Synthesizable VHDL Models for FSMs, A how-to

By Alan L. Hege, University of North Carolina at Charlotte

VHDL is a hardware description language (VHSIC Hardware Description Language (VHSIC – Very High Speed Integrated Circuit)). That's right an acronym within an acronym. It was originally intended to simulate complex logic systems at a high level. A subset of the instructions can be used to actually describe a logic circuit that can be physically built from the instructions. The process of converting the code to a circuit implementation is called *'Synthesis'*. If the VHDL code synthesized, it is said to be a *'synthesizable model'*. Generally speaking there are two forms of modeling within VHDL; (1) the structural model and (2) the behavioral model. The structural model is basically the built up from primitive gate definitions and a net-list (i.e., text description of the components and the way in which they are wired.) Structural modeling is very conducive in describing hierarchal systems. Behavioral descriptions are where the power of VHDL is realized. As the name implies, we are describing the behavior of the circuit using VHDL code.

The intention of this document is to allow the student to easily model Finite State Machines (FSM) using VHDL. It is not intended to teach VHDL. We will be using VHDL models of FSMs to create circuits much easier than by hand as already demonstrated in class. The 'by hand' method (or manual method) of FSM design consists of (1) creating a State Diagram, (2) determine the State Encoding scheme, (3) filling in the State Transition Table and (4) synthesizing the circuit. With VHDL behavioral modeling we will go directly from a State Diagram to the behavioral model, then use software to do the synthesis and we're done.

Concurrence:

> Although VHDL looks like a programming language for a microprocessor, it is not. It has a major difference … all statements are executing at the same time. This property is called *'concurrence'*. Microprocessor code executes sequentially (one instruction after another).

Templates:

Many different CAD packages give templates as a starting point for designing a VHDL module/model. The figure to the right**Error! Reference source not found.** is a VHDL template that is created with Xilinx ISE Project Navigator … some of the comment lines are removed for conciseness.

```
----------------------------------------------------
-- Company:        UNC Charlotte
-- Engineer:       A. Hege
--
----------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity temp is
    Port ( clk : in std_logic;
           I : in std_logic;
           Z : out std_logic;
           reset : in std_logic);
end temp;

architecture Behavioral of temp is

begin


end Behavioral;
```

Format:

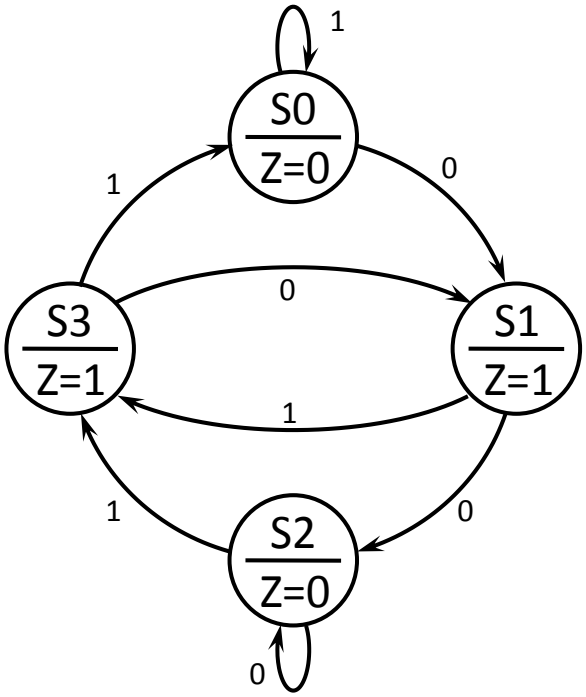
For the purposes of this class do not modify the ‘library’ and ‘use’ statements shown.

The green colored text are comments; they are preceded with two dashes. The blue colored text are reserve words and have a specific meaning within VHDL. The magenta colored text are VHDL defined signal types. The black colored text are the variable and names the user has given.

Under the ‘entity’ line is the ‘Port’ section. The port section defines the port mapping for the module/model, e.g., it defines the I/O for the module. For this module, the inputs are ‘clk’, ‘I’ & ‘reset’ and the output is ‘Z’.

Example FSM:

The design methodology will be given via an example. In this example we will be designing the VHDL behavioral model for a dual edge detector circuit (gives a pulse one clock cycle wide on both edges of the input signal 'I'). The figure to the right gives the state diagram for the system.



For this design, we will let the Synthesizer determine the state encoding or we could force the state encoding type via CAD software; a tutorial exists for doing this for the Xilinx ISE Project Navigator 7.1i software. For ease of coding and simulation, we will enumerate the states as follows:

Type stateType is (s0, s1, s2, s3);

Where 'stateType' is a new signal type for our design. Next, we will define the signals for next-state (ns) and current-state (cs).

signal ns: stateType;
signal cs: stateType;

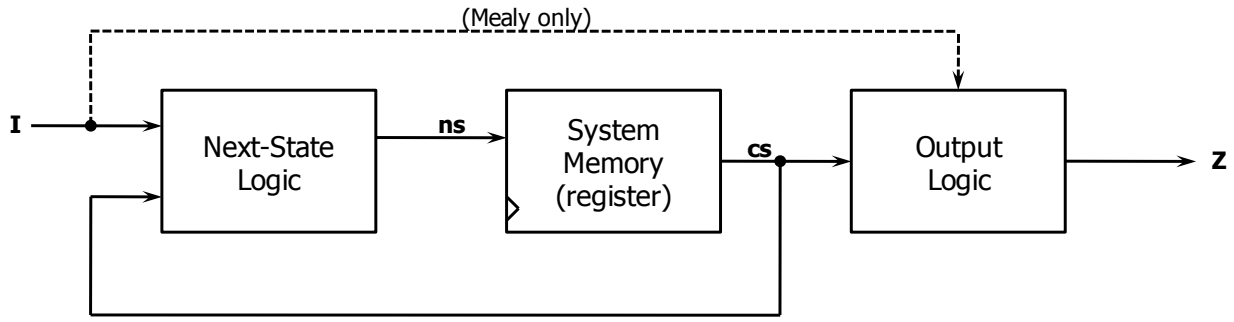
or

signal cs, ns: stateType;

These are considered internal signals to the module and have a specific location within the module. Internal signals are defined on the lines between 'Architecture …' and 'begin' … see the figure to the right.

```
----------------------------------------------
-- Company:        UNC Charlotte
-- Engineer:       A. Hege
--
----------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declarat
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity temp is
    Port ( clk : in std_logic;
           I : in std_logic;
           Z : out std_logic;
           reset : in std_logic);
end temp;

architecture Behavioral of temp is
   type stateType is (s0, s1, s2, s3);
   signal cs, ns: stateType;
begin


end Behavioral;
```

Now that the setup is complete we will start coding the blocks in the FSM. Remember that FSMs can be thought of in terms of the following block diagram.



Each block in the state diagram will be coded separately, starting with the Next State Logic block. The logic coding resides between the 'begin' and the 'end Behavioral;' statements.

In VHDL, conditional statements like 'if-then-else' and 'case' statements must be within a 'process' statement. Process statements have the following format:

Process()
begin

{other VHDL statements}

End process;

The author usually delineates his blocks with a comment statement '—Next-state Logic Block'. In parenthesis next to the process line is the 'sensitivity list'. The sensitivity list contains signal names that will trigger the process when the signal level changes.

To the right is the next-state logic block for our design. A lot has changed … start by looking at the key elements of the Process statement. The main element within the process statement is the 'case' statement. In the case statement each state is addressed and the next-state vector (ns) is assigned new values based on the inputs (I). The user can assign new values to signals, multiple times, within a process statement.

```
    Port ( clk : in std_logic;
           I : in std_logic;
           Z : out std_logic;
           reset : in std_logic);
end temp;

architecture Behavioral of temp is
   type stateType is (s0, s1, s2, s3);
   signal cs, ns: stateType;
begin

   --Next-state Logic
   process(cs, I)
   begin
      case cs is
         when s0 =>
            if I='0' then
               ns <= s1;
            else
               ns <= s0;
            end if;

         when s1 =>
            if I='1' then ns <= s3;
            else ns <= s2;
            end if;

         when s2 =>
            if I='1' then ns <= s3; else ns <= s2; end if;

         when s3 =>
            if I='0' then ns <= s1; else ns <= s0; end if;

         when others =>
            ns <= s0;

      end case;
   end process;


end Behavioral;
```

However, there can be only one signal source. The process statement counts as one signal source for a signal. For example, the following would not be allowed by the Synthesizer:

```
Process(a)                      -- process to make a 4-input Mux
begin
        if a="00" then
                y <= i0;        -- assign value on i0 to the signal y
        elsif a="01" then
                y <= i1;
        elsif a="10" then
                y <= i2;
        else
                y <= i3;
        end if;
end process;

Y<= '0' when reset='1' else y<=yint;      -- conditional assignment statement
```

This code snippet would be incorrect because the signal y has more than one source. Another example of too many sources would be as follows:

```
Process(a)
begin
        if a="00" then          y <= i0;
        elsif a="01" then       y <= i1;
        elsif a="10" then       y <= i2;
        else                    y <= i3;
        end if;
end process;

Process(b)
begin
        if b="00" then          y <= i0;
        elsif b="01" then       y <= i1;
        elsif b="10" then       y <= i2;
        else                    y <= i3;
        end if;
end process;
```

This snippet would be invalid because each process statement is sourcing y. Multiple sources is not allowed.

Now let's discuss the syntax of the different VHDL statements. First, binary values can be denoted in multiple ways. For simplicity the author will only discuss std_logic formats. Std_logic is a VHDL signal

type. Std_logic is a discrete signal type and the values must be written as a one or a zero with single quotes around the value for example a zero would be '0' and a one would be '1'. Strictly speaking the std_logic type has nine possible values: 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H' & '-'. For the scope of this document will only discuss using '0's and '1's. This is mentioned here because the student must know that just because something is not a '0' does not mean it is automatically a '1'. Another type of signal we will be using is called the std_logic_vector. The std_logic vector is a bus or 'bit vector'. Bit vector values must be expressed in terms of one's and zero's enclosed in double quotes. For example, the value 3Ah in a 7-bit bus would be expressed as "0111010" in VHDL. The VHDL code snippet, to the right, would be a VHDL module for assigning the value 3Ah to a 7-bit vector z.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if i
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity temp2 is
    Port ( z : out std_logic_vector(6 downto 0));
end temp2;

architecture Behavioral of temp2 is

begin

    z <= "0111010";

end Behavioral;
```

Notice the double quotes around the binary value. Also, notice the port-map for the module

Std_logic_vector(6 downto 0)

This defines the signal z as being an array of signals having indexes from 6 to 0; 6 in this case is the msb and 0 is the lsb. Treating bus members as individual signals would be performed as follows: z(6)=0, z(5)=1, z(4)=1, z(3)=1, z(2)=0, z(1)=1, & z(0)=0. For a better understanding of how bit vectors (busses) work in VHDL, consider shifting z in the above example left one bit and the shift- in value would be a one:

z1 <= z(6 downto 1) & '1';

The ampersand (&) is the operator for concatenating two vectors. The new value z1 would now contain the value "1110101". Of course, more would need to be done to the module above in order for this value to have any meaning (not produce an error). For instance, the user would need to define the signal. If it were an internal signal then the signal definition, placed between the 'architecture …' line and the 'begin' statement would need to be:

signal z1: std_logic_vector(6 downto 0);

Back to the example FSM, within the case statement were If-then-else statements. See Appendix A for the various syntax forms. If-the-else statements can only appear within process statements. If-then-else and case statements can be nested for complex logical behavior.

State Memory Block:

The state memory block is nothing but a register. This can be defined with a process statement that has clk as the sensitivity list as follows:

```
process(clk)
begin
        If clk'event and clk='1' then
                Q <= D;
        end if;

end process;
```

In this process clk is examined and on the rising edge the value on D is assigned to Q. D and Q could be a bus in which case this would be a register or D and Q could be discrete signals in which case this would be a D-type flip-flop. For a negative edge, the conditional part of the if-then-else statement would be:

```
        If clk'event and clk='0' then
                Q <= D;
        end if;
```

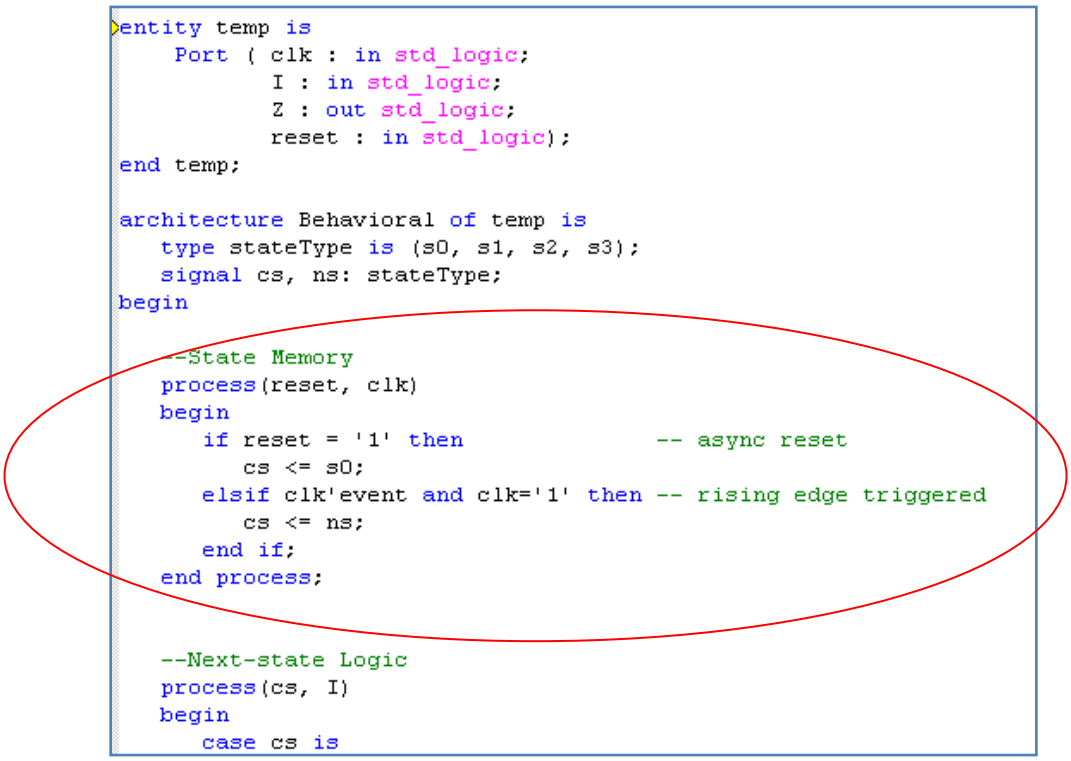
This could be read as "when the clock changes and the new value is zero then assign D to Q" … e.g., on the negative edge. Other variations would be when a reset is present. For an asynchronous reset:

```
process(reset, clk)
begin
        if reset='1' then
                cs <= s0;
        elsif clk'event and clk='1' then
                cs <= ns;
        end if;

end process;
```

Reset is active-high and the register is called cs (current state). Notice reset is independent of the clk … definition of asynchronous. The following snippet would be used if a synchronous reset (clear) is required:

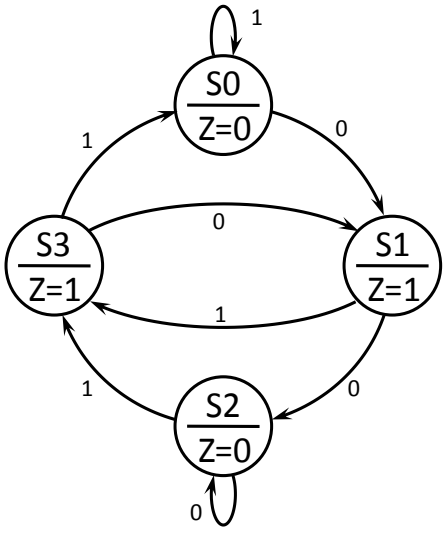
```
process(reset, clk)
begin
        if clk'event and clk='1' then
                if reset='1' then
                        cs <= s0;
                else
                        cs <= ns;
        end if;

end process;
```

For this example, an asynchronous reset is required. See the figure below for the state memory block design solution.



```
entity temp is
    Port ( clk : in std_logic;
           I : in std_logic;
           Z : out std_logic;
           reset : in std_logic);
end temp;

architecture Behavioral of temp is
   type stateType is (s0, s1, s2, s3);
   signal cs, ns: stateType;
begin

   --State Memory
   process(reset, clk)
   begin
      if reset = '1' then              -- async reset
         cs <= s0;
      elsif clk'event and clk='1' then -- rising edge triggered
         cs <= ns;
      end if;
   end process;


   --Next-state Logic
   process(cs, I)
   begin
      case cs is
```

Output Logic Block:



As a reminder of the state diagram it is shown again here to analyze the output logic. Here we see that the output z is true when in states s1 & s3. The following concurrent statements could be used:

Z <= '1' when cs=s1 or cs=s3 else '0';

or

with cs select
Z <= '1' when s1 | s3,
'0' when others;

If the outputs are more complicated, then a process statement could be used. Note in the last statement the reserve word 'others' is used. It means to assign zero to Z when anything other than s1 and s3 are the values of cs.

```
entity temp is
    Port ( clk : in std_logic;
           I : in std_logic;
           Z : out std_logic;
           reset : in std_logic);
end temp;

architecture Behavioral of temp is
   type stateType is (s0, s1, s2, s3);
   signal cs, ns: stateType;
begin

   --Output Logic
   Z <= '1' when cs=s1 or cs=s3 else '0';

   --State Memory
   process(reset, clk)
   begin
      if reset = '1' then              -- async reset
         cs <= s0;
```

This project synthesized without any errors. Within Xilinx Project Navigator you can examine the console window, after the module was synthesized, and discover that grey code state encoding was used. This was determined by the software … notice in the design that we did not assign values to the states (s0, s1, s2 & s3). There are ways to force the synthesizer to use a specific state encoding scheme and they are covered in a tutorial (located at http://www.coe.uncc.edu/~alhege/Xilinx Tutorials/Misc Tutorials/Xilinx FSM notes.pdf).

Appendix A: VHDL Syntax (abridged)

Defining signals:

- I/O ports on for the modules are created by Xilinx ISE Project Navigator Wizard for VHDL module creation.
- Internal signals (between 'Architecture …' line and 'begin' line):
    - Constant declaration and value assignment:
      Constant MAXCOUNT: std_logic_vector(7 downto 0) := "10101010"
    - Discrete signal:
      signal reset: std_logic;
    - Bus:
      signal kount: std_logic_vector(31 dowto 0);
    - Enumeration:
      type sType is (init, firstState, secState, s3, wait);
    - Use of enumerate type above:
      signal d: sType;

Relational Operators (conditional operators):

- '=' – equals
- '/=' – not equals
- '>' – greater than
- '>=' – greater than or equal
- '<' – less than
- '<=' – less than or equal

Assignment Statements:

- Signal-name <= expression;

- Conditional Assignment:

signal-name <= *expression* when *boolean-expression* else
expression when *boolean-expression* else
…
expression when *boolean-expression* else
expression;

- Selected signal-assignment:

with expression select
signal-name <= *signal-value* when *choices*,
signal-value when *choices*,
…
Signal-value when *choices;*

Example:

```
with N select
    F <=    '1' when "0001",
            '1' when "0010",
            '1' when  "0011" | "0101" | "0111",
            '1' when "1011" | "1101",
            '0 when others;
```

Conditional Statements:

- If-then-else:
    - if *boolean-expression* then *sequential-statements*
      end if;
    - if *boolean-expression* then *sequential-statements*
      else *sequential-statements*
      end if;
    - if *boolean-expression* then *sequential-statements*
      elsif *boolean-expression* then *sequential-statements*
      …
      elsif *boolean-expression* then *sequential-statements*
      end if;
    - if *boolean-expression* then *sequential-statements*
      elsif *boolean-expression* then *sequential-statements*
      …
      elsif *boolean-expression* then *sequential-statements*
      else *sequential-statements*
      end if;

- case:
    - case expression is
        when choices => sequential-statements
        …
        when choices => sequential-statements
      end case;

Appendix B: Synthesizable VHDL Behavioral Model Skeleton

Declaration - States:

```
Architecture Behavioral of FSM_name is

        type sType is (init, s1, s2);
        signal cs, ns: sType;

begin

        {{{model body}}}

End Behavioral;
```

Model Body – Next-state Logic Block:

```
process(cs, {inputs})
begin
        case cs is
                when init =>
                        {{{next-state options}}}

                when s1 =>
                        {{{next-state options for s1}}}

                when s2 =>
                        {{{ next-state options for s2}}}

                when others =>
                        {{{next-state options for everything else}}}

        end case;
end process;
```

Model Body – State Memory block:

Basic register ns is D and cs is Q (rising-edge triggered):

```
process(clk)
begin
        if clk'event and clk = '1' then
                cs <= ns;
        end if;

end process;
```

Basic register (trailing-edge triggered):

```
process(clk)
begin
        if clk'event and clk = '0' then
                cs <= ns;
        end if;

end process;
```

Register w/ Asynchronous reset:

```
process(clk, reset)
begin
        if reset ='1' then
                cs <= init;
        elsif clk'event and clk = '1' then
                cs <= ns;
        end if;

end process;
```

Register w/ async reset and clock enable(CE):

```
process(clk, reset, CE)
begin
        if reset = '1' then
                cs <= init;
        elsif clk'event and clk = '1' and CE = '1' then
                cs <= ns;
        end if;

end process;
```

Register w/ synchronous reset and clock enable(CE):

```
process(clk, reset, CE)
begin
        if clk'event and clk = '1' then
                if reset = '1' then
                        cs <= init;
                elsif CE = '1' then
                        cs <= ns;
                end if;
        end if;

end process;
```

Model Body – Output Logic Block:

- Use Simple Assignment, Conditional Assignment, Selected Signal Assignment or any combination of these statements or combo with the following.

- Use a process statement where outputs may be dependent on the inputs (case, or if-then-else).