ENGR 2323 Digital Design Lab
State Machine Design using VHDL

VHDL Designs

A VHDL design consists of two parts: the entity and the architecture. The entity specifies the interface of the design and the architecture specifies the operation of the design.

Structural architectures are similar to a circuit schematic in how design is described; the operation is described in terms of components and the interconnections between the components. Behavioral architectures are similar to a high-level program implementation of an algorithm; the operation is described in terms of the algorithm performed.

Behavioral architectures can be used to describe sequential circuit operation in terms of the states, state transitions, and outputs. The architecture for a sequential circuit can come from the state transition table or state diagram. The architecture for more complex designs can come from a register transfer language description of the design.

VHDL Process Statements

A process statement is executed concurrently with other concurrent statements. The statements inside of a process statement are executed sequentially like in a typical high-level programming language.

The general format of an architecture using a process statement is shown in Figure 1.

```
ARCHITECTURE behavior of design IS
signal declarations
BEGIN
   PROCESS (process sensitivity list)
   BEGIN
      sequential statement
      sequential statement
      etc.
   END PROCESS;

   concurrent statement
   concurrent statement
      etc.

END behavior;
```

Figure 1. Architecture using Process

The sensitivity list of the process must contain all signals that if changed the process needs to respond to. The process starts execution when an event (signal value change) for one of the signals in the sensitivity list occurs. The process statements are executed sequentially in order.

When the last statement is executed, the process is finished, and any signal assignment is performed. The process is not executed again until another event occurs for a signal in the sensitivity list.

For synchronous sequential circuits, the event starting a process is typically a change in the synchronizing signal (clock).

State Memory

For synchronous sequential circuits, flip-flops or registers based on flip-flops are used for the state memory. For asynchronous sequential circuits, latches are used for the state memory. State memory can be inferred in VHDL by using a conditional statement that does not specify all the possible cases (if statement with no else portion). The value of the signal when not specified in this case is interpreted as a hold of the last value. The VHDL segment of Figure 2 illustrates this for the state transitions of a synchronous sequential circuit with asynchronous reset.

```
PROCESS(CLOCK, RESETN)
BEGIN
   IF RESETN = '0' THEN
      STATE <= STATE0;
   ELSIF RISING_EDGE(CLOCK) THEN
      CASE STATE IS
         WHEN STATE0 =>
            IF X = '0' THEN
               STATE <= STATE0;
       ELSE
        STATE <= STATE1;
           END IF;
       WHEN STATE1 =>
         IF X = '0' THEN
          STATE <= STATE0;
       ELSE
          STATE <= STATE1;
       END IF;
        END CASE;
   END IF;
END PROCESS;
```

Figure 2. VHDL Segment Illustrating State Memory

The sensitivity list of the process contains the CLOCK and RESETN signals. The state memory should only change if the state machine is reset (any time) or on the active portion of the clock (rising edge). Notice that the behavior of the circuit is not specified for cases where the RESETN is high and the CLOCK is not a rising edge. The VHDL compiler infers a hold and for these cases the STATE will hold its value (STATE <= STATE).

Sequential Circuit Design using VHDL

Following is a VHDL design of a modulo 4 counter (2-bit counter). The counter will count up (00, 01, 10, 11, 00, …) when the control is activated and hold at the current count when control is deactivated. This is the same design that was realized using a schematic design for lab 5. The state diagram and state transition table for the modulo-4 counter are provided in Figures 3 and 4. Since this will be designed using a behavioral architecture, the next state and output equations are not needed. For a structural architecture, the next state and output equations would be needed.

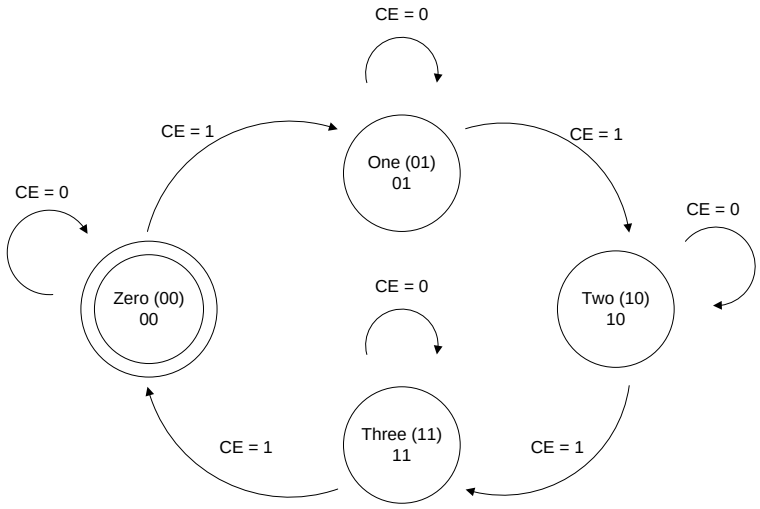

Figure 3. State Diagram of Modulo 4 Counter (2-bit counter)

Q1 Q0 CE	F1 F0	Q1+ Q0+
0 0 0	0 0	0 0
0 0 1	0 0	0 1
0 1 0	0 1	0 1
0 1 1	0 1	1 0
1 0 0	1 0	1 0
1 0 1	1 0	1 1
1 1 0	1 1	1 1
1 1 1	1 1	0 0

Figure 4. State Transition Table of Modulo 4 Counter (2-bit counter)

The entity of the VHDL design for the modulo 4 counter is given in Figure 5 and the architecture is given in Figure 7.

The counter inputs are the clock (CLOCK), reset (RESETN), and count enable (CE). The counter output is a 2-bit signal corresponding to the count value (F).

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY counter IS
PORT(CLOCK, RESETN, CE : IN STD_LOGIC;
       F : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END counter;
```

Figure 5. Module 4 Counter Entity

The architecture consists of a signal STATE of the user defined type STATETYPE used to specify the current and next state; a PROCESS statement specifying the next state behavior, and a WHEN ELSE statement specifying the output as a function of the state. Remember that statements in the process are executed sequentially and the process statement is executed concurrently with the output logic statement.

The sensitivity list of the process contains the CLOCK and RESETN signals but not the CE signal. State transitions should only occur if the asynchronous reset is activated or on a rising edge of the clock. This architecture is not using explicit state variables, Q1 and Q0, but there are times where explicit state assignment would be desirable. The example in Appendix A uses explicit state assignment. An alternative to using the RISING_EDGE function to detect the rising edge of the clock is to check that both CLOCK'EVENT and CLOCK = '1' are true.

The functional simulation for the design is given in Figure 6.

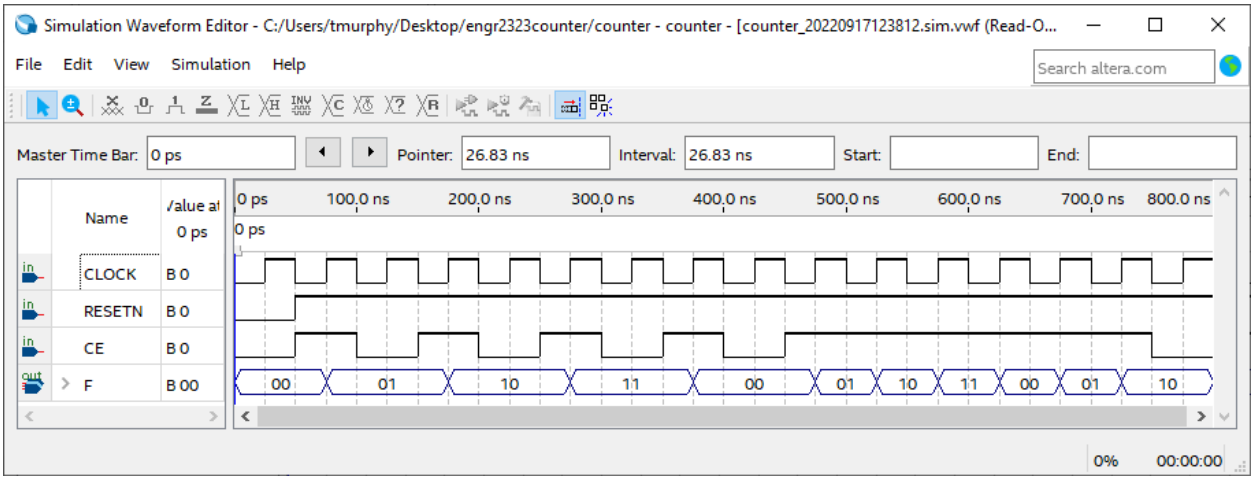

Figure 6. Module 4 Counter Functional Simulation

To verify the operation of the counter, for each count the counter holds the count (CE = 0) and then the counter counts up by one (CE = 1). This is repeated for all four states verifying the operation for all the state transitions. The simulation also shows the normal counting operation by having the count enabled enough clock cycles to count through the full sequence and roll over.

```
ARCHITECTURE behavior of counter IS
TYPE STATETYPE IS (COUNT0, COUNT1, COUNT2, COUNT3);
SIGNAL STATE : STATETYPE;
BEGIN
   -- next state logic
   PROCESS(CLOCK, RESETN)
   BEGIN
      IF RESETN = '0' THEN
         STATE <= COUNT0;
      ELSIF RISING_EDGE(CLOCK) THEN
         CASE STATE IS
            WHEN COUNT0 =>
               IF CE = '1' THEN
                  STATE <= COUNT1;
        ELSE
           STATE <= COUNT0;
        END IF;
       WHEN COUNT1 =>
        IF CE = '1' THEN
           STATE <= COUNT2;
        ELSE
           STATE <= COUNT1;
        END IF;
       WHEN COUNT2 =>
        IF CE = '1' THEN
           STATE <= COUNT3;
        ELSE
           STATE <= COUNT2;
        END IF;
       WHEN COUNT3 =>
        IF CE = '1' THEN
           STATE <= COUNT0;
        ELSE
           STATE <= COUNT3;
        END IF;
         END CASE;
      END IF;
   END PROCESS;

   -- output logic
   F  <= "00" WHEN STATE = COUNT0 ELSE
      "01" WHEN STATE = COUNT1 ELSE
      "10" WHEN STATE = COUNT2 ELSE
      "11" WHEN state = COUNT3 ELSE
      "00";

END behavior;
```

Figure 7. Module 4 Counter Architecture

References

None

Appendix A, Design of a 2-bit Up and Down Counter

The state diagram for a 2-bit up and down counter is given in Figure 8 and the state transition table for the counter is given in Figure 9. The counter counts up through the 2-bit sequence when CE = 1 and UDN = 1 and counts down when CE = 1 and UDN = 0. When not enabled, the counter holds the current count.

The state names and codings are: count0 Q = 00, count1 Q = 01, count2 Q = 10, and count3 Q = 11. The output is the count, F = Q.

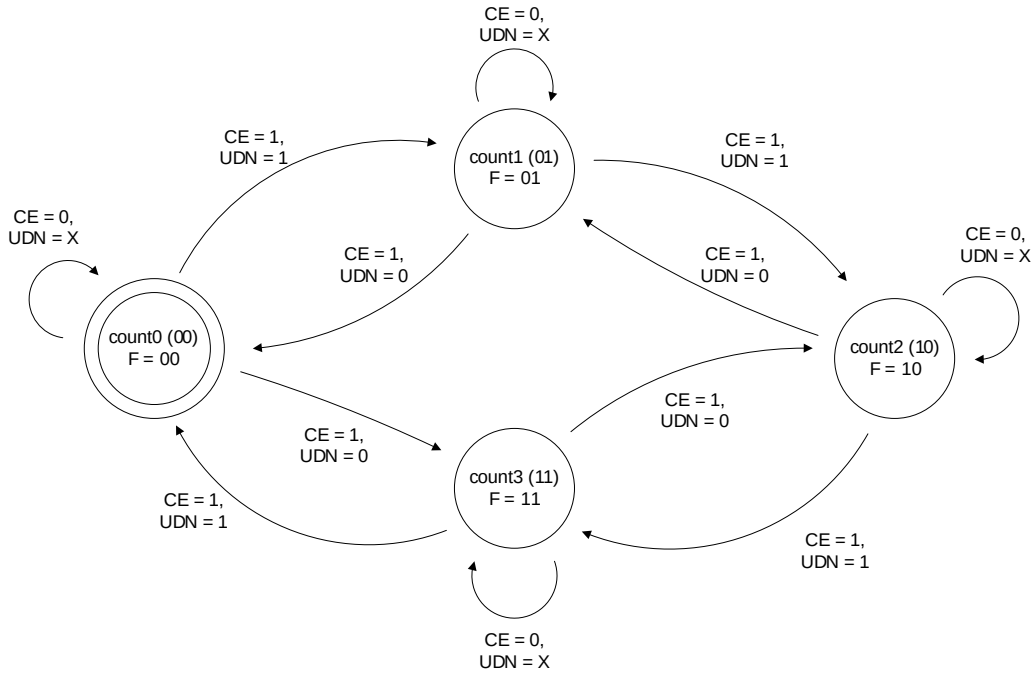

Figure 8. State Diagram for 2-bit Up and Down Counter

The next state and output expressions are not needed for a behavior design. The VHDL description for a behavior sequential circuit can come from the state diagram or state transition table.

The entity of the VHDL design for the 2-bit up and down counter is given in Figure 10. A portion of the architecture showing the process for the state transitions from count0 is given in Figure 11, and the state assignment and output logic are given in Figure 12. The full design is given in Appendix B.

Count	Q1 Q0 CE UDN	F1 F0	Q1+ Q0+
count0	0 0 0 0	0 0	0 0
	0 0 0 1	0 0	0 0
	0 0 1 0	0 0	1 1
	0 0 1 1	0 0	0 1
count1	0 1 0 0	0 1	0 1
	0 1 0 1	0 1	0 1
	0 1 1 0	0 1	0 0
	0 1 1 1	0 1	1 0
count2	1 0 0 0	1 0	1 0
	1 0 0 1	1 0	1 0
	1 0 1 0	1 0	0 1
	1 0 1 1	1 0	1 1
count3	1 1 0 0	1 1	1 1
	1 1 0 1	1 1	1 1
	1 1 1 0	1 1	1 0
	1 1 1 1	1 1	0 0

Figure 9. State Transition Table for 2-bit Up and Down Counter

The entity has the state as an output for testing purposes. Typically, the state is not an output of a sequential circuit.

The CE and UDN inputs are concatenated into a 2-bit signal to allow for next state selection based on both inputs using a CASE statement. The reset is asynchronous and does not depending upon the clock signal. Note in the conditional statements in the process, that single quotes are used to specify a 1-bit constant and double quotes are used to specify a multiple bit constant.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY counterupdown IS
PORT(CLOCK, RESETN : IN STD_LOGIC;
   CE, UDN  : IN STD_LOGIC;
   Q : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
      F : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END counterupdown;
```

Figure 10. Entity for 2-bit Up and Down Counter

```
ARCHITECTURE behavior of counterupdown IS
TYPE STATETYPE IS (COUNT0, COUNT1, COUNT2, COUNT3);
SIGNAL STATE : STATETYPE;
SIGNAL CE_UDN : STD_LOGIC_VECTOR(1 DOWNTO 0);

BEGIN
   -- concatenate the CE and UDN signals
   CE_UDN <= CE & UDN;

   -- next state logic
   PROCESS(CLOCK, RESETN)
        BEGIN
           IF RESETN = '0' THEN
              STATE <= COUNT0;
           ELSIF RISING_EDGE(CLOCK) THEN
              CASE STATE IS
                 WHEN COUNT0 =>
                    CASE CE_UDN IS
                       WHEN "00" => STATE <= COUNT0;
                       WHEN "01" => STATE <= COUNT0;
                       WHEN "10" => STATE <= COUNT3;
                       WHEN "11" => STATE <= COUNT1;
                    END CASE;
```

Figure 11. Architecture for 2-bit Up and Down Counter

A case statement is used to specify the state transitions since there are four state transitions from each state. In the modulo 4 counter example prior, the state transitions were specified using an if statement since there were only two transitions from each state.

```
   -- output logic
   F  <= "00" WHEN STATE = COUNT0 ELSE
     "01" WHEN STATE = COUNT1 ELSE
     "10" WHEN STATE = COUNT2 ELSE
     "11" WHEN state = COUNT3 ELSE
     "00";

   -- state assignments
   Q  <= "00" WHEN STATE = COUNT0 ELSE
     "01" WHEN STATE = COUNT1 ELSE
     "10" WHEN STATE = COUNT2 ELSE
     "11" WHEN state = COUNT3 ELSE
     "00";
END behavior;
```

Figure 12. Architecture for 2-bit Up and Down Counter

The state assignment and output logic are defined outside the process and are updated in parallel with state transitions in the process.

The functional simulation of the 2-bit Up and Down Counter is given in Figure 13. The simulation does not fully verify the operation of the counter as all of the count holds have not been tested.

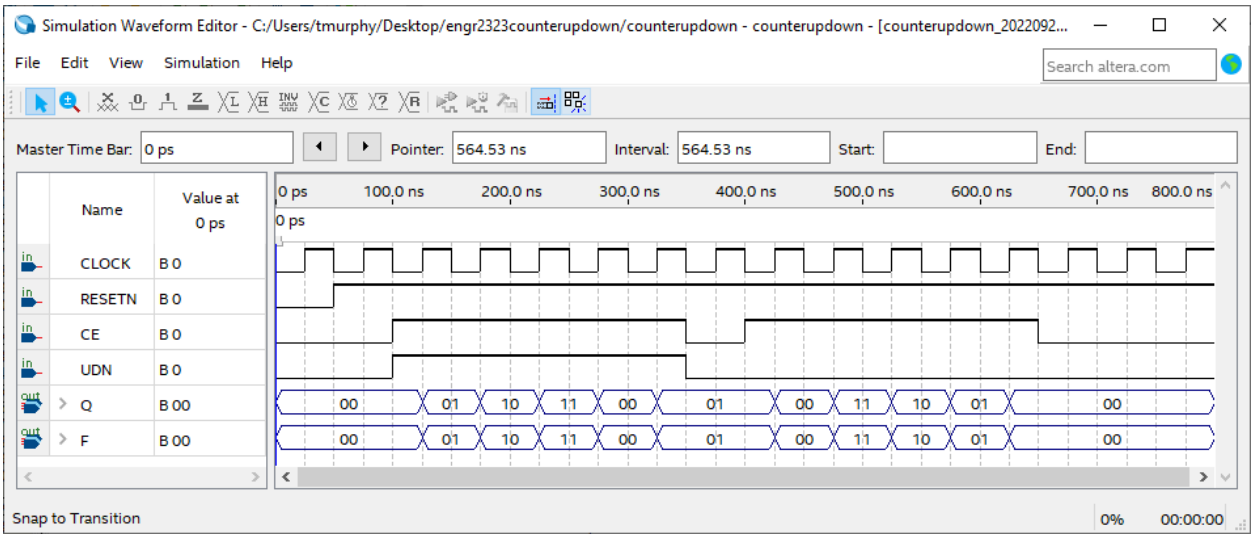

Figure 13. Functional Simulation for 2-bit Up and Down Counter

Appendix B, VHDL for 2-bit Up and Down Counter

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY counterupdown IS
PORT(CLOCK, RESETN : IN STD_LOGIC;
   CE, UDN  : IN STD_LOGIC;
   Q : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
   F : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END counterupdown;

ARCHITECTURE behavior of counterupdown IS
TYPE STATETYPE IS (COUNT0, COUNT1, COUNT2, COUNT3);
SIGNAL STATE : STATETYPE;
SIGNAL CE_UDN : STD_LOGIC_VECTOR(1 DOWNTO 0);

BEGIN
   -- concatenate the CE and UDN signals
   CE_UDN <= CE & UDN;

   -- next state logic
   PROCESS(CLOCK, RESETN)
      BEGIN
         IF RESETN = '0' THEN
            STATE <= COUNT0;
         ELSIF RISING_EDGE(CLOCK) THEN
            CASE STATE IS
               WHEN COUNT0 =>
                  CASE CE_UDN IS
                     WHEN "00" => STATE <= COUNT0;
                     WHEN "01" => STATE <= COUNT0;
                     WHEN "10" => STATE <= COUNT3;
                     WHEN "11" => STATE <= COUNT1;
                  END CASE;
               WHEN COUNT1 =>
                  CASE CE_UDN IS
                     WHEN "00" => STATE <= COUNT1;
                     WHEN "01" => STATE <= COUNT1;
                     WHEN "10" => STATE <= COUNT0;
                     WHEN "11" => STATE <= COUNT2;
                  END CASE;
               WHEN COUNT2 =>
                  CASE CE_UDN IS
                     WHEN "00" => STATE <= COUNT2;
                     WHEN "01" => STATE <= COUNT2;
                     WHEN "10" => STATE <= COUNT1;
                     WHEN "11" => STATE <= COUNT3;
```

```
            END CASE;
          WHEN COUNT3 =>
            CASE CE_UDN IS
               WHEN "00" => STATE <= COUNT3;
               WHEN "01" => STATE <= COUNT3;
               WHEN "10" => STATE <= COUNT2;
               WHEN "11" => STATE <= COUNT0;
            END CASE;
        END CASE;
      END IF;
   END PROCESS;

   -- output logic
   F  <= "00" WHEN STATE = COUNT0 ELSE
      "01" WHEN STATE = COUNT1 ELSE
      "10" WHEN STATE = COUNT2 ELSE
      "11" WHEN state = COUNT3 ELSE
      "00";

   -- state assignments
   Q  <= "00" WHEN STATE = COUNT0 ELSE
      "01" WHEN STATE = COUNT1 ELSE
      "10" WHEN STATE = COUNT2 ELSE
      "11" WHEN state = COUNT3 ELSE
      "00";
END behavior;
```

Last modified Thursday, September 22, 2022