VHDL Capture Guidelines

February 5, 2014

Charles Fulks
Intuitive Research and Technology Corporation
Huntsville, Alabama

Abstract

This document sets guidelines for the development of digital designs (circuits) and their subsequent capture with a Hardware Description Language (HDL); specifically, VHDL. This document is a guideline intended to be enforced by the design team with appropriate engineering judgement. *Strict adherence by all Field Programmable Gate Array (FPGA) or Complex programmable logic device (CPLD) code is not required.* The main purposes of these guidelines are to reduce the probability of logic errors and to enhance VHDL portability among designers.

This document is intended for digital design engineers with experience in programmable logic (FPGA, CPLD) design.

Comments are welcome at fpga@irtc-hq.com

Intuitive Research and Technology Corporation
VHDL Capture Guidelines

Table of Contents

List of Figures

List of Tables

Source Listings

Intuitive Research and Technology Corporation
VHDL Capture Guidelines

1 Preamble

1.1 Acknowledgments

IEEE 1076 specifies the VHDL language. This document is a collection of opinions related to minimizing errors when capturing designs and test benches with VHDL. The author compiled opinions from almost every engineer with whom he has worked.

1.2 Design Process

While this document does not define a programmable logic development process, it is prudent to comment on the philosophy of good, value added design process.

The digital design must be well defined prior to attempting to capture it in HDL. The level of detail is debatable; however, if you cannot describe your design with timing diagrams, block diagrams, and some text, you are not ready to describe it with HDL accurately. *In short, there is more to the story than the source code.* A process provides a disciplined and documented way to capture critical design artifacts.

An HDL simulation is extremely valuable in verifying a design, but it should not be used as a crutch to assist in the design process. Don't create a rough draft of a design, then simulate and modify until it works. This leads to a lack of understanding of how and why it works. It also leads to the introduction of latent defects.

Experimental HDL to understand a syntax or principle is not product development and does not fall under good design process; hack away! Don't do it in the product development folder and *do not ship experimental code in a product.*

When you design a digital circuit you almost always sketch a block diagram, a timing diagram, and other useful representations of the architecture. It is little effort to combine these into a Design Document and discuss it with a colleague prior to capturing it with HDL. Explaining your design helps you find your errors early[1] in your design process. The Design Document provides *evidence of understanding* and is an invaluable aid when you need to modify the design in the future. *Any design not accompanied by documentation is not acceptable.*

1.3 Synchronicity

Synchronous circuit design and clock domain crossing are addressed in the design phase of the FPGA development process. This document is intended to address the design capture[2] phase of the process.

A concise definition of synchronous circuit design was presented by Bob Zeidman in *Introduction to CPLD and FPGA Design*[4] (section 4.3) at the Embedded Systems Conference in June 2006.

1. Synchronous design simply means that all data are passed through combinatorial logic and flip-flops that are synchronized to a single clock.
2. Delay is always controlled by flip-flops, not combinatorial logic.
3. No signal that is generated by combinatorial logic can be fed back to the same group of combinatorial logic without first going through a synchronizing flip-flop.
4. Clocks cannot be gated - in other words, clocks must go directly to the clock inputs of the flip-flops without going through any combinatorial logic.

[1] Read: before long hours of debugging and rework.
[2] In other words, the entry of the design into a computer readable, synthesizable, simulatable form.

5. One clock for the system. Do not clock entities or processes with outputs of other entities or processes.

1.4 Tool Warnings

As a goal, maintain zero compiler, synthesis, and Lint warnings. This is not always practical, especially in code re-use situations. Remaining warnings must be deemed no impact on a rule-by-rule basis. This can be a significant issue when vendor Intellectual Property (IP) causes several hundred synthesis warnings. An important warning may be intermingled and is easy to overlook.

As a general rule, the VHDL compiler should not be used to catch syntax errors. One could ask, "If you are not concentrating on VHDL syntax enough to get it correct, what are you thinking about?"[3]

1.5 Code Re-use

Re-use requires proof that the old code works as intended in the new application. This is more than running a few simulations. In order to reuse a design you must first ensure that the *design* is appropriate for the new application. Only then is it appropriate to verify that the *VHDL capture* functions correctly in the new design.

2 General Coding Style Discussion

A typical counter is described in Listing 2.1. This style contains multiple nested `if` statements that may add confusion and additional opportunities for error. One alternative is to separate the sequential (clocked) logic from the combinatorial logic, as in Listing 2.2. Keep in mind, however, that this places a significant burden on synthesis tools to resolve and merge the two logics into a single Configurable Logic Block (CLB) with a fixed underlying architecture. Therefore, the priority of the designer is to *adhere to the manufacturer recommended coding styles* for FPGA HDL capture.

Listing 2.1: Example counter

```
1  -- counter
2  p_count : process( clk, aclr )       -- The label is: p_<primary signal>
3  begin
4    if ( aclr = '1' ) then             -- Asynchronous reset
5      count <= 0 after TCO;            -- Include clock-to-output for simulation
6    elsif rising_edge( clk ) then      -- clock input
7      if ( srst = '1' ) then           -- Synchronous reset
8        count <= 0 after Tco;          -- Include clock-to-output for simulation
9      elsif ( ce = 1 ) then            -- clock enable input
10       if ( count >= MAX_COUNT ) then
11         count <= 0 after Tco;        -- load condition is typically first
12       else
13         count <= ( count + 1 ) after TCO;
14       end if;
15     else
16       count <= count;                -- No change if enable is low
17     end if;
18   end if;
19 end process;
```

[3] A valid counter-argument is: "I'm focused on design details."

Listing 2.2: Separate sequential and combinatorial logic

```
1   -- counter
2   p_counter : process( clk, aclr )
3   begin
4     if ( aclr = '1' ) then              -- Asynchronous reset
5       count <= 0 after TCO;             -- Include clock-to-output for simulation
6     elsif rising_edge( clk ) then       -- clock input
7       if ( srst = '1' ) then            -- Synchronous reset
8         count <= 0 after TCO;           -- Include clock-to-output for simulation
9       elsif ( ce = 1 ) then             -- clock enable input
10        count <= count_next after TCO;
11      end if;
12    end process;
13
14    count_next <= ( count + 1 )      when ( count < MAX_COUNT ) else
15                  ( others => '0' ) when ( count >= MAX_COUNT ) else
16                  count;  -- always include the default case
```

3 VHDL Capture Guidelines

"Wherever two or more competing rules would be similarly able to prevent bugs but only one of those rules can be enforced automatically, the more enforceable rule is recommended." Michael Barr[2]

3.1 File Name Selection

Rule:
A component file name shall match the entity name.

Reasoning:
Several industry tools look for file names that match the entity name.

Rule:
The entity name should convey the one thing that entity does. If the name implies multiple functions, then the entity should be divided into multiple components.

Reasoning:
Mixing multiple functions increases the opportunity for confusion and error. If you can't name the entity properly, it is unlikely that you understand the entity well enough to design it.

3.2 File Name Decorators

Rule:
Append `_tb` to test bench files; Append `_pkg` to package files.

Reasoning:
This keeps associated files together when sorted.

3.3 File Structure

Rule(s):
Create a separate file for the entity declaration as in Listing 3.1 and Listing 3.2.

Reasoning:
A separate entity file facilitates reuse and multiple architectures. Furthermore, it makes importing the entity into documentation easier.

Rule(s):
In the entity declaration, keep related ports together.

Reasoning:
Keeping related ports together increases readability.

Rule(s):
Limit each file to one architecture.

Reasoning:
One architecture per file increases readability and makes reuse easier.

An exception to these rules may be made for code managed in a design entry tool, provided that the code can be exported as readable text in accordance with this rule.

Listing 3.1: Example entity declaration

```
1  entity entity_name is
2  generic(
3    TCO : time -- Register time from clock-to-output; for functional simulation only
4  );
5  port(
6    -- Interface 1
7    port_a_in  : in  std_logic; -- Keep related ports together
8    port_b_out : out std_logic;
9    -- Interface 2
10   port_c_in  : in  std_logic; -- Keep related ports together
11   port_d_out : out std_logic;
12   -- System Interface
13   clk  : in std_logic; -- Always place clk and reset last
14   rst  : in std_logic  -- Synchronous reset
15 );
16 end entity_name;
```

Listing 3.2: Example architecture

```
1  architecture entity_name_arch of entity_name is
2    -- Procedures and Functions
3    -- Constants
4    -- Components
5    -- Signals
6  begin
7    -- Processes and combinatorial logic
8  end entity_name_arch;
```

Rule(s):
Assign constants for register addresses and place them in a VHDL package(see Listing 3.3). Place state type definitions in a package[a].

Reasoning:
The package allows other entities, such as a test bench, access to the constants and type definitions.

[a] Placing signals in a package can cause them to be *visible in any file that uses the package*; a dangerous practice.

Listing 3.3: Example VHDL package

```
1  library IEEE ;
2    use IEEE.std_logic_1164.all ;
3
4  package my_design_pkg is
5
6    constant ADR_TIME_STAMP_0     : std_logic_vector( 7 downto 0 ) := X"0C" ;
7    constant ADR_TIME_STAMP_1     : std_logic_vector( 7 downto 0 ) := X"0D" ;
```

```
 8   constant ADR_TIME_STAMP_2     : std_logic_vector( 7 downto 0 ) := X"0E" ;
 9   constant ADR_TIME_STAMP_3     : std_logic_vector( 7 downto 0 ) := X"0F" ;
10
11   type state_type is
12   (
13     idle ,
14     addr ,
15     control
16   ) ;
17
18 end package my_design_pkg ;
```

3.4 Naming Conventions

Table 3.1: Naming conventions

Item	Description
case	Do not use CamelCase. Separate words in names with underscores.
generic, constant	All capitals.
port	Append `_in`, `_out`, or `_inout`.
function	Append `_fcn`.
procedure	Append `_proc`.
type	Append `_type`.
boolean signal	Prepend `is_`, as in `is_ready`.
active low signal	Append `_n`, as in `src_rdy_n_out`.
process	Prepend `p_` to the signal name whose behavior the process defines as in `p_signal : process ( clk )`. For processes that define multiple outputs, choose the primary output or the overall function of the process as in `p_fsm_output: process ( clk )`
component	Prepend `inst_` to the component name as in `inst_the_component`.
inter-module signal names	Prepend the instance name of the source component as in `<source module name>_<signal name>`. As an example, consider the name of the signal `src_rdy_n_out` from `inst_frame_generator` component. The signal is then named `frame_generator_src_rdy_n_out`.

3.5 Port Types

Rule:
Use port types `in` and `out`. Avoid `inout` except at the top level entity.

Reasoning:
Using `data_in`, `data_out`, and `data_t` for components facilitates reuse. This also confines tristate devices to the top level entity.

3.6 Generics

Rule:
Generics shall be all capitals.

Reasoning:
This differentiates generics from signals.

Rule:
Generics shall not have default values except at the top level of the design[a].

Reasoning:
This requires the designer to set generic values at instantiation. This avoids instantiation errors related to assuming generic values.

[a] In rare cases it may be prudent to include a default value for a generic.

Rule:
Include a clock to output (`TCO`) generic in all entities for use in simulation. Include a propagation delay (`TPD`) generic when needed.

Reasoning:
This ensures proper timing relationships during simulation.

3.7 Signed and Unsigned Libraries

Rule:
Do not use the IEEE `std_logic_arith`, `signed`, or `unsigned` libraries. Use `IEEE.numeric_std.all` and declare signals signed or unsigned.

Reasoning:
Avoiding these libraries forces the designer to keep track of the "signedness" of signals[a]. This is illustrated in Listing 3.4 and Figure 3.1.

[a] Even in a test bench it is important for the designer to keep track of the types of signals used.

Listing 3.4: Type conversion

```
1  typical_in  : in  std_logic_vector( 3 downto 0 );
2  typical_out : out std_logic_vector( 3 downto 0 );
3        :
4  signal ram_addr_int : natural range 0 TO RAM_BLOCK_SIZE-1;
5  signal ram_addr_slv : std_logic_vector(6 downto 0);
6  signal some_sig     : signed( 3 downto 0 );
7        :
8  some_sig <= signed( typical_in );
9  typical_out <= std_logic_vector( some_sig );
10
11 ram_addr_slv <= std_logic_vector( TO_UNSIGNED( ram_addr_int, ram_addr_slv'length ));
12 ram_addr_int <= TO_INTEGER( unsigned(ram_addr_slv));
```

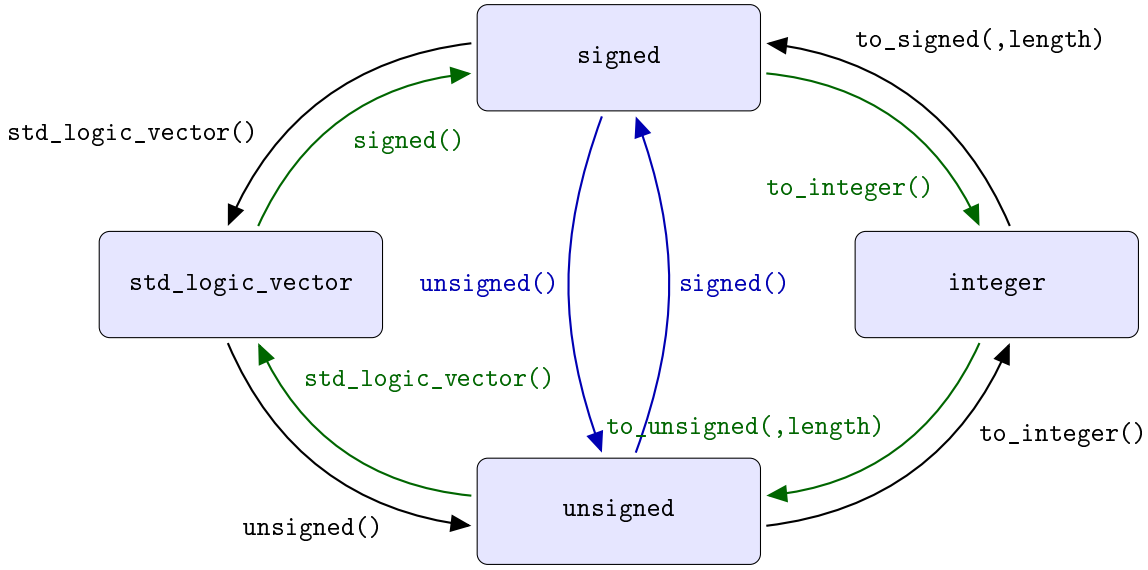

Figure 3.1: VHDL Type Conversions

3.8 Vector Types

Rule:
All signals and variables that are used in arithmetic statements should be declared as `signed`, `unsigned` or an `integer` with a defined range. Use `std_logic_vector`[a] only for non-numeric signals and for interfacing with vendor libraries and with the outside world[b].

[a] Do not use `bit` or `bit_vector`.
[b] Type `boolean` is acceptable.

Reasoning:
This forces the designer to remain cognizant of the data types and ranges.

Rule:
Use `Y downto X` where possible. When not possible, create a wrapper around the `X to Y` core to maintain `Y downto X` vectors within all other new code[a].

[a] X to Y is acceptable when needed based on engineering judgment.

Reasoning:
Standardizing on `X downto Y` reduces the probability of positional bit errors.

3.9 VHDL Version

Rule:
In general, VHDL should be written to comply with IEEE 1076-1993. Where necessary, newer versions are permitted. Note the VHDL standard used in the file header.

Reasoning:
Using unproven tools is generally unwise. Use VHDL 93 when it is good enough to do the job at hand[a].

[a] Updating or modifying old designs may be accomplished using the VHDL version of the original design.

3.10 Indentation

Rule:
Indent 2 spaces. Do not use tabs[a].

[a]It is seldom prudent to spend engineering labor hours changing a file from 3 space indentation to 2 space indentation.

Reasoning:
This improves readability. Documents with tabs may "line up" in one tool, but look "ragged" in another. Tabs can be changed to spaces by tools automatically.

3.11 Line Widths

Rule:
The width of all lines should not exceed 80 characters[a].

[a]Lines may exceed 80 characters when it improves the readability of code or comments.

Reasoning:
Code reviews and other documentation require HDL readability. Lines longer than 80 characters cause wrap and decrease readability.

3.12 Operators

Rule:
Include a space before and after all operators.

Reasoning:
This improves readability. Clustering multiple mathematical or logical symbols together adds *perceived* complexity.

3.13 Reset and Clear

Rule:
Prefix asynchronous reset and clear signals to make it obvious that they are asynchronous[a].

[a]Designers should ascertain the reset or clear is asynchronous to the clock in question. A signal can be asynchronous to one clock but not another.

Reasoning:
This prevents confusion at the component level.

3.14 Parentheses

Rule:
Use parentheses to ensure proper evaluation of all mathematical expressions. Include a space between the parentheses and its contents.

Reasoning:
Do not rely on VHDL operator precedence rules. Improves readability.

3.15 Type Casting

Rule:
Each cast should include a comment describing how the code ensures proper behavior.

Reasoning:
Casting a signal can cause significant confusion. A comment will minimize the opportunity for error.

Listing 3.5: Type cast

```
1 -- cast example
2 some_slv <= std_logic_vector( to_unsigned( an_integer, some_slv'length ) );
```

3.16 process Sensitivity List

Rule:
Use processes for clocked logic elements and concurrent statements for combinatorial logic.

Reasoning:
This avoids sensitivity list errors[a]. If the `process` construct is reserved for clocked logic, the sensitivity list should contain only `clock` and maybe an asynchronous `reset`.

[a] VHDL 2008 keyword `all` notwithstanding.

3.17 if Statements

Rule:
All `if` statements in non-clocked processes (combinatorial circuits) should include an `else` clause.

Reasoning:
This avoids inferred latch errors.

3.18 case Statements

Rule:
All `case` statements in non-clocked processes should include a `when others` clause[a].

[a] It may be acceptable to use default values instead of `when others`.

Reasoning:
This *theoretically* ensures all cases are covered[a].

[a] If the `when others` clause covers a large number of cases it may not be synthesizable. For instance, a 12 state one-hot FSM has $2^{12} - 12 = 4084$ unused combinations; likely an insurmountable problem for timing closure.

3.19 Commented HDL

Rule:
No block of disabled code should remain in the source code of a released product.

Reasoning:
Commented out code adds confusion and the risk of unexpected code being compiled or synthesized.

3.20 Comment Content

Rule:
All comments should add value[a] and be written in clear and complete sentences with proper spelling, grammar, and punctuation.

[a] Avoid explaining the obvious.

Reasoning:
Well written, understandable comments increase the understanding of the function of the code[a].

[a] ...and the reputation of the engineer.

3.21 File Markings

Rule:
Include a header, such as shown in Listing 3.6, at the beginning of every file that includes, as a minimum: file name, HDL standard, target devices, description, dependencies[a], and revision[b]. Tool versions should be tracked in the build instruction section of the design document.

[a] List other files required.
[b] We highly recommend the use of a revision control tool.

Reasoning:
A standard file header provides information regarding the origin of the file and any unique characteristics.

Listing 3.6: File header

```
1  --------------------------------------------------------------------------------
2  -- Company:         company_name
3  -- Engineer:        engineer_name
4  -- File Name:       file_name.vhd
5  -- HDL Standard:    VHDL87 / VHDL93 / VHDL2002 / VHDL2008
6  -- Target Devices: Manufacturer and device number or "Not target specific"
7  --
8  -- Description:
9  --   Write a brief, high level description of the function of this entity.
10 --
11 -- Dependencies:
12 --   List all subordinate files. Include entities, packages, etc.
13 --
14 -- Revision: (date initials description)
15 --   15-Dec-2008 cf  Created
16 --    2-Feb-2009 cf  Peer reviewed with <Name>.
17 --    4-Feb-2009 cf  Completed simulation.
18 --
19 -- Additional Comments:
20 --
21 --------------------------------------------------------------------------------
```

Rule:
When required, include a copyright notice[a] such as shown in Listing 3.7.

[a] When not required, the copyright statement must not be included.

Reasoning:
Legal, government, or company requirement.

Listing 3.7: Copyright Notice

```
1  --------------------------------------------------------------------------------
2  -- Copyright (C) <year> <company name>, all rights reserved
3  --------------------------------------------------------------------------------
```

Rule:
When required, include the following proprietary marking[a].

[a]When not required, the proprietary marking must not be included.

Reasoning:
Legal, government, or company requirement.

Listing 3.8: Proprietary Marking

```
1 ------------------------------------------------------------------------------
2 -- This document contains proprietary information of <COMPANY NAME> and is to
3 -- be used only for the purpose for which it has been supplied. This document
4 -- is not to be duplicated in whole or in part without prior written
5 -- permission from a duly authorized representative of <COMPANY_NAME>.
6 ------------------------------------------------------------------------------
```

Rule:
When required, include the following export control warning[a].

[a]When not required, the export control warning must not be included.

Reasoning:
This may be a legal, government, or company requirement.

Listing 3.9: Export Control Warning

```
1 ------------------------------------------------------------------------------
2 --                          EXPORT CONTROL WARNING
3 -- Warning: Information Subject to Export Control Laws This document contains
4 -- technical data whose export is restricted by the Arms Export Control Act
5 -- (Title 22, U.S.C. 2751 et seq) or Executive Order 12470. Violation of these
6 -- export control laws is subject to severe criminal penalties. Dissemination
7 -- of this document is controlled under DoD Directive 5230.25.
8 ------------------------------------------------------------------------------
```

3.22 Finite State Machines (FSMs)

Rule:
Limit FSMs to no more than 15 *complex states*[a].

[a]This does not include simple states that have trivial transition equations

Reasoning:
A large number of states quickly becomes difficult to *completely* understand; even for the designer[a].

[a]If you require a large number of states, either divide the FSM into multiple FSM or use a different design technique.

Rule:
Limit the number of entry paths to a single FSMs state.

Reasoning:
State entry paths contribute to the complexity of the state equations. As the number of entry paths increases, so does the logic required to implement it.

Rule:
Always draw a state transition diagram such as the example shown in Figure 3.2.

Reasoning:
The designer's understanding of the FSM is paramount to correct FSM operation. The diagram will greatly assist maintenance. Use any documentation method to increase understanding.

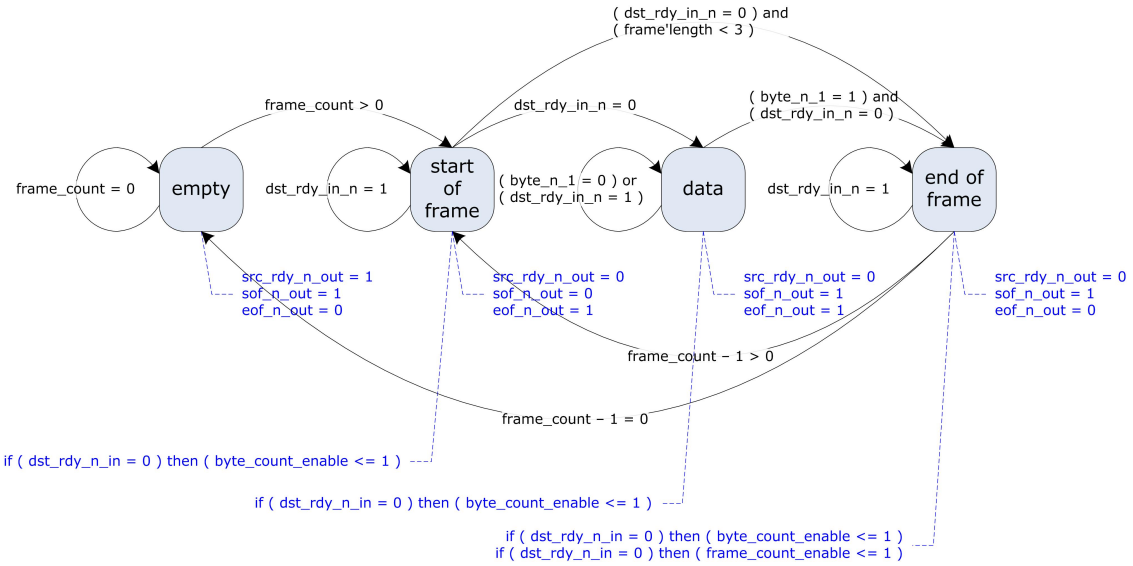

Figure 3.2: FSM State Transition Diagram

Rule:
State machines should have no more than 20 complex states[a].

Reasoning:
Large numbers of complex states lead to errors. Split large state machines into multiple smaller state machines or use a different design technique.

[a] Beware of `when others`. A 12 state one-hot FSM has 4084 unused states. The synthesis tool may ignore a `when others` clause.

Rule:
Include separate sections for the state register (Listing 3.10), Mealy output generation (dependent on state only, Listing 3.12) and Moore output generation (dependent on state and inputs, Listing 3.11).

Reasoning:
Separating the functions significantly lowers complexity and improves readability.

Listing 3.10: FSM state register and next state logic

```
1   ------------------------------------------------------------------------
2   -- state register
3   ------------------------------------------------------------------------
4   p_state : process( sysclk )
5   begin
6
7     if rising_edge( sysclk ) then
```

```
8       if ( reset = '1' ) then -- synchronous reset
9         state <= S0_empty;
10      else
11
12        case state is
13          when empty =>
14            if ( frame_count > 0 ) then
15              state <= S1_sof;
16            else
17              state <= S0_empty;
18            end if;
19
20          when sof =>
21            if ( dst_rdy_in_n = '0' ) then
22              state <= S2_data;
23            else
24              state <= S1_sof;
25            end if;
26
27          when data =>
28            if ( byte_N_1 = '1' ) and ( dst_rdy_in_n = '0' ) then
29              state <= S3_eof;
30            else
31              state <= S2_data;
32            end if;
33
34          when eof =>
35            if ( dst_rdy_in_n = '0' ) then
36              if ( frame_count-1 > 0 ) then
37                state <= sof;
38              else
39                state <= empty;
40              end if;
41            else
42              state <= eof;
43            end if;
44
45          when others =>
46            state <= empty;
47
48      end case;
49    end if;
50  end process;
```

Rule:
Output sections should include `when others` or default values.

Reasoning:
This minimizes inferred latch errors[a].

[a] An inferred latch warning is the tool telling you that the designer did not effectively communicate their intention. To complete the equation for the described behavior, the tool had to *add a memory element the designer did not request!*

Listing 3.11: FSM Moore Outputs

```
1   -------------------------------------------------------------------------
2   -- Moore output logic (outputs that are dependent on state only)
3   -------------------------------------------------------------------------
4   p_moore_outputs : process ( state_reg )
5   begin
6
7     -- default values
```

```
 8     src_rdy_out_n       <= '0';
 9     sof_out_n           <= '1';
10     eof_out_n           <= '1';
11
12     case state_reg is
13
14       when empty =>
15         src_rdy_out_n   <= '1';
16         eof_out_n       <= '0';
17
18       when sof =>
19         sof_out_n       <= '0';
20
21       when eof =>
22         eof_out_n       <= '0';
23
24     end case;
25   end process;
```

Listing 3.12: FSM Mealy Outputs

```
1 -------------------------------------------------------------------------------
2 -- Mealy output logic (outputs that are dependent on state and inputs)
3 -------------------------------------------------------------------------------
4 shift_en_out <= '1' after Tpd when ( state_reg = idle and msg_rdy = '1' ) else
5                 '0';
```

A Definition of Acronyms

CLB	Configurable Logic Block
CPLD	Complex programmable logic device
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
VHDL	VHSIC Hardware Description Language

References

[1] P.J. Ashenden. *The Designer's Guide to VHDL.* Morgan Kaufmann Pub, 3rd edition, 2008.

[2] Michael Barr. *Embedded C Coding Standard.* CreateSpace, 1st edition, 2008.

[3] P.P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability.* Wiley-Interscience, 2006.

[4] Bob Zeidman. Introduction to CPLD and FPGA Design. EE Times Education and Training, 2006.