list; it will be automatically constructed based on the contents of the process. This can also lead to errors, as many processes are written using the copy-paste-modify technique. A signal buried in the process that was inadvertently not modified will automatically be put into the sensitivity list. If an explicit sensitivity list is required, the software tools may either flag an error or issue a warning, alerting the design engineer to carefully check the design.

4.10 State Machines

4.10.1 Design state machines with one, two, or three processes. This NASA Technical Handbook does not recommend any one particular structure as the "best" structure for a particular application. The number of processes is application dependent and is often a matter of personal preference.

4.10.2 Note that VHDL states are logical states and not physical states. As such, the "`others`" clause of a case statement does not cover states that are not present in the VHDL text. Some logic synthesizers with appropriate attributes may use the "`others`" clause to cover all physical states. The following is a brief and not comprehensive introduction to state machine synthesis, which is in general outside the scope of this style guide:

a. For example, if there are five (5) states and an enumerated type is used for the state variable, and the logic synthesizer generates a binary encoded state machine, the VHDL synthesizer generally will not implement transitions from the three "undefined" states to some desired state (as written in the others clause).

b. If the logic synthesizer uses a one-hot encoded state machine using the same 5-state enumerated state machine described above, the vast majority of states (2^5 – 5 = 27, more if 1 or more finite state machine [FSM] bits are replicated) will be illegal states and "uncovered" by the others clause in the case statement.

c. One can also use hard-coded constants to represent the states and you may carefully define transitions from each of them. This is no panacea, as the logic synthesizer's optimization module may detect that the states are unreachable and eliminate the transitions. Again, understand, use, and document the attributes and synthesizer settings to aid in review, maintenance, and reuse.

d. Numerous design techniques exist to ensure that the state machine is robust and does not lock up or do something undesired. These techniques are not "VHDL style" but are in fact

design techniques (including synthesizer attributes and settings) and are outside the scope of this NASA Technical Handbook. What may look like a robust state machine in the VHDL text may not result in the hardware desired. Use of logic synthesizer attributes can be tricky, and can be product, version, and revision-level dependent. Inspecting the generated logic with, for example, a schematic viewer can be required for design verification of critical circuits.

4.10.3 The 1076.6 "FSM_COMPLETE" attribute (reference IEEE 1076.6, VHDL Register Transfer Level (RTL) synthesis, section 7.1.10) provides for transitioning out of an illegal state. This attribute may not be implemented by a particular logic synthesizer. As such, if this VHDL feature is used, it should be well documented.

IEEE 1076.6 briefly describes the intent of this attribute as follows:

> NOTES
>
> 1—FSM_COMPLETE augments the state machine hardware with transitions that allow it to recover if an invalid or unused state value occurs, as might happen because of a power glitch or single-event upset.
> 2—Typical ways to make a default state assignment are by the others clause of a case statement, the else clause of an if statement, or by an initializing value unconditionally assigned to the state register.

4.10.4 Enumerations I – Make the first enumeration the default or benign state when defining an enumeration type. There is no guarantee that this will have an impact on the synthesis tool but is a reasonable guideline. Synplicity®, for example, has a "safe" encoding attribute. If this VHDL feature is used, it should be well documented. Synplicity's® reference manual states:

> If recovery from an invalid state is a concern, it may be appropriate to use this encoding style, in conjunction with one hot, sequential or gray, in order to force the state machine to reset. When you specify safe, the state machine can be reset from an unknown state to its reset state.

4.10.5 Enumerations II – Consider using user-defined enumeration encoding if supported by the logic synthesizer if you are using enumerations and want to control the particular state assignments so that a particular encoding is used without using constants. Controlling the encoding gives the designer the ability to control individual bits and the power to design the state machine with a Hamming distance of the designer's choosing. While IEEE 1076.6 calls this attribute "enum_encoding," Synplicity® calls this "syn_enum_encoding" which is inconsistent with the IEEE standard. For example:

```
type state_type is (S0, S1, S2);

attribute syn_enum_encoding: string;
attribute syn_enum_encoding of state_type : type is "001 010 101";
```

As with many VHDL features, each tool vendor may do as they wish. As we can see with this example, the same attribute may have a different name in different tools, making the design more complex and error prone.

A slightly modified version of the example above was run through a logic synthesizer. While the synthesizer acknowledged the designer's intent and, in this case, wanted a machine with three flip-flops and a particular encoding, the synthesizer decided it knew best and mapped the three-flip-flop design into two flip-flops, using an encoding of its own choosing. Rather than using yet more attributes to get the synthesizer to "listen" and "grant the desires of the engineer," it is recommended that this VHDL feature not be used, and simpler methods, such as constants be used. While this is a bit more work, and a bit clumsier, it is easier to get through the tool chains while maintaining the designer's intent.

4.10.6 Enumerations III – Some texts and guides state that defining the enumeration such that the number of enumerations is a power of 2 (apparently when using a binary or gray encoding scheme; all one-hot encodings have unused states) and using the others clause in a case statement will eliminate the possibility of lockup from the state machine reaching an illegal state (e.g., from a single-event upset (SEU) or "brown out"). While theoretically this is correct, in practice a logic synthesizer's optimizer can determine that certain states are unreachable and not implement "unneeded" logic. One way the style of the description may (no guarantee) avoid this issue is to make separate reset and idle states with as many intermediate states from the reset to the idle state to make the total number of states a power of 2.

4.10.7 Consider assigning default values to outputs derived from the FSM before the case statement. This helps prevent the generation of unwanted latches and makes it easier to read because there is less clutter from infrequently used signals. For example, the "`done`" signal may be asserted for one state in the sequence. With the use of the default value, "`done`" will appear in the text only where it is asserted. For example:

```
else if rising_edge (clk_20mhz)
      then done            <= '0';
           address_int_old <= address_int;
           case state is
              when home    => address_int <= unsigned (left_limit_fine_search);
                   if (go = '1')
                     then state <= fetch_rx_left_data;
                     else state <= home;
                     end if;

                                        …

              when pause_10             => state <= finished;
              when finished             => state <= home;
                                           done  <= '1';
              when others               => state <= home;
```

4.10.8 Simulation and Debugging – Enumerated types are very nice when simulating the state machine, as the state name is readable on the waveform display, for example. However, a

"translation" process is required to bring the state variable out to pins in a physical device during debug operations; and the particular encoding must be known or controlled.

Overall, if the enumerated type is "safe" for the application (synthesizer assigned values and transitions are acceptable; other directives or circuits result in a sufficiently safe state machine for the application), it is easier to work with (and why programming languages and HDLs have enumerated types) since time is more efficiently spent in the simulator checking out the design with enumerated names than in real hardware debugging binary values. The "better" style, either enumerated or controlling the values representing each state, depends on the application and the individual designer. See also section 4.10.6, Enumerations II, in this NASA Technical Handbook.

4.10.9 "Safe" State Machines with Correction – Synopsys® can encode a state machine using a Hamming-3 code. This is briefly described in Synopsys®'s white paper; it is only supported in their "premier" edition. Different vendors may have different support for fault tolerance (e.g., no lockup, TMR of flip-flops, auto-correction capability). Comments should clearly describe the desired behavior and specify when a specific tool's capability is being used. Also note vendors, as they update their software, may add, subtract, or modify features in each of their versions.

4.11 Arithmetic

4.11.1 Use `ieee.numeric_std` and avoid using old Synopsys® packages like `ieee.std_logic_arith`, `ieee.std_logic_signed`, and `ieee.std_logic_unsigned`. Do not mix the two types of packages. See also section 4.11.6 in this NASA Technical Handbook for a further discussion on the use of different packages.

4.11.2 Do not use unconstrained integers, as synthesis tools may create a 32-bit resource. Instead, specify a range, which also increases the chance of tools flagging what may be an error in the description (e.g., out-of-range error).

4.11.3 Consider using a subrange of integers for counters. This makes the design easier to size with generics, and the counter can be used to directly index VHDL arrays.

4.11.4 For multiplication and division, consider using explicit shifts and adds, if needed. While synthesizers should recognize the structure and optimize the logic, there is no guarantee that they will do so; or the tool may infer and generate hardware that the designer does not want.

Note that some synthesizers will automatically use hard digital signal processing (DSP) macros for multiplications greater than a certain width. This width may be settable to aid in controlling the target hardware. A Synopsys® Application Note ("Inferring Microsemi SmartFusion®2, IGLOO2™, and RTG4™ MACC Blocks,") states, for example:

```
By default, the tool maps all multiplier inputs with a
width of 3 or greater to MACC blocks. If the input width is
```

```
smaller, it is mapped to logic. You can change this default
behavior with the syn_multstyle attribute (see Controlling
Inference with the syn_multstyle Attribute, on page 4).
```

A different style is manually instantiating the hard macro, which provides the designer the most control and is the clearest for maintainers and reviewers to understand. The description is bulkier and harder to modify.

4.11.5 Be careful of using "=" when comparing counts, as "<=" or ">=" are more tolerant to upsets. For example, a 3-bit up counter with an "if = 5" conditional would take longer to recover if the count skipped from 2 to 6 as it would have to roll over after 7, then count up to 5. For large counters, the delay until recovery can be unacceptably long, adding problems. A more robust implementation may be "`if x >= 5 then …`"; an upset circuit can recover more quickly. See section 4.12.2 in this NASA Technical Handbook.

4.11.6 The `numeric_std_unsigned` package can eliminate the need to convert to "signed" reducing the clutter in the textual description. Conversely, using the bulkier style with conversions, explicitly preserves the intended use of the signal. In general, VHDL conversions between numerous types are bulky and awkward, but provides the strongest type checking by the tools. Either method is acceptable, and comments should make any non-obvious text explicit and clear.

4.12 Declarations and Constants

4.12.1 Put clock domain information into the signal name for classes of signals that can be used across different clock domains. For example, use "`clk5mhz`" and "`clk33mhz`" rather than "`clk1`" and "`clk2`". Also, use "`por_5mhz_n`" and "`por_33mhz_n`" instead of "`por_1_n`" and "`por_2_n`". This will minimize confusion, reduce errors, and make the design more readable, reviewable, and maintainable.

4.12.2 Use subtypes to constrain the range of a type. For example:

```
subtype Int0_5_type is integer range 0 to 5;
```

Constraint errors can be detected at compile and/or run time in the simulator. The subtype is compatible with the base type (a new type is not created). During synthesis, the subtype constrains the size of any implied registers.

4.12.3 The leftmost bit of an array should be the most significant bit, regardless of the bit ordering. Generally, the rightmost bit is bit 0. However, in some architectures and specifications (MIL-STD-1553B being a widely used one), the most significant (and leftmost) bit is bit 0, and the least significant (and rightmost) bit is bit 15. Clearly comment/document any non-standard usage.

4.12.4 The index ordering (using `to` or `downto`) of the model's top-level entity port clause's signals should be identical to the data sheet of the component being modeled, to minimize confusion.

4.12.5 For bit string literals:

a. Write hexadecimal bit string literals with uppercase characters (A-F) for readability. VHDL is not case sensitive.

b. Use the "`D`" format (VHDL-2008, section 15.8) for decimal constants with the length value (which precedes the base specifier). For example, `12D"13"` is equivalent to the 12-bit literal `B"0000_0000_1101"`. Not all tools accept this VHDL 2008 feature, such as ISE®.

c. Use the length specifier (VHDL-2008, section 15.8) for hexadecimal bit string literals whose bit length is not an integer multiple of 4. Note that there are unsigned (e.g., X and UX) and signed base specifiers (SX which can be used for sign extension). For example, `18X"1234"` is equivalent to the 18-bit literal `B"00_0001_0010_0011_0100."`

d. Avoid using a length for signals, variables, and literals that do not match the hardware. In the example in the section above, 20-bit logic could be used; then use the bit string literal `X"01234"` and rely on the logic synthesizer to eliminate the two "unused" upper bits to produce an 18-bit design. This would be confusing for a reviewer or someone picking up the logic for maintenance, as the VHDL description would not match what the hardware is doing (or intended to do). This is an error-prone method and, if used, should be applied carefully and well documented. For example, the logic simulator may use 20-bit logic and not detect an out-of-range error for the intended 18-bit value.

4.12.6 Avoid dependence on implementation of defined limitations, such as a 32-bit limitation on the integer and time types. Problems may arise if a vendor changes their defaults, or if the description is used by another tool, perhaps during review or reuse.

4.12.7 As a general rule, do not initialize ports or signals for a synthesizable design. There are valid exceptions to this guideline, and these should be used with care. For example:

a. A divide-by-two flip-flop is often used to condition a board-level clock, and there is no reset applied to that flip-flop. Once powered up, a real flip-flop with divide-by-two logic will toggle regardless of its power-up state. However, a simulated flip-flop at the beginning of a simulation will go from an unknown state to the same unknown state. It is quite awkward to define a special reset just for simulation so that the simulation will "start." Using simulator force commands or their equivalent in a test bench that might have to reach into the hierarchy (e.g., Modelsim's® "spy") is also awkward and makes the model simulator dependent.

b. For some FPGA technologies, there is a known start-up state for flip-flops, and different mechanisms are used in different technologies to accomplish this goal. One may think that by initializing the flip-flop in simulation, this will accurately model the hardware. However,

it is not a good design practice to rely on such initializations, as these are not verifiable in behavioral simulation. More importantly, if a logical reset is asserted after power has been applied, the flip-flop will not be reinitialized.

Another consequence of relying on the initial state of the flip-flop is that there may be subtle issues with the initialization process as has been found in some devices under certain conditions, resulting in the need for a redesign or having a latent error in the design escape the testing and verification efforts.

4.13 Miscellaneous

4.13.1 Use a common component or components for synchronization functions. This makes it easier to add synthesis constraints to prevent logic replication or reductions. Along with properly modularizing the design, this style enables self-checking and reviewing to be less error prone.

4.13.2 Vendors often supply "parameterizable macro generators" that will generate a VHDL description for a particular function (adder, multiplier, Random Access Memory (RAM), first in first out (FIFO) memory, etc.) that is optimized for the target technology. Another approach is to write the design description in "pure VHDL" which makes the design more portable. Either style is acceptable.

The macro generator will generally produce more efficient designs faster and more accurately than the hand-written VHDL design.

The hand-written VHDL design is more portable and promotes reuse, but this style puts a larger burden on the logic synthesizer as well as the design engineer for the initial test and verification efforts.

Often using the vendor's macro generator for RAM components is preferred, as the details of each RAM module is highly dependent on the technology being used. With more standard functions such as high-speed adders, coupled with the fact that most FPGA technologies having built-in carry chains, the high-speed adder function is typically best handled by the logic synthesizer, which will also produce a cleaner description. The choice of style is application and macro-generator dependent, as macro generators may be optimized for a particular parameter not suitable for the application.

4.13.3 Register duplication is often something to be avoided, and vendor-specific directives are often given to the logic synthesizer to not duplicate any registers. This prevents a part of the state machine from having two different values at the same time for the "same" signal due to an error such as an SEU or power transient.

However, register duplication is sometimes used; and the VHDL text is explicitly written to duplicate registers. One example may be distributing a signal in a very high-speed design to different parts of the FPGA or ASIC. In such cases, the duplicated registers should be described

such that the flip-flop clocks in new data on each clock cycle, and only one flip-flop's output feeds back into the state machine.

4.13.4 Partitioning – When partitioning a design into various blocks, it is often a good design practice to have registers at functional or modular boundaries. If well designed, this can present a simple interface between blocks and minimize the coupling between the design blocks. For example, if the two blocks are in different clock domains, the receiving domain can present a set of flip-flops that clock in data from the transmitting domain on the leading edge of a pulse. If the blocks are in the same clock domain, critical timing in one block will have a minimal effect in timing on the other block. Thus, the design of the transmitting domain is simple and decoupled from the implementation of the receiver logic; and design complexity is minimized.

4.13.5 Resource Sharing – When writing the VHDL text, keep in mind the hardware structures that will be generated by the logic synthesizer may be dependent on the style of the written description. Use comments to document the design intent.

For structures that take substantial or critical resources, such as DSP or arithmetic blocks that are used in several places in the description, determine if these structures should be duplicated for higher performance and resource usage, or if these resources should be time-shared for a more compact design. The size and complexity of the multiplexing and control logic for a shared approach need to be traded off against the size and performance of the manually duplicated structures.

4.13.6 Performance-Driven Description – Write the VHDL text describing the precise circuitry you want when dealing with timing-critical signals. An explanatory comment should be included for any subtle coding techniques that are used, or if a manual examination of the netlist using a netlist viewer should be performed.

4.13.7 Documenting Changes and Version Information – Once the design is under configuration control, such as at the Critical Design Review or at some other reasonable time, the VHDL description should include a list of changes, the areas that are changed should be flagged, and an identifying label should be maintained. With the popularity of reprogrammable FPGAs, maintaining such information is important for being able to recreate and troubleshoot problems that have occurred in the laboratory, during integration and test, or during flight.

Large IP blocks, either procured or written in-house, can have similar version information and description of changes.

Following is a simple and effective method, Version Identification (ID), for controlling design changes; it may be tailored for a particular project:

a. At the top of the constants package, a 16-bit version ID is defined. The documentation should include the version ID, the date, the name of the engineer that made the change, and a brief description of the change. Each time the design is revised, a new version ID and additional ancillary information is added, and the previous information is kept in an ongoing list or log.

Along with being human readable at the top of the constants package, the version ID should be machine-readable. This is important for being able to understand any issues or problems, as the test equipment should be programmed to read out and log the FPGA (and IP) version ID(s).

b. At each point where the VHDL description is modified, add a comment that includes the Version ID and a detailed description of the changes. This makes the design electrically searchable for changes. Optionally, add the date of change and identify who made the change.

APPENDIX A

REFERENCES

A.1 PURPOSE

This Appendix provides additional guidance information for user reference. This Handbook contains much original material. Some of the material was derived from the references listed below.

A.2 REFERENCE DOCUMENTS

Reference documents may be accessed at https://standards.nasa.gov or obtained directly from the Standards Developing Body, other document distributors, or the office of primary responsibility.

"Actel HDL Coding Style Guide," Actel Corporation.

Brooks, F.P., The Mythical Man-Month (Addison-Wesley, 1975), p. 180.

Essential VHDL: RTL Synthesis Done Right, Sundar Rajan, 1998.

"Inferring Microsemi SmartFusion2, IGLOO2 and RTG4 MACC Blocks," Synopsys® Application Note, May 2016.

"Programmable Logic Device Coding and Design Document (CDD)," GRC-PLD-CDD, March 22, 2013.

"Routing Resources," Microsemi, 2019.

VHDL Coding Styles and Methodologies, 2nd Edition, Ben Cohen, Kluwer Academic Publishers, 1999.

"VHDL Math Tricks of the Trade," Jim Lewis, SynthWorks Design Inc., MAPLD International Conference, 2003.

"VHDL Modelling Guidelines," ESA, September 1994.

VHDL-2008, IEC 61691-1-1, Edition 2.0 2011-05, Part 1-1: VHDL Language Reference Manual, IEC 61691-1-1:2011(E), IEEE 1076-2008.

IEEE 1076-2019, VHDL Language Reference Manual, IEEE Computer Society.

VHDL Register Transfer Level (RTL) Synthesis, IEC 62050, IEEE 1076.6, First Edition, 2005-07.

VHDL-2008, Just the New Stuff, Ashenden and Lewis, Elsevier, Inc., 2008.

IEEE 1076.6, VHDL Register Transfer Level (RTL) synthesis.

MIL-STD-1553B, Digital Time Division Command/Response Multiplex Data Bus

MSFC Form 4657, Change Request for a NASA Engineering Standard.