IEEE 1076.6-200X: VHDL Synthesis Coding Styles for the Future

Jim Lewis, SynthWorks Design Inc. Jim@SynthWorks.com

Abstract

The current standard for VHDL Register Transfer Level (RTL) Synthesis, IEEE standard 1076.6, is based on a finite set of templates. The templates supported by 1076.6 are based on a limited set of defacto practices that were in use at the time the standard was written. However, VHDL is much more powerful and allows for greater flexibility in terms of modeling.

Currently, enhancements for IEEE 1076.6-200X are in the final stages of standardization. Rather than adding another finite set of limited templates, this effort generalized the number templates. This made it possible to subsume the previous templates and include a wide variety of new coding styles. While the breadth of the standard does permit some unusual coding styles, the intent is to make sure that we did not leave out any good coding styles.

The intent of this paper is to illustrate some of the good coding styles that have resulted from this effort and explain how a designer will benefit from these coding styles.

1. Introduction

IEEE 1076.6 standard is important to designers as it lets you know what coding styles are supported by compliant synthesis tools. It also gives a designer a single item to ask for when asking an EDA vendor to create tools that support vendor independent coding styles.

This paper focuses on a designers needs and presents coding style supported by the enhanced standard.

2. Single Edged Registers

The keystone to the register enhancements is a set of rules that govern the creation of a register. The following is a simplified version of the rules. A signal or variable that reads a value that was written on a previous edge of the same clock, has clock and any asynchronous control signals on the sensitivity list, and never has a synchronous assignment override an asynchronous assignment will create a register. These enhanced rules permit processes, procedures, and concurrent assignments to be used to create registers.

The enhanced standard permits a wide variety of coding styles for registers. This allows designers to move away from an arbitrarily restrictive set of limited templates. It also allows the choices of register coding styles to be based on what is most effective to implement the design and what is most effective in simulation.

2.1. IF and Registers

The enhanced standard allows conditions that are synchronous to clock to be included in the condition with the clock edge specification. This is shown by the following example of a register with load enable.

```
LoadEnRegProc : process (Clk)
begin
 if LoadEn='1' and rising_edge(Clk) then
   Q <= D ;
 end if;
end process ; -- LoadEnRegProc
```

Going further, assignments can be made with separate code blocks provided that asynchronous updates to signals always have precedence over synchronous updates to signals. The following process shows reset being coded separately from clock. The advantage of this coding style is that it allows reset to be specified for some signals without having any logic implication for signals that do not require reset. This is shown in the following example where Q1 has asynchronous reset and Q2 does not.

```
TwoReg2Proc : process( Clk, nReset)
begin
  if rising_edge(Clk) then
    Q1 <= D1 ;
    Q2 <= D2 ;
  end if ;
  if  nReset = '0' then    -- no reset on Q2
    Q1 <= '0' ;
  end if ;
end process ; -- TwoReg2Proc
```

Going further, the clock statement can be divided into multiple pieces.

As long as the previous rules are followed, combinational logic, latches, and registers may be mixed in the same process. Note that from a simulation efficiency point of view, this is not recommended.

2.2. Wait and Registers

The enhanced standard permits any form of wait that clearly expresses a clock edge to be used to model a register, provided that the wait statement is either first or last in the process. The following shows a recommended way to code a register with load enable.

```
LoadEnReg1Proc : process
begin
  wait on Clk until LoadEn='1' and Clk='1' ;
  Q <= D ;
end process ; -- LoadEnReg1Proc
```

From a synthesis perspective, the following wait statements are equivalent to the above wait statement. Note, in simulation one may be more efficient than another.

```
wait on Clk until LoadEn='1' and rising_edge(Clk) ;
wait until LoadEn='1' and rising_edge(Clk)  ;
wait until LoadEn='1' and Clk='1' and Clk='event ;
```

Going further, the standard permits any form of a wait statement to be used as a sensitivity list provided that there is additional code that identifies the clock edge. For example, a register with asynchronous reset can be modeled as follows:

```
AsyncResetProc : process
begin
  wait until nReset='1' or Clk='1' ;
  if nReset='1'  then
    Q <= '0' ;
  elsif rising_edge(Clk) then
    Q <= D ;
  end if ;
end process ; -- AsyncResetProc
```

2.3. Concurrent Assignments and Registers

Conditional signal assignment is now permitted to be used in its full VHDL-93 form (without the else). As a result both registers and latches can be modeled. The following creates a register.

```
Q <= D when rising_edge(Clk) ;
```

The following creates a register with asynchronous reset and load enable:

```
Q <=
   '0'  when nReset = '0' else
   D   when LoadEn = '1' and rising_edge(Clk) ;
```

From a simulation perspective, the above coding styles may execute slower than a process with a proper sensitivity list (D and LoadEn not on sensitivity list). Note that with time it is expected that simulators will be able to optimize the code appropriately.

2.4. Subprograms and Registers

The following example shows a procedure created in a package that can be used to create a register.

```
package RegPkg is
  procedure DFF(
     signal Clk      : in   Std_Logic ;
              D        : in   Std_Logic ;
     signal Q         : out Std_Logic
   ) ;
   procedure DFFR( . . .) ;
   procedure DFFLE( . . .) ;
   . . .
end package RegPkg ;

package body RegPkg is
    procedure DFF(
     signal Clk      : in   Std_Logic ;
              D        : in   Std_Logic ;
     signal Q         : out Std_Logic
    ) is
    begin
     if rising_edge(Clk) then
       Q <= D;
     end if;
    end DFF;
   . . .
end package RegPkg ;
```

Registers get created when the subprogram is called. The following example creates three registers in a sequence.

```
  DFFR(Clk, nReset, D,       Reg1) ;
  DFFR(Clk, nReset, Reg1,  Reg2) ;
  DFFR(Clk, nReset, Reg3,  Q) ;
```

By referencing a different package, it would be possible to switch from an asynchronous reset register (favorable for an FPGA implementation) to a synchronous reset register (favorable for an ASIC implementation).

3. Latches

For latches the usage of concurrent assignments and procedures is now supported.

3.1. Concurrent Assignments and Latches

Latches can be created with either conditional signal assignment or selected signal assignment as shown below.

```
Q <= D when Clk = '1' ;
```

```
with Gate select
 Q2 <= D2 when '1',
      unaffected  when others ;
```

3.2. Procedures and Latches

The following example shows a procedure created in a package that can be used to create a latch.

```
begin Package LatchPkg is
procedure latch(
   ENABLE, D :in std_logic;
   signal Q :out std_logic ) ;
end package ;

package body LatchPkg is
 procedure latch(
   ENABLE, D :in std_logic;
   signal Q :out std_logic ) is

  if ENABLE ='1' then
    Q <= D;
  end if;
 end ;
end package ;
```

The following code creates a latch using the procedure in LatchPkg.

```
   latch(Sel, A, Y);
```

3.3. Problematic Latch Coding Style

The following code is potentially problematic. Should it be implemented as a latch or combinational logic (multiplexer with feedback)?

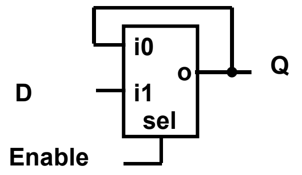
```
ALat : process ( ENABLE, D, Q)
begin
 if ENABLE = '1' then
   Q <= D;
 else
   Q <= Q ;
 end if;
end process;  -- ALat
```

In the extended standard, identity assignments, such as "Q <= Q ;" are deleted by default before considering the process. As a result, by default, Q results in a latch.

This behavior will be overridden when the attribute combinational is set to true for the process as shown below.

```
attribute COMBINATIONAL of ALat: process is TRUE;
```

In this case, the process will create a combinational logic with feedback. A potential implementation is shown below.



4. Combinational Logic

Few restrictions. Complete sensitivity list. Write before Read. Usage of Z implies tristate. Assignment of a metavalue UXW- implies don't care.

4.1. Read Before Write, Signal

When a signal is read before it is written and a clock condition is not present, combinational logic will be produced. This is shown in the example below. Furthermore since all signals read in the process are required to be on the sensitivity list, read before write of a signal when a clock edge is not present will either produce combinational logic or it will produce an error.

```
CombProc : process (A, B, C, D)
begin
 Y <= C or D ;
 C <= A and B ;
end process ;
```

Note, the code is inefficient in simulation and it is recommended that one avoid this type of code.

4.2. Read Before Write, Variable

It is recommend to avoid code that reads a variable before writing it when a clock edge is not present. The synthesis tool is free to either produce an error or generate any logic that will match the simulation behavior of this code. In general, this code is a logic error and it is recommended that synthesis tool vendors produce an error when encountering it.

```
StrangeProc : process (A, B, D)
 variable C : std_logic ;
begin
 Y <= C or D ;
 C := A and B ;
end process ;
```

5. Modeling ROM and RAM

5.1. RAM

RAM may be modeled for any register or latch. Typically the RAM will be based on an array of integer, std_logic, or std_logic_vector. A RAM is required to be created when the “ram_block” attribute is applied to a signal or variable. The example below shows a level sensitive type of RAM.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.rtl_attributes.all;
entity ramlatch is
  generic (
    WIDTH : Natural := 8;
    DEPTH : Natural := 16);
  port (
    a : in std_logic_vector(DEPTH-1 downto 0) ;
    we : in std_logic ;
    d : in std_logic_vector(WIDTH-1 downto 0) ;
    q : out std_logic_vector(WIDTH-1 downto 0)
  );
end entity ramlatch;
architecture RTL of ramlatch is
   type ram_typ is array(0 to 2**DEPTH – 1) of
      std_logic_vector(WIDTH-1 downto 0);
   signal ram : ram_typ;
   attribute ram_block of ram : signal is "";
begin
   Ram_Proc: process (a, d, we) is
   begin
      if we = '1' then
         ram(to_integer(unsigned(a)))  <= d;  -- write RAM
      end if;
    end process Ram_Proc;
    q <= ram(to_integer(unsigned(a)))  ;  -- read RAM
end architecture RTL;
```

Either the writing to the RAM or reading from the RAM can be register based by putting the appropriate piece of code in a clocked process. A non-null value for the ram_block attribute will cause a specific library cell to be picked as a candidate for implementation.

Use of the attribute, logic_block, as shown below will require the model to be implemented with either latches or registers (based on the code).

```
attribute logic_block of ram : signal is "TRUE";
```

5.2. ROM

A ROM may be modeled by either assigning a constant value in a case statement or by looking up values in a constant array. A ROM is required to be created when the “rom_block” attribute is applied to a signal or variable. The example below shows a ROM with case statement.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.rtl_attributes.all;
entity ROM is
  port (
    Z : out  std_logic_vector(3 downto 0);
    A : in    std_logic_vector(2 downto 0)
  );
end entity ROM;
architecture RTL of ROM is
   attribute rom_block of Z : signal is "ROM32Kx16";
begin
  Rom_Proc : process (A) is
  begin
     case A is
       when "000" => Z <= "1011";
       when "001" => Z <= "0001";
       when "100" => Z <= "0011";
       when "110" => Z <= "0010";
       when "111" => Z <= "1110";
       when others => Z <= "0000";
     end case;
  end process Rom_Proc;
end architecture RTL;
```

The same ROM implemented with a constant array.

```
architecture RTL2 of ROM is
   type mem_typ is array(0 to 7) of
      std_logic_vector(3 downto 0);
   constant ROMINIT : mem_typ :=
    ( 0 => "1011", 1 => "0001", 2 => "0011",
      3 => "0010", 4 => "1110", others => "0000");
   attribute rom_block of ROMINIT : constant is
        "ROM_CELL_XYZ01";
begin
   Z <= ROMINIT(to_integer(unsigned(A)));
end architecture RTL2;
```

Use of the attribute logic_block will require the model to be implemented as discrete random logic rather than a ROM.

6. Multiple Edged Registers

6.1. Using IF and Dual-Edged flip-flops

The extended standard supports flip-flops with multiple edges. When multiple edges are specified for flip-flops, the priority relationships of the clocks are ignored by synthesis.

```
DualEdgeFF : process( nReset, Clk1, Clk2)
begin
  if rising_edge(Clk1) and nReset = '1' then
    Q <= D ;  -- Functional Data
  elsif rising_edge(Clk2) and nReset = '1' then
    Q <= SD ;  -- Scan Data
  elsif (nReset = '0') then
    Q <= '0' ;
  end if ;

  -- RTL_SYNTHESIS OFF
  if rising_edge(Clk1) and rising_edge(Clk2) then
    report "Warning: . . ." severity warning ;
    Q <= 'X' ;
  end if ;
  -- RTL_SYNTHESIS ON
end process;
```

The meta-comments, "-- RTL_SYNTHESIS OFF" and "-- RTL_SYNTHESIS ON" cause the synthesis tool to ignore the code between them. This code can be used to validate that the assumptions that were made for synthesis are valid. In this case the code makes sure both clocks do not change at the same time. If the RTL code is written this way (and they work), RTL simulations will compare with Gate level simulations.

The Dual-Edge coding concept can be used to handle rising and falling edges of the same clock as shown below:

```
DualEdge_Proc: process (Clk, Reset) is
begin -- process DualEdge_Proc
  if Reset = '1' then
    Q <= (others => '0');
  elsif rising_edge(Clk) then
    Q <= D4Rise;
  elsif falling_edge(Clk) then
    Q <= D4Fall;
  end if;
end process DualEdge_Proc;
```

6.2. Implicit Finite Statemachines

Implicit statemachines use multiple clock specifications (in the form of wait statements) in a single process to model statemachines. The state-register is not explicitly identified. This modeling permits the description of a statemachine at the protocol or algorithmic level. The following code creates a multiplier using a shift and add algorithm.

```
MultProc : process
begin
  wait until clk = '1';
  if start = '1' then
    done <= '0';
    intY <= (others => '0');

    for i in A'range loop
      wait until clk = '1';
      if A(i) = '1' then      -- compute state 1
        intY <= (intY(6 downto 0) & '0') + B ;
      else
        intY <= (intY(6 downto 0) & '0') ;
      end if;
    end loop;

    done  <= '1';
  end if;
end process;
```

7. Creating Safe Statemachines

A safe statemachine is one that recovers from unused states the implementation. Consider the following statemachine. There are five values in the enumerated type. In the process, NextStateProc, each value in the enumerated type has a corresponding case target. In a simulation, the others statement is never executed. As a result, most synthesis tools do not consider the others statement when creating an implementation.

Setting the FSM_COMPLETE attribute to TRUE requires a synthesis tool to use the mappings specified in the others statement to specify a value for states in the implementation that do not have a value specified in any other way. When FSM_COMPLETE is true, it is an error if the statemachine has any unreachable states.

The FSM_STATE attribute allows either an encoding type or state values to be assigned. For state encodings, there are BINARY, GRAY, ONE_HOT, ONE_COLD, or AUTO. By specifying state values, any encoding can be created. The encoding in the example specifies each state to have a hamming distance of two. With this encoding, if noise or an SEU flip a bit in the state vector, the state vector will end up in an invalid state and will end up going to the recovery state specified in the others clause.

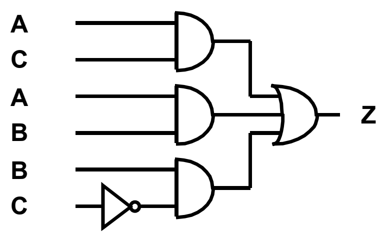
Specification of state values can be problematic if a state is reduced. With this insight, it would have been preferable to have an encoding type to create a binary encoded statemachine with a hamming distance of two (such as HAMMING2) and a one to create a hamming distance of three (such as HAMMING3). It is likely that this will be part of in a future version of 1076.6.

```
type StateType is (S0, S1, S2, S3, S4);
signal state, next: StateType;
attribute FSM_STATE of state : signal is
  "0000 0011 0110 1100 1001" ;
attribute FSM_COMPLETE of state : signal is TRUE;
. . .
NextStateProc : process
begin
  wait until Clk = '1' ;
  if nReset = '0' then
    state <= S0
  else
    case state is
      when S0 =>          state <= S1;
      when S1 =>          state <= S2;
      when S2 =>          state <= S3;
      when S3 =>          state <= S4;
      when S4 =>          state <= S0;
      when others =>      state <= S0;
    end case;
  end if  ;
end process;
```

8. Controlling Logic Terms

The job of a synthesis tool is to optimize logic. Sometimes logic gets removed that needed in the final circuit. The hierarchy attributes give a designer the ability to control some of this by partitioning a circuit into separate pieces.

For example, in the following diagram, the “A and B” term is redundant and would be optimized away. This section will demonstrate how to use attributes to prevent this behavior. Using the techniques shown here, it is possible to constrain the implementation to a limited set of choices.



8.1. Using KEEP

When the keep attribute is applied to a signal, it directs the synthesis to preserve this term through synthesis. This is effectively identical to inserting a non-movable buffer on the net. To preserve the AB term in the circuit above, create a signal for each output of the “AND” gates and apply the attribute KEEP to each signal.

```
signal AC, AB, BC : signal ;
attribute KEEP of AC, AB, BC : signal is TRUE;
. . .
AC <= A and C ;
AB <= A and B ;
BC <= B and not C ;

Z <=  AC or AB or BC ;
```

8.2. Using CREATE_HIERARCHY

When the CREATE_HIERARCHY attribute is applied to a block, it indicates that the boundary around the attributed object is to be maintained. This means that synthesis is not permitted to optimize terms across this boundary. To preserve the AB term in the circuit shown previously, create a signal for each output of the “AND” gates, enclose the logic that creates the “AND” gates in a block statement, and apply the attribute CREATE_HIERARCHY attribute to the block statement label.

```
signal AC, AB, BC : signal ;
attribute CREATE_HIERARCHY of InterBlk : label is
TRUE ;
. . .
InterBlk : block
begin
  AC <= A and C ;
  AB <= A and B ;
  BC <= B and not C ;
end block ;

Z <=  AC or AB or BC ;
```

8.3. Cautions About Attributes

Attributes should be used to convey designer intent of the circuit. If the actions specified by the attributes are only appropriate for a particular implementation of the design, then synthesis tool commands should be used rather than attributes.

9. Sensitivity Lists

In the enhanced standard, processes that create latches and combinational logic are required to be specify all signals read in the process. As a result a synthesis tool is required to produce an error for the following examples.

```
StrangeLatProc : process (A)         -- missing B
begin
  C <= A and B ;
end process ;

CouldBeARegProc: process (Clk)   -- missing D
begin
  if (Clk = '1') then                       -- no clock condition
    Q <= D ;
  end if ;
end process ;
```

For a more detailed explanation, see [LewisSingh02].

10. Subprograms

With the enhanced standard, subprograms can be created to represent any simple functionality. This is an area that can be further exploited by future standards that could create a small macro function library similar to the 74 series of digital board parts.

In synthesis, recursion is supported provided that the procedure can be statically inlined. Variables declared in subporgrams are initialized on each call, and hence, cannot retain a store value between calls of the subprogram.

11. Syntax Enhancements

Syntax enhancements include aliases, configurations, entity instantiation (VHDL-93), conditional signal assignment without an else clause (VHDL-93), and guarded blocks (for latch and register creation). For examples, see [LewisSingh02].

12. Creating Gated Clocks

By setting the GATE_CLK attribute to true, a synthesis tool will transform a register coded as a load enable to a register with a gated clock.

```
attribute GATE_CLK  : boolean;
attribute GATE_CLK of Clk : signal is true;
. . .
LoadEnReg2Proc : process
begin
  wait on Clk until LoadEn='1' and Clk='1' ;
  Q <= D ;
end process ; -- LoadEnReg2Proc
```

13. Vendor Support

Writing a standard is the first step toward getting portability in coding styles and methods for RTL synthesis. The next step is vendor implementation. Supporting a standard is a business decision. To make this an easy decision for the vendors, please be vocal in letting them know that you want them to support the standard IEEE 1076.6-200X. Often the most effective person to pass this information to the vendors is the person who is purchasing your EDA tools. A particularly effective time to do this is when buying new licenses or renewing your current licenses.

14. Supporting Standards

VHDL standards are IEEE standards. As a VHDL community member it is both your right and responsibility to join IEEE committees and participate in VHDL standards. If you don't participate, the changes you envision and wish for (no matter how simple or obvious) will not happen.

To learn more about VHDL standards see the following websites:

EDA Standards:	http://www.eda.org
VHDL-200X:	http://www.eda.org/vhdl-200x
RTL Synthesis:	http://www.eda.org/siwg.

15. Acknowlegements

This paper includes many examples taken from the SIWG reflector. The authors would like to thank the many people who have contributed examples to the SIWG reflector.

16. References

[LewisSingh02] Jim Lewis and Vinaya Singh, "Extensions to the VHDL RTL Synthesis Standard", published in HDLCon 2002.

17. About the Author

Jim Lewis, the founder of SynthWorks, has seventeen years of design, teaching, and problem solving experience. In addition to working as a Principal Trainer for SynthWorks, Mr. Lewis does ASIC and FPGA design, custom model development, and consulting. Mr. Lewis is an active member of IEEE Standards groups including, VHDL (IEEE 1076), RTL Synthesis (IEEE 1076.6), Std_Logic (IEEE 1164), and Numeric_Std (IEEE 1076.3). Mr. Lewis can be reached at jim@SynthWorks.com.

For more information about SynthWorks, see http://www.SynthWorks.com

About SynthWorks VHDL Training