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

by

Jim Lewis, SynthWorks

Jim@SynthWorks.com

Author Biography

SynthWorks

Jim Lewis, Director of Training, SynthWorks Design Inc.

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 VHDL Standards groups including, VHDL (IEEE 1076), RTL Synthesis (IEEE 1076.6), Std_Logic (IEEE 1164), and Numeric_Std (IEEE 1076.3).

The author can be reached at jim@synthworks.com or (503) 590-4787

The Problem:

- HDLs = vendor independent simulation, however,
  HDLs /= vendor independent synthesis implementation

IEEE 1076.6 = the VHDL solution

- IEEE 1076.6 specifies:
  - Synthesis coding styles that compliant EDA vendors are required to implement
  - Synthesis coding styles that compliant IP/Model developers must use to achieve portability

- Compliant synthesis tool + Compliant IP (purchased) =
  Vendor independent code = No Synthesis problems

IEEE 1076.6-200X, Goals

- Goals: Simplify and Enhance VHDL synthesis coding capability

- First standardized in 1999, P1076.6-200X brings you:
  - Broader register coding styles
    - Multiple edges, subprograms, concurrent assignments
    - Potentially better simulation efficiency
  - RAM & ROM Models
  - Attributes (Statemachine Control)

- Status:
  - Ballot 1 passed
  - Resolving issues and currently re-balloting

- Website: http://www.eda.org/siwg

Register Enhancements

- Replace limited set of templates with a simplified rule (algorithm) based method.

- Abstracted rules:
  - A signal or variable stores a value under the direct control of a clock edge,
  - Asynchronous conditions always have priority over synchronous conditions (matches hardware),
  - Sensitivity list (if required) must have clock and any asynchronous control signals (not required for wait)

- These rules subsume the templates from IEEE 1076.6-1999

If and Synchronous Conditions

- Synchronous conditions can be coded with clock

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

- Benefit: Code more compact and simulation efficient

Wait and Synchronous Conditions

SynthWorks

- Synchronous conditions can be coded with clock

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

- Alternate similar clock edge forms to the above:

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

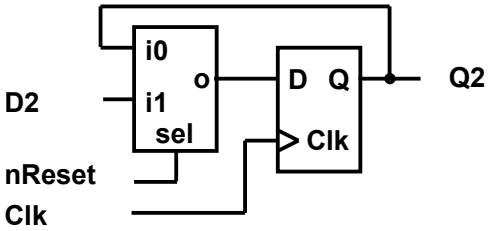
- Benefit: Code more compact and simulation efficient

Problematic Asynchronous Reset

SynthWorks

```
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
```

- When coded as above, Q2 produces the following logic:



Asynchronous Reset

- Asynchronous reset can be coded separately from clock:

```
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
```

- Benefit:
  - Allows registers with and without reset to be correctly coded in same process.

Concurrent Assignments

- Register and Latch creation with concurrent assignments

Register

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

Register w/ Async Reset

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

Latch

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

- Benefit: Greatly simplified coding style

Registers & Latches in Subprograms

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

```
-- Create Registers & Latches
-- with procedure calls
-- Three Registers
DFFR(Clk, nRst, D,    Reg1) ;
DFFR(Clk, nRst, Reg1, Reg2) ;
DFFR(Clk, nRst, Reg3, Q) ;

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


Multiple Edge Registers

SynthWorks

```
DualEdgeFF : process( nReset, Clk1, Clk2)
begin
  if (nReset = '0') then
    Q <= '0' ;

  elsif rising_edge(Clk1) then   -- Functional Clock
    Q <= D ;

  elsif rising_edge(Clk2) then   -- Scan Clock
    Q <= SD ;
  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;
```

Register Using Both Edges of Clk

SynthWorks

```
DualEdge_Proc: process (Clk, Reset) is
begin
  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;
```


RAM Entity

SynthWorks

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

entity  RAM  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  RAM ;
```

VHDL Generics make the RAM Address & Data parameterizable

Architecture **register** of **RAM** is

```
begin
  Ram_Proc: process (a, d, we) is
    type ram_type is array(0 to 2**DEPTH - 1) of
        std_logic_vector(WIDTH-1 downto 0);
    variable ram : ram_type;
  begin
    wait until clk = '1' ;
    if we = '1' then                          -- write RAM
      ram(to_integer(unsigned(a))  := d ;
    end if;
    q <= ram(to_integer(unsigned(a)) ;        -- read RAM
  end process Ram_Proc;
end architecture register ;
```

- With ram_block attribute, creates a RAM with registered IO:

```
attribute ram_block of ram : signal is "";
```

- With logic_block attribute, creates a registers:

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

- Without either, synthesis tool's choice

Architecture **latch** of **RAM** is

```
begin
  Ram_Proc: process (a, d, we) is
    type ram_type is array(0 to 2**DEPTH - 1) of
        std_logic_vector(WIDTH-1 downto 0);
    variable ram : ram_type ;
  begin
    if we = '1' then                          -- write RAM
      ram(to_integer(unsigned(a))  := d ;
    end if;
    q <= ram(to_integer(unsigned(a)) ;        -- read RAM
  end process Ram_Proc;
end architecture latch;
```

- With ram_block attribute, creates a RAM:

```
attribute ram_block of ram : signal is "";
```

- With logic_block attribute, creates a latches:

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

- Without either, synthesis tool's choice

```
architecture RTL_CASE 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_CASE ;
```

- With logic_block attribute, creates combinational logic:

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

ROM

SynthWorks

```
architecture RTL2 of ROM is
    type rom_type is array(0 to 7) of
        std_logic_vector(3 downto 0);

    constant ROMINIT : rom_type := (
        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;
```


Statemachines


```
type StateType is (S0, S1, S2, S3, S4);
signal state : StateType;
. . .
StateProc : 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;
```

What if anything is specified for states S5, S6, and S7?

Logically these states don't exist, so nothing is implied

Safe Statemachines

- Setting the FSM_COMPLETE attribute to true requires a synthesis tool to use the transitions specified by the VHDL default assignment to specify transitions for unused states in the implementation.

```
attribute FSM_COMPLETE of state : signal is TRUE;
```

What if anything is specified for states S5, S6, and S7?

Transition to S0 as specified by the others statement.

- Note: When FSM_COMPLETE attribute is true, it is an error if the statemachine has any unreachable states.

Specifying State Values

- The FSM_STATE attribute permits specification of a state encoding style such as binary, gray, one_hot, one_cold, or AUTO:

```
attribute FSM_STATE of state : signal is "BINARY" ;
```

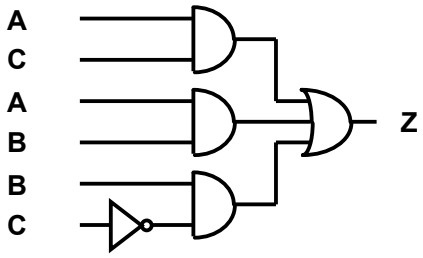
- The FSM_STATE attribute also permits specification of a state encoding enumeration value:

```
attribute FSM_STATE of state : signal is
   "0000 0011 0110 1100 1001" ;
```

  - The above example gives the encoding a hamming distance of two which is good for safe statemachine environments. It was perhaps an oversight that state encodings for HAMMING2 and HAMMING3 were left out.

Preserving Logic Terms

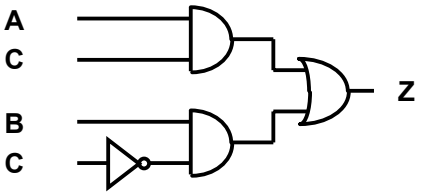
- Required Circuit:



- Code

```
Z <=
     (A and C) or
     (A and B) or
     (B and not C) ;
```

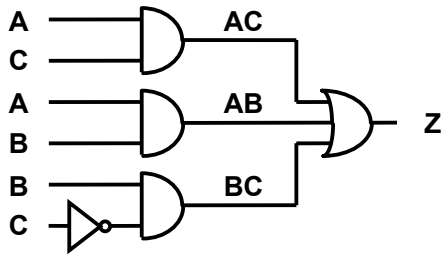
- Resulting Circuit:



- What happened?
  - The term AB is reducible and was removed by the synthesis tool.
  - Is it necessary to keep AB?
    - For asynchronous logic, the AB term prevents glitches

Preserving Logic Terms

- Step 1: Create Intermediate signals



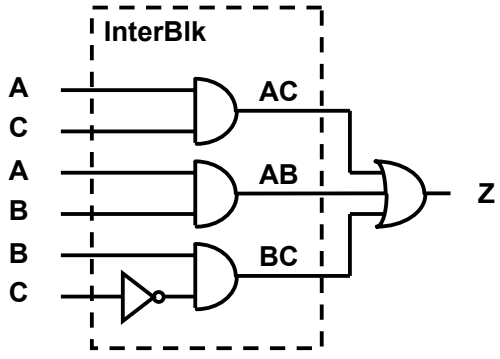

```
signal AC, AB, BC : signal;
. . .
AC <= A and C ;
AB <= A and B ;
BC <= B and not C ;

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

- Step 2: Apply the keep attribute to a AC, AB, BC to preserve them through synthesis. This is equivalent to inserting a non-movable buffer on the signal.

```
attribute KEEP of AC, AB, BC : signal is TRUE;
```

Preserving Logic Terms

- Step 1: Create Hierarchy Using VHDL Block Statement




```
InterBlk : block
begin
  AC <= A and C ;
  AB <= A and B ;
  BC <= B and not C ;
end block ;

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

- Step 2: Apply the CREATE_HIERARCHY attribute to the block statement label to preserve it through synthesis.

```
attribute CREATE_HIERARCHY of InterBlk : label is TRUE;
```


Resolution of Problems:

- Combinational Logic and Incomplete Sensitivity Lists:

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

- Resolution:
  Synthesis tool should produce an error and not produce any results

---

Resolution of Problems:

- Is the following a register or a latch?

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

- Cannot be a latch since it has an incomplete sensitivity list
- Cannot be a register since it has no clock edge specification

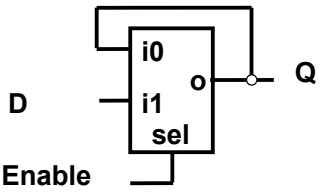
- Resolution:
  Synthesis tool should produce an error and not produce any results

Resolution of Problems:

- Problematic Latch Coding Style:

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

Either creates a latch or:



- Resolution: By default it should create a latch
  - ✦ If combinational logic is needed use the attribute:

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

Vendor Support of Standards

- Business view of supporting EDA standards
  - Supporting a feature of a standard is an investment
  - Feature support is market driven
    - If you don't ask, they don't support it.

- ✦ If you see new features you want to use, tell your EDA vendor

On-Going VHDL Projects

- Significant enhancements are currently being made to VHDL.
- This paper summarizes the efforts being made by the following IEEE working groups:

Agenda	Website
VHDL-200X	http://www.eda.org/vhdl-200x
IEEE 1164	http://www.eda.org/vhdl-std-logic
IEEE 1076.3/numeric std	http://www.eda.org/vhdlsynth
IEEE 1076.3/floating point	http://www.eda.org/fphdl
IEEE 1076.6	http://www.eda.org/siwg
VHDL-200X Fast Track	http://www.eda.org/vhdl-200x/vhdl-200x-ft

Caution: All activities here are work in progress.

SynthWorks VHDL Training

Comprehensive VHDL Introduction 4 Days
http://www.synthworks.com/comprehensive_vhdl_introduction.htm
A design and verification engineers introduction to VHDL syntax, RTL coding, and 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

