

Modeling Concepts

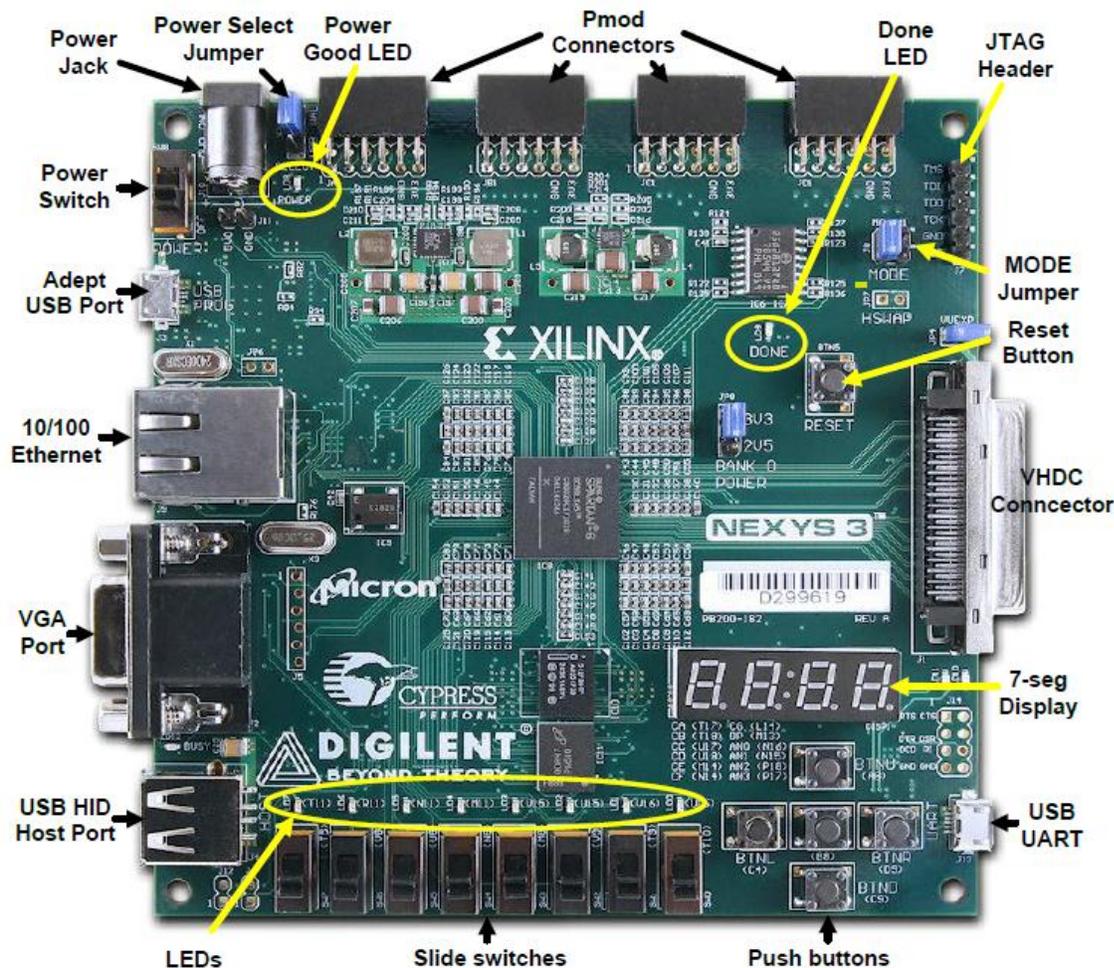
Introduction

The Very High Speed Integrated Circuit Hardware Description Language (VHDL) modeling language supports three kinds of modeling styles: dataflow, structural and behavioral. Dataflow and structural modeling are used to model combinatorial circuits whereas behavioral modeling is used for both combinatorial and sequential circuits. This lab illustrates the use of all three types of modeling by creating simple combinatorial circuits targeting Nexys3 board and using the PlanAhead software tool. *Please refer to the PlanAhead tutorial on how to use the PlanAhead tool for creating projects and verifying digital circuits.*

The Nexys3 board has the following components:

- 16Mbyte Cellular RAM (x16)
- 16Mbytes SPI (quad mode) PCM non-volatile memory
- 16Mbytes parallel PCM non-volatile memory
- 10/100 Ethernet PHY
- USB-UART and USB-HID port (for mouse/keyboard)
- 8-bit VGA port
- 100MHz CMOS oscillator
- 72 I/O's routed to expansion connectors
- GPIO includes 8 LEDs, 5 buttons, 8 slide switches and 4-digit seven-segment display

The Nexys3 board is shown below.



Objectives

After completing this lab, you will be able to:

- Create scalar and wide combinatorial circuits using dataflow, structural and behavioral modeling
- Write models to read switches and push buttons, and output on LEDs and 7-segment displays
- Simulate and understand the design output
- Create hierarchical designs
- Synthesize, implement and generate bitstreams
- Download bitstreams into the board and verify functionality

VHDL Structure

Part 1

Before describing the different modelling styles in VHDL, it is useful to describe a VHDL module components.

A VHDL module has a well-defined structure that may appear bewildering to someone just learning VHDL but allows the module to be defined in a clear and logical manner. A typical VHDL module has two main portions: (1) entity declaration and (2) architecture block. The entity declaration defines the module's input and output ports of a device. The architecture block in VHDL defines the functionality of the device.

```
entity example_code is port(
    port_1 : in std_logic;
    port_2 : out std_logic_vector(1 downto 0)
);
architecture example_code_arch of example_code is
...
end example_code_arch;
```

The entity may contain the port names, the port sizes and the directions (input/output). The architecture block may include instantiated components and local signals/nets. The architecture block can be further broken down into three sub-sections: (1) component declarations, (2) signal declarations and a (3) functional code. The component declaration is required to describe a hierarchical design. The signal declarations are required for local connections between various blocks within the functional code. The functional code can be described in number of ways, as explained later in this lab.

```
...
architecture example_code_arch of example code is
    component instance1 is port(
        port_in : in std_logic;
        ...
        port_out : out std_logic
    );

    component instance2 is port(
        ...
        port_out4 : out std_logic_vector(3 downto 0);
    );

    signal sig_a : std_logic;
    signal sig_b : std_logic_vector(1 downto 0);
    signal sig_c : std_logic_vector(3 downto 0);
    ...

begin
    comp1 : instance1
    port map (
        port_in => sig_a;
```

```

        ...
        port_out => sig_b;
    );

    comp2 : instance2
    port map (
        ...
        port_out4 => sig_c;
    );

    comp3 : instance1
    port map (
        ...
    );

    sig_a <= port_1 and sig_c(0);

    process(sig_c) begin
        port_2 <= sig_c(3 downto 1);
    end process;
end example_code_arch;

```

The functional block of architecture block is where the module functionality and how it is implemented are defined. The above example shows instantiated components with their ports mapped to signals. This allows multiple use of the same components as one would use multiple ICs of a same kind in a typical system.

Processes and direct assignments may also reside in the same architecture block. Assignments outside of processes are nearly always combinatorial, as can be seen in the example above where an **and** operation is performed on *port_1* and *sig_c(0)* and output to *sig_a*. Such statements are used to describe dataflow modeling. Processes are used if sequential operations need to be executed when specific signals toggle, as can be seen above where the sensitivity is to *sig_c*. If *sig_c* is a clock signal then the process block may describe a sequential behavior. Such blocks are used to describe functionality behaviorally.

Dataflow Modeling

Part 2

Dataflow modeling can be used to describe combinational circuits. The basic mechanism used is the concurrent assignment. In a concurrent assignment, a value is assigned to a **signal**. The syntax of a concurrent assignment statement is:

```
LHS_signal <= RHS_expression;
```

Where LHS_signal is a destination net of one or more bit, and RHS_expression is an expression consisting of various operators. The target in the expression can be one of the following:

1. A scalar net (e.g. 1st and 2nd examples above)
2. A vector net
3. Constant bit-select of a vector
4. Constant part-select of a vector
5. Combinations of any of the above

The assignment operations involve the basic Boolean functions (operators): **and**, **or**, **xor**, **nand**, **nor** and **xnor**. These are (by default) two inputs and one output. The example below shows dataflow modelling for a two input/one output circuit.

```
entity AND_gate is
```

```

port (a: in std_logic;
      b: in std_logic;
      c: out std_logic
);

architecture AND_gate_dataflow_arch of AND_gate is
begin
    c <= a and b;
end AND_gate_dataflow_arch;

```

Here is another example showing **or** function:

```
z <= x or y;
```

To have multiple inputs for a logical operator, one can cascade multiple operations in an assignment statement:

```
z <= v and w and x and y;
```

The example above represents a four input to one output **and** gate. Multiple inputs and various logical operations can be combined in a signal output function, such as:

```
z <= v and w or x nor y;
```

Interconnections between various objects must explicitly be done through nets. Nets may be scalar or vector and must be defined before they are used. For example,

```

signal y : std_logic; // scalar net
signal sum : std_logic_vector(3 downto 0); // vector net

```

STD_LOGIC defines a scalar net and STD_LOGIC_VECTOR defines a vector net with of a specified width/size in bits. The destination (LHS) can also be either a scalar or a vector.

There are many other operators supported by VHDL. The **&** operator concatenates two signals. For example,

```

signal m : STD_LOGIC; -- scalar
signal switches : STD_LOGIC_VECTOR (7 downto 0); -- vector
...
switches <= switches(7 downto 1) & m;

```

The operation above concatenates signal *m* as the last bit to form the eight bit wide switches.

The **not** operation can have only one input and out output. The logical value that is present at the input of the operation will be inverted at the output. For inputs with a constant logic value, pull-up and pull-down with a single output (no input) only are also supported in the VHDL syntax as **'H'** or **'L'** constant values if the destination is of STD_LOGIC. For example:

```

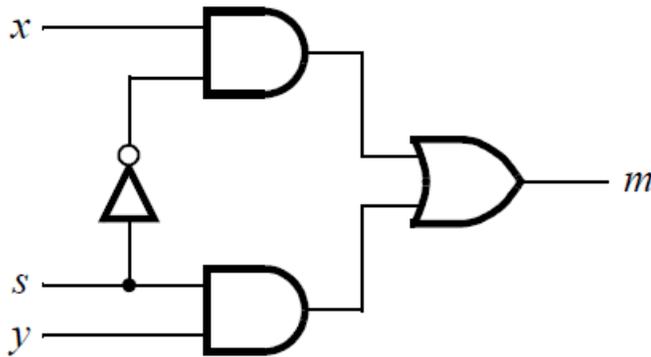
z <= 'H';

y <= 'L';

```

Dataflow modeling is useful when a circuit is combinational. An example is the multiplexer. A multiplexer is a simple circuit which connects one of many inputs to an output.

2-1. Create a 2-to-1 multiplexer using dataflow modeling.



- 2-1-1. Open PlanAhead and create a blank project called lab1_2_1.
- 2-1-2. Create and add the VHDL module with three inputs (x , y , s) and one output (m) using dataflow modeling.
- 2-1-3. Create and add the UCF file to the project. Assign **SW0** and **SW1** to x and y , **SW7** to s , and **LED0** to m (refer Step 2 of the PlanAhead Tutorial).
- 2-1-4. Synthesize the design.
- 2-1-5. Implement the design.
- 2-1-6. Generate the bitstream, download it in to the Nexys3 board, and verify the functionality.

2-2. Create a two-bit wide 2-to-1 multiplexer using dataflow modeling.

- 2-2-1. Open PlanAhead and create a blank project called lab1_2_2.
- 2-2-2. Create and add the VHDL module with two 2-bit inputs ($x[1:0]$, $y[1:0]$), a one bit select input (s), and two-bit output ($m[1:0]$) using dataflow modeling.
- 2-2-3. Create and add the UCF file to the project. Assign **SW0** and **SW1** to $x[1:0]$, **SW2** and **SW3** to $y[1:0]$, **SW7** to s , and **LED0** and **LED1** to $m[1:0]$.
- 2-2-4. Synthesize the design.
- 2-2-5. Implement the design.

2-2-6. Generate bitstream, download it in to the Nexys3 board, and verify the functionality.

Adding assignment delays

Delays can be added to concurrent assignments as seen below:

```
LHS_net <= RHS_expression after [delay] ns;
```

The statement is evaluated at any time any of the source operand value changes and the result is assigned to the destination net after the delay unit. For example,

```
out1 <= in1 and in2 after 2 ns; --perform the desired function and
assign the result after 2 nanoseconds.
```

Another example in which a scalar and vector nets are declared and used

```
signal A : in std_logic;      --scalar net declaration
signal B : in std_logic_vector (2 downto 0);  --vector nets
declaration
signal C : in std_logic_vector (3 downto 0);

C <= '0' & B & A after 5 ns; -- A & B are concatenated to a 1-bit '0'
value and is then assigned to vector C. The operation is executed after
5 ns.
```

2-3. Model a two-bit wide 2-to-1 multiplexer using dataflow modeling with net delays of 3 ns.

- 2-3-1. Open PlanAhead and create a blank project called lab1_2_3.
- 2-3-2. Create and add the VHDL module with two 2-bit inputs ($x0, x1, y0, y1$), a one bit select input (s), and two-bit output ($m0, m1$) using dataflow modeling. Each assignment statement should have 3 ns delay.
- 2-3-3. Create and add the UCF file to the project. Assign **SW0** and **SW1** to $x[1:0]$, **SW2** and **SW3** to $y[1:0]$, **SW7** to s , and **LED0** and **LED1** to $m[1:0]$.
- 2-3-4. Add the provided testbench (mux_2bit_2_to_1_dataflow_tb.vhd) to the project.
- 2-3-5. Simulate the design for 100 ns and analyze the output.
- 2-3-6. Synthesize the design.
- 2-3-7. Implement the design.
- 2-3-8. Generate the bitstream, download it in to the Nexys3 board, and verify the functionality.

Structural Modeling

Part 3

Structural modeling involves connecting instantiated components to define the functionality of a circuit. Component instantiations can be of other modules and/or device primitives. Using gate-level allow the construction of simple combinatorial circuits.

Repeating the example from Part 2 above,

```

entity AND_gate_structural is
port (a: in std_logic;
       b: in std_logic;
       c: in std_logic;
       d: out std_logic
);

architecture AND_gate_struct of AND_gate_structural is
  component and2 port
  (
    i0, i1 : in bit;
    o : out bit
  ) end component;
  Signal a_int : STD_LOGIC;
begin
  and_comp_1 : and2 port map (
    i0 => a;
    i1 => b;
    o => a_int;
  );
  and_comp_2 : and2 port map (
    i0 => a_int;
    i1 => c;
    o => d;
  );
end AND_gate_arch;

```

Components can also be connected via signals declared in the architecture block for more complex circuit implementations. The simple example below illustrates how this is done.

```

entity AND_OR_gate_structural is
port (a: in std_logic;
       b: in std_logic;
       c: in std_logic;
       d: out std_logic;
);

architecture AND_OR_gate_struct of AND_OR_gate_structural is
  component and2 port
  (
    i0, i1 : in bit;
    o : out bit
  ) end component;

  component or2 port
  (
    i0, i1 : in bit;
    o : out bit
  ) end component;

```

```

    Signal e : bit;

begin
    and_comp : and2 port map (
        i0 => a;
        i1 => b;
        o => e;
    );

    or_comp : or2 port map (
        i0 => c;
        i1 => e;
        o => d;
    );

end AND_gate_arch;

```

3-1. Re-create the earlier lab 2-2 using structural modeling.

- 3-1-1. Open PlanAhead and create a blank project called lab1_3_1.
- 3-1-2. Create and add the VHDL module with two 2-bit inputs ($x0$, $x1$, $y0$, $y1$), a one bit select input (s), and two-bit output ($m0$, $m1$) using dataflow modeling. Each assignment statement should have 3 ns delay.
- 3-1-3. Create and add the XDC file to the project. Assign **SW0** and **SW1** to $x[1:0]$, **SW2** and **SW3** to $y[1:0]$, **SW7** to s , and **LED0** and **LED1** to $m[1:0]$.
- 3-1-4. Synthesize the design.
- 3-1-5. Implement the design.
- 3-1-6. Generate the bitstream, download it in to the Nexys3 board, and verify the functionality.

Behavioral Modeling

Part 4

Behavioral modeling is used to describe complex circuits. In VHDL, behavioral modeling is done in the architecture block. Within the architecture block, processes are defined to model sequential circuits. The mechanisms (statements) for modeling the behavior of a design are:

```

--The following process is only used to initialize signals in a design
at the beginning of runtime.
process begin
    a <= '1';
    b <= '0';
    ...
    wait;
end process;

```

```

--The following process runs when any signal in the sensitivity list
has an event, i.e. change in a value.

```

```

process (c, d, e)
begin
    <behavioral code here>
end process;

```

A module may contain an arbitrary number of **process** statements and these may contain one or more statements within them. The statements appearing within the process statement body are categorized as procedural statements. The processes are executed in a concurrent manner (i.e. the order in which they appear in the model does not matter) with respect to each other whereas the procedural statements are executed in a sequential manner (i.e. the order in which they appear does matter). A procedural_statement is one of the following:

1. Procedural assignments
2. Conditional statements
3. Case statements
4. Loop statements
5. Wait statements

The first process block in the example above is executed at time 0. A **wait** statement is needed at the last line of the process block to stop the process block from executing again. The **process** statements with a sensitivity list are executed during the rest of the time whenever any of the signals in the sensitivity list changes.

The **process** statement may be synthesizable, and the resulting circuit may be a combinatorial or sequential circuit. In order for the model to generate a combinatorial circuit, the **process** block (i) should not have edge sensitive statements and must have all output generated in every conditional statement or case statement within the process.

Here is an example of a 2-to-1 multiplexer model. Note that begin and end statements in this example are redundant. The code is also truncated for better readability

```

signal m : STD_LOGIC_VECTOR;
...
process (x, y, s)
begin
    if (s='0') then
        m <= y;
    else
        m <= x;
    end if;
end;

```

4-1. Create a 2-to-1 multiplexer using behavioral modeling.

4-1-1. Open PlanAhead and create a blank project called lab1_4_1.

4-1-2. Create and add the VHDL module with three inputs (x, y, s) and one output (m) using behavioral modeling. Use the example code given above for modeling a multiplexer.

4-1-3. Add the UCF file, created in 1-1, to the project. Assign **SW0** and **SW1** to x and y, **SW7** to s, and **LEDO** to m.

4-1-4. Synthesize the design.

4-1-5. Implement the design.

4-1-6. Generate bitstream, download it in to the Nexys3 board, and verify the functionality.

4-2. Create a two-bit wide 2-to-1 multiplexer using behavioral modeling.

4-2-1. Open PlanAhead and create a blank project called lab1_4_2.

4-2-2. Create and add the VHDL module with two-bit input ($x[1:0]$, $y[1:0]$), a one bit select input (s), and two-bit output ($m[1:0]$) using behavioral modeling.

4-2-3. Add the UCF file to the project that you had created in 1-2. Assign **SW0** and **SW1** to $x[1:0]$, **SW2** and **SW3** to $y[1:0]$, **SW7** to s , and **LED0** and **LED1** to $m[1:0]$.

4-2-4. Synthesize the design.

4-2-5. Implement the design.

4-2-6. Generate bitstream, download it in to the Nexys3 board, and verify the functionality.

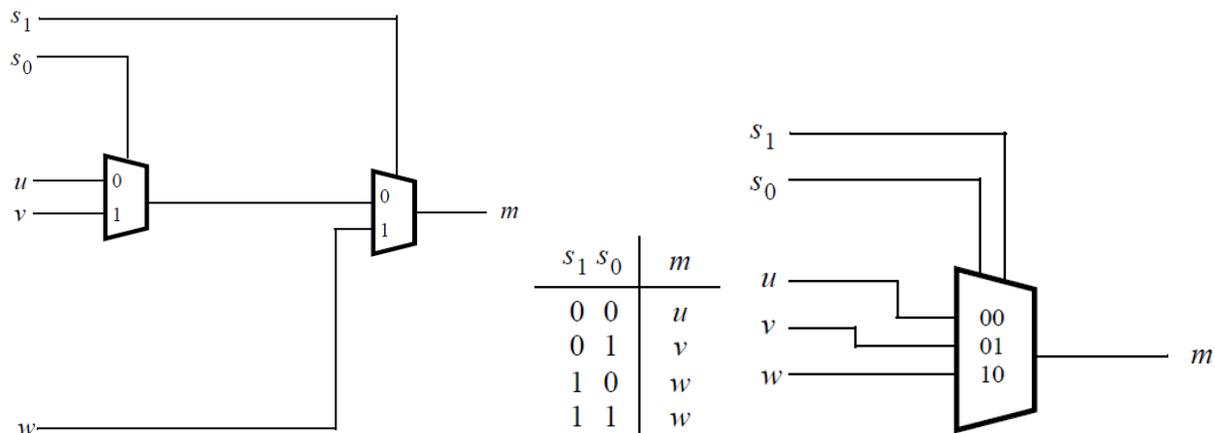
Mixed-design Style Modeling

Part 5

Complex systems can be described in VHDL using mixed-design style modeling. This modeling style supports hierarchical description. The design can be described using:

- Dataflow modeling (covered in Part 2),
- Structural modeling (covered in Part 3),
- Behavioral modeling (covered in Part 4),
- and combinations of the above.

As an example of mixed style modeling, one can build 3-to-1 multiplexer using multiple instances of 2-to-1 multiplexer.



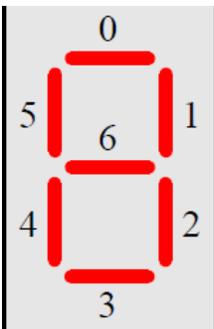
In the above diagram, u , v , w are data inputs whereas S_0 , S_1 are select signals, and the output is m . It uses two instances of 2-to-1 multiplexer. The truth table and the top-level symbol are as provided.

5-1. Model a 3-to-1 multiplexer using 2-to-1 multiplexers.

- 5-1-1. Open PlanAhead and create a blank project called lab1_5_1.
- 5-1-2. Create a top-level VHDL module with three data inputs (u, y, w), two select inputs ($s0, s1$), and one bit output (m) using the previously defined 2-to-1 multiplexer. You can use any style designed 2-to-1 multiplexer (1-1, 2-1, or 3-1). Wire them up as shown in the above diagram.
- 5-1-3. Add the used 2-to-1 model file to the project.
- 5-1-4. Create and add the UCF file to the project. Assign **SW0**, **SW1**, and **SW2** to $u, y,$ and w respectively, **SW3** to $s0$, **SW4** to $s1$, and **LEDO** to m .
- 5-1-5. Synthesize and implement the design.
- 5-1-6. Generate bitstream, download it to the Nexys3 board, and verify the functionality.

5-2. Model a BCD to 7-Segment Decoder.

A 7-segment display consists of seven segments, numbered 0 to 6 or a to g which can be used to display a character. Depending on the input type, a type conversion may be needed. If you want to display a binary coded decimal (BCD) using 4-bit input, a BCD to 7-segment decoder is required. The table below shows the bit pattern you need to put to display a digit (note that to turn ON a segment you need to put logic 0).

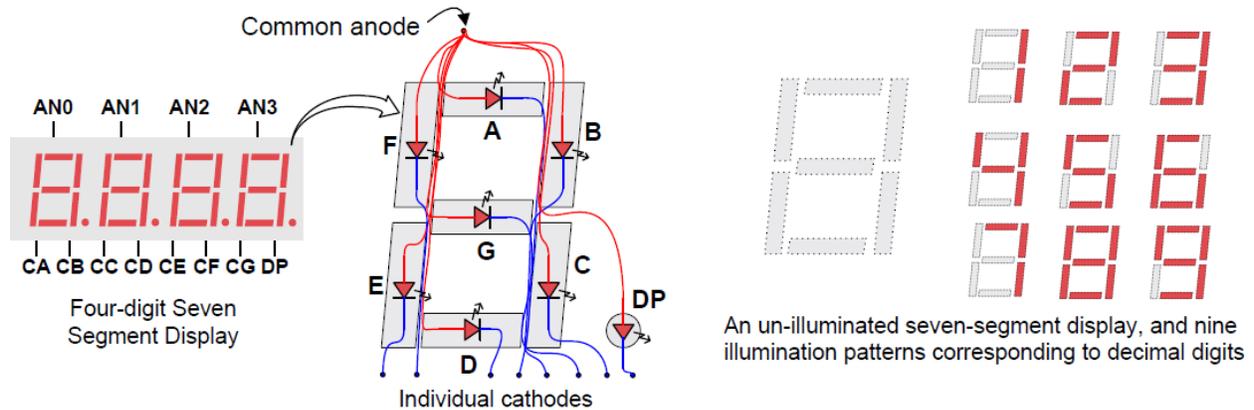


Input	0 or a	1 or b	2 or c	3 or d	4 or e	5 or f	6 or g
0000	0	0	0	0	0	0	1
0001	1	0	0	1	1	1	1
0010	0	0	1	0	0	1	0
0011	0	0	0	0	1	1	0
0100	1	0	0	1	1	0	0
0101	0	1	0	0	1	0	0
0110	0	1	0	0	0	0	0
0111	0	0	0	1	1	1	1
1000	0	0	0	0	0	0	0

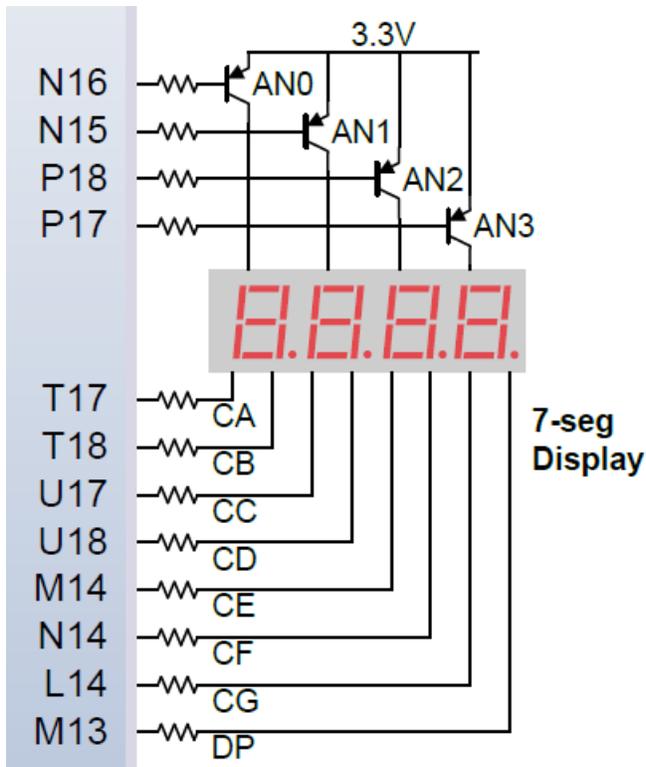
1001	0	0	0	0	1	0	0
1010 to 1111	X	X	X	X	X	X	x

Where x is don't care.

The Nexys3 board contains a four-digit common anode seven-segment LED display. Each of the four digits is composed of seven segments arranged in a pattern shown below, with an LED embedded in each segment. Segment LEDs can be individually illuminated, so any one of 128 patterns can be displayed on a digit by illuminating certain LED segments and leaving the others dark. Of these 128 possible patterns, the ten corresponding to the decimal digits are the most useful.



The anodes of the seven LEDs forming each digit are tied together into one “common anode” circuit node, but the LED cathodes remain separate. The common anode signals are available as four “digit enable” input signals to the 4-digit display. The cathodes of similar segments on all four displays are connected into seven circuit nodes labeled CA through CG (so, for example, the four “D” cathodes from the four digits are grouped together into a single circuit node called “CD”). These seven cathode signals are available as inputs to the 4-digit display. This signal connection scheme creates a multiplexed display, where the cathode signals are common to all digits but they can only illuminate the segments of the digit whose corresponding anode signal is asserted.



A scanning display controller circuit can be used to show a four-digit number on this display. This circuit drives the anode signals and corresponding cathode patterns of each digit in a repeating, continuous succession, at an update rate that is faster than the human eye can detect. Each digit is illuminated just one-quarter of the time, but because the eye cannot perceive the darkening of a digit before it is illuminated again, the digit appears continuously illuminated. If the update or “refresh” rate is slowed to around 45 hertz, most people will begin to see the display flicker. You will design and use the scanning circuit starting with Lab7 (Architecture Wizard and CoreGEN).

- 5-2-1. Open PlanAhead and create a blank project called lab1_5_2.
- 5-2-2. Create a top-level VHDL module, named `bcdto7segment_dataflow` with 4-bit data input (`x[3:0]`), anode enable output signals (`an[3:0]`), and 7-bit output (`seg[6:0]`) using dataflow modeling (Hint: You will have to derive seven expressions for the 7 segments on paper). Assign appropriate logic to `an[3:0]` in the model so you can display only on the right most display..
- 5-2-3. Create and add the XDC file to the project. Assign **SW3-SW0** to `x[3:0]`. Assign **SEG0** to `seg[6:0]`, and pins **P17, P18, N15, N16** to `an3, an2, an1, an0`. Refer to the board’s user’s manual and master XDC file provided in the sources directory.
- 5-2-4. Synthesize the design.
- 5-2-5. Implement the design.
- 5-2-6. Generate bitstream, download it in to the Nexys3 board, and verify the functionality.

Conclusion

In this lab, you learned three types of modeling. You created PlanAhead projects to develop various models. You implemented the design and verified the functionality.