Extensions to the VHDL RTL Synthesis Standard

Jim Lewis
SynthWorks Design Inc.
Jim@SynthWorks.com

Vinaya K. Singh
Cadence Design Systems
vinaya@cadence.com

Abstract

In 1999, IEEE 1076.6, Standard for VHDL Register Transfer Level Synthesis was first standardized. This standard places coding restrictions on model developers and language support requirements on synthesis tool developers.

The goal of this standard is portability of RTL code on different synthesis tools. If a model developer writes code that is compliant to this standard, it will be supported by all compliant synthesis tools. The 1999 revision of this standard set the restrictions on tool vendors low with the intention of getting to portability quickly.

Currently enhancements are being finalized for the next draft of the standard. Syntax enhancements include VHDL-93 support as well aliases and configurations to name a few. Semantic enhancements transition us from a template based semantic methodology to an algorithm based semantic methodology. One of the benefits of this is a richer, more flexible set of register coding styles.

This paper discusses the enhancements and shows how they will help Joe Designer to efficiently write a wide variety of hardware elements at a higher level. We will also be asking for your support to encourage EDA vendors to support VHDL standards.

1. Introduction

This paper is organized into four main sections: semantics, sensitivity lists, syntax, and attributes. The enhancements are shown in a case study format. We show an example and explain how the enhancement will benefit a designer.

Note this paper represents work in progress and as such may change before the standard is finalized

2. Semantic Enhancements

2.1. Load Enable Registers

The current standard requires that clock be the only condition in the clock statement of a register. To use the current standard to code a load enable register, first code the clock statement and then, separately, code the semantics for the load enable. This is shown in the following code segment:

```
LoadEnReg1Proc : process
begin
  wait until Clk = '1' ;
  if (LoadEn = '1') then
     Q <= D ;
  end if ;
end process ; -- LoadEnReg1Proc
```

The extended standard allows conditions in the clock statement of a register. This allows the semantics for clock and load enable to be coded in a single statement as follows:

```
LoadEnReg2Proc : process
begin
  wait on Clk until LoadEn='1' and Clk='1' ;
  Q <= D ;
end process ; -- LoadEnReg2Proc
```

This can result in code that simulates faster. The process only wakes up on the rising edge of Clk where LoadEn (load enable) is active. As a result, the code following the clock statement will be evaluated less often.

The extended standard also allows clock and load enable to be coded together when using a similar "if" style register:

```
LoadEnReg3Proc : process (Clk)
begin
 if LoadEn='1' and Clk='1' and Clk'event then
   Q <= D ;
 end if;
end process ; -- LoadEnReg3Proc
```

2.2. Register with Asynchronous Reset

Implementing asynchronous reset requires reset to have priority over Clk. In the current standard, the only way to code this is using an "if-then-elsif" priority structure as shown below:

```
ResetReg1Proc : process( Clk, nReset)
begin
  if  nReset = '0' then
    Q <= '0' ;
  elsif rising_edge(Clk) then
    Q <= D ;
  end if ;
end process ; -- ResetReg2Proc
```

The extended standard allows an algorithmic approach to implementing hardware. A register with asynchronous reset may be coded as follows with the extended standard:

```
ResetReg2Proc : process( Clk, nReset)
begin
  if rising_edge(Clk) and nReset = '1' then
    Q <= D ;
  elsif nReset = '0' then
    Q <= '0' ;
  end if ;
end process ; -- ResetReg2Proc
```

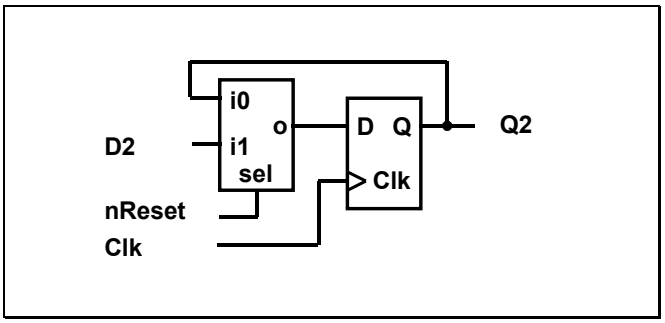
This can result in code that simulates faster. Typically the reset condition is only active at the beginning of a simulation. By coding the most prevalent condition first, less conditions need to be evaluated for most executions of the process.

2.3. Registers with Mixed Reset Conditions

Some designers code all registers in a block in the same process. If some registers require reset and some do not, the following code can result:

```
TwoReg1Proc : process( Clk, nReset)
begin
  if  nReset = '0' then
    Q1 <= '0' ;
  elsif rising_edge(Clk) then
    Q1 <= D1 ;
    Q2 <= D2 ;
  end if ;
end process ; -- TwoReg1Proc
```

This code results in extra logic being created. The extra logic (multiplexer shown in the figure below) maintains the value of Q2 during reset. This is an accurate reflection of the above code since during reset Q2 does not get a new value, even if clock changes.



This extra logic results in a larger, slower design. To create efficient hardware with the current standard one must use two processes.

The algorithmic understanding of semantics of the extended standard gives us more options to implement the hardware. With the extended standard, the priority structure of the asynchronous reset can be created with an "if-then-end if" priority structure as shown below:

```
TwoReg2Proc : process( Clk, nReset)
begin
  if rising_edge(Clk) then
    Q1 <= D1 ;
    Q2 <= D2 ;
  end if ;
  if  nReset = '0' then
    Q1 <= '0' ;
  end if ;
end process ; -- TwoReg2Proc
```

During reset of TwoReg2Proc, Q2 will get updated only by the rising-edge of clock and not by a value of nReset. Hence, no extra logic will be produced.

2.4. Registers and Subprograms

The extended standard permits registers to be modeled inside a subprogram.

```
architecture RTL of  Shift4x1 is  -- 4 Stage Shift Reg
  procedure FF(
    signal Clk      : in   Std_Logic ;
           nReset : in   Std_Logic ;
           D        : in   Std_Logic ;
    signal Q        : out Std_Logic
   ) is
  begin
    if nReset='0' then
      Q <= '0';
    elsif Rising_Edge(Clk) then
      Q <= D;
    end if;
  end FF;
  -- Clk, nReset, D, Q are ports of std_logic
  signal Reg1, Reg2, Reg3 ;
begin
  FF(Clk, nReset, D, Reg1);
  FF(Clk, nReset, Reg1, Reg2);
  FF(Clk, nReset, Reg3, Q);
end RTL ;
```

This feature allows the creation of a package with a generic set of registers in it. A practical use of this would be to switch from asynchronous resets (in an FPGA implementation) to synchronous resets (in an ASIC implementation).

2.5. 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;
```

2.6. Mixing Logic, Latches, and Registers

The algorithmic nature of the extended standard allows mixing of registers, latches, and combinational logic in the same process. To illustrate this, consider the following process:

```
-- Note SPLIT and SIZE are really generics
Constant SPLIT : integer := 3 ;
Constant SIZE   : integer := 8 ;
signal Q1 : std_logic_vector (SIZE – 1 downto 0) ;
-- note that lsb and msb are defined identically to Q1
 . . .
RegPlusLatProc : process(Clk, nReset, D3)
begin
  for i in Q1'range loop
    if reset ='1' then
      if( i <  SPLIT ) then
        Q1(i) <= '0';
      else
        Q1(i) <= '1';
      end if;
    elsif( clk'event and clk='1' ) then
      if( i < SPLIT ) then
        Q1(i) <= lsb (i);
      else
        Q1(i) <= msb (i-SPLIT);
      end if;
    end if;
  end loop;

  if Clk = '1' then
    Q3 <= D3 ;
  end if ;
end process ;
```

We do not necessarily recommend coding this way. In fact, this can be coded in a potentially more simulation efficient manner as follows:

```
AltProc : process(Clk, nReset)
begin
  if (nReset = '0') then
    Q1 <= ((Q1'left - SPLIT) downto 0 => '1') &
            ((SPLIT - 1) downto 0 => '0') ;
  elsif rising_edge(Clk) then
    Q1 <= msb ((Q1'left - SPLIT) downto 0)  &
             lsb((SPLIT - 1) downto 0) ;
  end if ;

Q3 <= D3 when Clk = '1' ;
```

2.7. Implicit Finite Statemachines

Implicit statemachines use multiple clock specifications in a single process to model statemachines. The state-register is not explicitly identified. This modeling helps a designer describe state machine at the protocol or algorithmic level.

```
UartTxFunction : Process
Begin
  TopLoop : loop
    if (nReset = '0') then
      SerialDataOut <= '1' ;
      TxRdyReg <= '1' ;
    end if ;

    -- Wait for data and then send start bit
    wait until nReset = '0'  or
          (rising_edge(UartTxClk) and DataRdy = '1') ;
    next TopLoop when nReset = '0' ;
    SerialDataOut <= '0' ;
   TxRdyReg <= '0' ;

    -- Send 8 Data Bits
    for i in 0 to 7 loop
      wait until nReset = '0' or rising_edge(UartTxClk) ;
      next TopLoop when nReset = '0' ;
      SerialDataOut <= DataReg(i) ;
      TxRdyReg <= '0' ;
    end loop ;

    -- Send Parity Bit
    wait until nReset = '0' or rising_edge(UartTxClk) ;
    next TopLoop when nReset = '0' ;
    SerialDataOut <= DataReg(0) xor DataReg(1) xor
      DataReg(2) xor DataReg(3) xor DataReg(4) xor
      DataReg(5) xor DataReg(6) xor DataReg(7) ;
    TxRdyReg <= '0' ;

    -- Send Stop Bit
    wait until nReset = '0' or rising_edge(UartTxClk) ;
    next TopLoop when nReset = '0' ;
    SerialDataOut <= '1' ;
    TxRdyReg <= '1' ;
  end loop ;
end process ;
```

3. Sensitivity List Enhancements

This section explains the basis for additional rules on sensitivity lists imposed by the extended standard. The goal of these additional rules is to ensure portability of a designer's code.

3.1. Sensitivity Lists & Combinational Logic

Consider the following code. Note that only A is on the sensitivity list. What should this code create?

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

How does this process work? When A changes, C gets updated. When B changes, C maintains its current value until A changes sometime later. This is a circuit that has a storage element and updates on a change of a signal. A detailed analysis shows this creates a register plus combinational logic [Molenkamp].

What do synthesis tools do with this code? Some produce a register plus combinational logic (as the code implies) and some create combinational logic (as the designer most likely intended). We have a problem. The code is not portable.

Synthesis tools that are compliant to the extended standard are required to terminate with an error in this situation. The code is not permitted to combinational logic since the process does not have a complete sensitivity list. The code is not permitted to create a register since there is not an explicit edge condition (such as Clk='1' and Clk'event).

3.2. Sensitivity Lists & Latches

Consider the following code. Note that there is no clk'event. Is this a register or a latch?

```
CouldBeARegProc: process (Clk)
begin
  if (Clk = '1') then
    Q <= D ;
  end if ;
end process ;
```

How does this process work? When Clk changes, the process will execute. If Clk='1' then Q will get updated. Hence, Q will get updated on the rising edge of Clk. This code suggests a register.

What do synthesis tools do with this code? Some produce a register (as the code implies) and some create a latch (ignoring the sensitivity list). Again we have a portability problem.

Synthesis tools that are compliant to the extended standard are required to terminate with an error in this situation. The code is not permitted to create a latch since the process does not have a complete sensitivity list. The code is not permitted to create a register since there is not an explicit edge condition (such as Clk='1' and Clk'event).

3.3. Latches: Bad and Good

Consider the following code. What should be created by process CouldBeALatch1 and CouldBeALatch2?

```
CouldBeALatch1 : process ( ENABLE, D)
begin
  if ENABLE = '1' then
    Q1 <= D;
  else
    Q1 <=Q ;
  end if;
end process;  -- CouldBeALatch1

CouldBeALatch2 : process ( ENABLE, D, Q2)
begin
  if ENABLE = '1' then
    Q2 <= D;
  else
    Q2 <= Q2 ;
  end if;
end process;  -- CouldBeALatch2
```

Either of these could be interpreted to be a latch under some situations. In example, CouldBeALatch1, the process never runs for Q1 changing, hence, the else clause never gets evaluated. In example, CouldBeALatch2, Q2 is on the sensitivity list. When Q2 changes and Enable is '0', Q2 will be updated with itself. This is a zero delay feedback path.

In both cases, the results depend on the synthesis tool. Some tools implement a multiplexer with the output (Q1 or Q2) fed-back to one of the inputs. This is zero delay feedback just as the code implies. Some tools implement a latch, probably as the designer intended.

In the extended standard, CouldBeALatch1 is an error since Q1 is not on the sensitivity list. It is the current belief of the standards group that CouldBeALatch2 is combinational logic with zero delay feed back just as the code implies.

Neither CouldBeALatch1 nor CouldBeALatch2 create a latch as the designer intended. An effective-portable coding style for a latch is:

```
AGoodLatch : process ( ENABLE, D)
begin
  if ENABLE = '1' then
    Q3 <= D;
  end if;
end process;  -- AGoodLatch
```

4. Syntax Enhancements

Syntax enhancements include aliases, configurations, entity instantiation (VHDL-93), conditional signal assignment without an else clause (VHDL-93), and guarded blocks.

4.1. Netlists and Entity Instantiation

The extended standard supports entity instantiation. This should facilitate creation of netlists with a text editor.

```
 U_MuxL  : entity work.Mux8x2
   port map (Sel, A, C, MuxL);

 U_MuxR  : entity work.Mux8x2
   port map (Sel, B, D, MuxR);

 U_Adder : entity work.Adder8
   port map (MuxL, MuxR, Add);
```

4.2. Aliases

The extended standard supports aliases in user code. An alias is a convenient way to give a data object parameter a known direction in a subprogram.

```
alias new_L: unsigned (L'length-1 downto 0) is L;
```

See the package std_logic_1164 for many complete examples. Note that the use of aliases in standard packages is supported by the current standard.

4.3. Latches and Concurrent Statements

The extended standard allows latches to be created with either conditional signal assignment (concurrent if) or selected signal assignment (concurrent case).

```
Q1 <= D1 when Gate = '1' ;  -- VHDL-93 feature

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

4.4. Registers and Concurrent Statements

The extended standard allows registers to be created with conditional signal assignment (concurrent if):

```
Q <= D when rising_edge(Clk) ;  -- VHDL-93 feature
```

Caution: From a language point of view, this is equivalent to the following process:

```
InefficientReg: process(Clk, D)
begin
  if rising_edge(Clk) then
    Q <= D ;
  end if ;
end process ;
```

The code is correct, but inefficient. Having D on the sensitivity list causes the process to run when D changes, but Q will not be updated when D changes (unless Clk changed simultaneously).

4.5. Latches and Guarded Blocks

The extended standard supports creating a latch with a guarded block as shown below:

```
guardedLatch : block  ( enable = '1' )
begin
     latch1 <=  guarded d;
end block ;
```

4.6. Registers and Guarded Blocks

The extended standard supports creating a register with a guarded block as shown below:

```
guardedRegSyncBlock : block ((not clk'stable and clk='1')
and set = '0' and reset = '0' )
begin
    Q <=  guarded D;
end  block  ;

guardedRegAsyncBlock : block  (set = '1' or reset = '1')
begin
    Q <= guarded '0' when reset = '1' else
              '1' when set = '1' else
              'X' ;
end  block ;
```

5. Attribute and Metacomment Additions

Attributes and Metacomments are used as guides put in the code to tell a synthesis tool the intent of the code. As such they provide a starting point for trying to understand and synthesize the code. The intent is to reach the desired result, and to reach it in an expedient manner.

Caution: this section is a small sampling of the work being discussed in committee and is not in any way mature. For current work on the attributes see the VHDL SIWG reflector (http://www.vhdl.org/siwg). Note that it is a good time to contribute ideas or express an opinion (good or bad) about the attributes being considered.

5.1. Attributes for design hierarchy

Hierarchy attributes control creation or destruction of hierarchy for entities, blocks, processes, and subprograms. These features control which RTL pieces of code are optimized together.

Using these features a piece of RTL code can be isolated and synthesized by itself simply by enclosing the code in a VHDL block statement and creating hierarchy.

Pieces of RTL code from separate entities can be optimized together by loading a higher level of the design and removing the hierarchy of the two entities.

5.2. Attributes for hardware implementation

5.2.1. Registers with Set and Reset

Coding a register with both set and reset involves a priority relationship between set and reset:

```
attribute SYNC_SET_RESET  : boolean;
attribute SYNC_SET_RESET of nReset : signal is true;
attribute SYNC_SET_RESET of nSet : signal is true;
. . .
SetResetReg1 : process (Clk, nSet, nReset)
begin
  if (nReset = '0') then
    Q <= '0' ;
  elsif (nSet = '0') then
    Q <= '1' ;
  elsif rising_edge(Clk) then
    Q <= D ;
  end if ;
end process ;
```

The intention of the attribute is to permit the synthesis tool to ignore the priority relationship between set and reset.

5.2.2. One Hot Multiplexers

A one-hot multiplexer is And-Or logic where the select lines are mutually exclusive from each other. When the control signal and the data signal are the same size, the code is as follows:

```
Y <=
   (ASel and A) or (BSel and B) or
   (CSel and C) or  (DSel) and D) ;
```

When the data signal is std_logic_vector and the control signal is std_logic, the situation is more interesting. The following correctly describes a one-hot multiplexer:

```
MuxSel <= ASel & BSel & CSel & DSel ;

OneHotMux_1 : process (MuxSel, A, B, C, D)
begin
  case MuxSel is  -- ieee rtl_synthesis infer one_hot_mux
    when "1000" =>       Y <= A;
    when "0100" =>       Y <= B;
    when "0010" =>       Y <= C;
    when "0001" =>       Y <= D;
    when "0000" =>       Y <= (others => '0');
    when others =>       Y <= (others => 'X');
  end case ;
end process ;
```

This metacomment is intended to give a strong hint to a synthesis tool about the implementation of the code. Note however, the code must correctly describe a one hot multiplexer or an error will result.

5.2.3. Encoded Multiplexers

When a multiplexer is not a power of two, some synthesis tools implement non-optimal logic.

```
EncMux_1 : process (MuxSel, A, B, C)
begin
  case MuxSel is  -- ieee rtl_synthesis infer mux
    when "00" =>       Y <= A;
    when "01" =>       Y <= B;
    when "10" =>       Y <= C;
    when others =>     Y <= (others => 'X');
  end case ;
end process ;
```

5.2.4. Clock Gates

To gate a clock with the current standard requires explicit modeling of the clock gate as shown below:

```
ClkGate <= ClkEnable and Clk ;  -- Clock Gate

GatedClkProc : process
begin
  wait until ClkGate = '1' ;
  Q <= D ;
end process ;
```

The extended standard allows a register with load enable to create a clock gate if the appropriate metacomment or attribute is set.

```
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
```

5.3. Additional Attributes

Other attributes under consideration include attributes that are for mapping operators/statements to specific hardware components, directing resource sharing, and directing subprogram implementation.

6. 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.

How do I find out more about VHDL Standards? Go to the web link: http://www.eda.org.

How do I make a VHDL issue report or feature request? Go the web link: http://www.eda.org/vasg.

How do I participate in IEEE VHDL standards? Join both Design Automation Standards Committee (DASC) and VHDL Analysis and Standardization Committee (VASG). DASC is responsible for all EDA standards within IEEE. VASG is the DASC group responsible for VHDL. By joining these committees, you will keep up on all VHDL and related standards developed by IEEE. Join DASC by going to http://dasc.org and clicking on "Application Form" link. Join VASG by going to http://www.eda.org/vasg [MenchAshenden].

How do I find out more about the VHDL synthesis standard? The 1999 revision of 1076.6 is published and sold by IEEE. Information about on going work can be found at: http://www.vhdl.org/siwg.

Are there other groups working on EDA or VHDL standards? Accellera Designers Forum, the group formed from the merge of VIUF (VHDL International Users Forum) and OVI (Open Verilog International), is sponsoring working groups for developing VHDL, Verilog , system level design, and related standards. To find out more about Accellera Designers Forum see the web link: http://www.eda.org/adf.

Why is my EDA vendor slow in supporting VHDL standards? Some EDA vendors are of the opinion that since their users do not request that standards be supported that they are not interested in the new standard. On the other hand, many users think that EDA vendors should support all VHDL standards relevant to their tools, and that they should not have to request their vendor to support the standards. As a result, we have a stand-off.

How do we get EDA vendors to support standards? To break the stand-off, EDA tool purchasers (perhaps the most influential people) and users need to notify EDA vendors that they want them to support all VHDL standards relevant to their tools.

7. 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.

8. References

[MenchAshenden] Paul Menchini and Peter Ashenden, "Some Personal Thoughts on VHDL 200X", to be published in HDLCon 2002.

[Molenkamp] Egbert Molenkamp and Gerhard E. Mekenkamp, "Processes with 'incomplete' sensitivity lists and their synthesis aspects", Proceedings of VIUF 1997

About SynthWorks VHDL Training

Comprehensive VHDL Introduction 4 Days
http://www.synthworks.com/comprehensive_vhdl_introduction.htm
Engineers learn VHDL Syntax plus basic RTL coding styles and simple procedure-based, transaction testbenches. Our designer focus ensures that your engineers will be productive in a VHDL design environment.

VHDL Coding Styles for Synthesis 4 Days
http://www.synthworks.com/vhdl_rtl_synthesis.htm
Engineers learn RTL (hardware) coding styles that produce better, faster, and smaller logic.

VHDL Testbenches and Verification 3 days
http://www.synthworks.com/vhdl_testbench_verification.htm
Engineers learn how create a transaction-based verification environment based on bus functional models.

For additional courses see: http://www.synthworks.com