Modeling Concepts

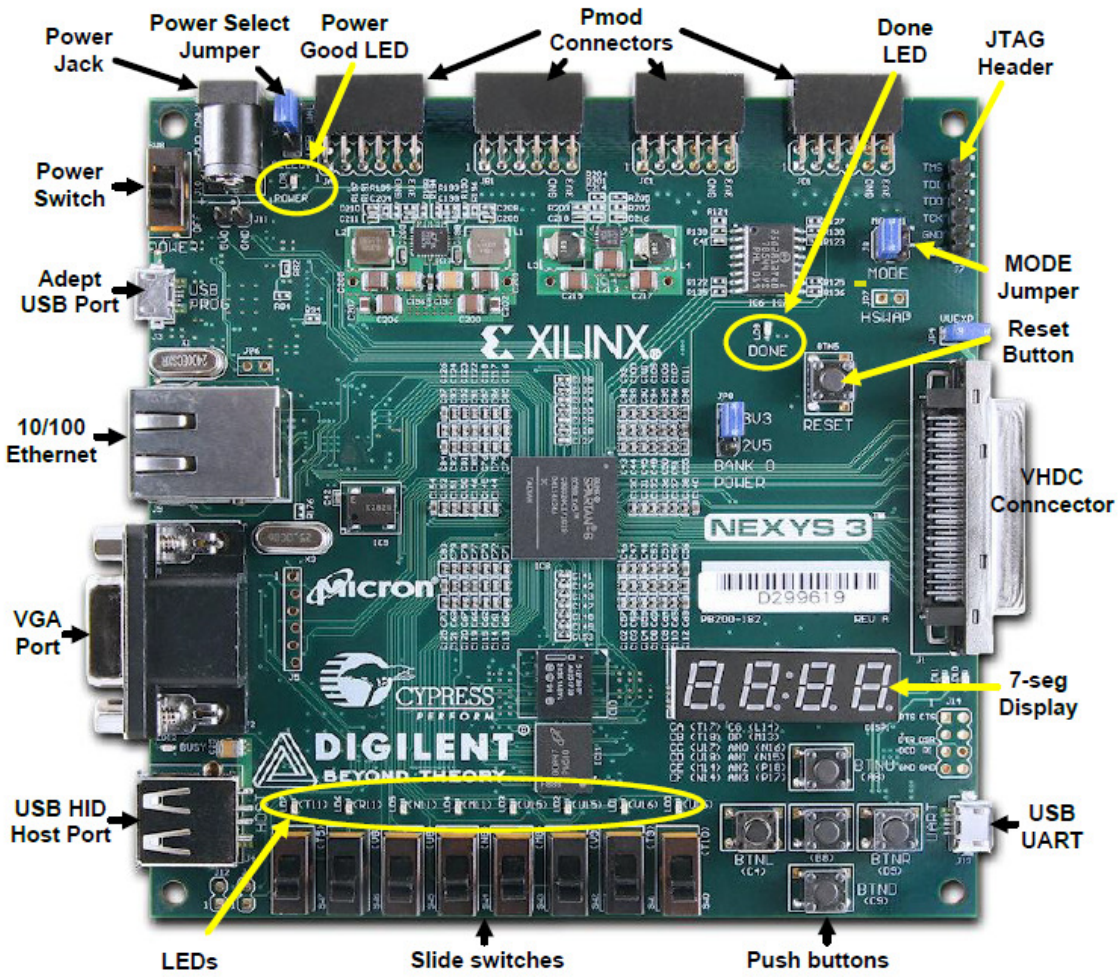
Introduction

Verilog HDL modeling language supports three kinds of modeling styles: gate-level, dataflow, and behavioral. The gate-level and datafow modeling are used to model combinatorial circuits whereas the 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 features:

- 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 gate-level, dataflow, 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

Gate-level Modeling Part 1

Verilog HDL supports built-in primitive gates modeling. The gates supported are multiple-input, multiple-output, tristate, and pull gates. The multiple-input gates supported are: **`and`**, **`nand`**, **`or`**, **`nor`**, **`xor`**, and **`xnor`** whose number of inputs are two or more, and has only one output. The multiple-output gates supported are **`buf`** and **`not`** whose number of output is one or more, and has only one input. The language also supports modeling of tri-state gates which include **`bufif0`**, **`bufif1`**, **`notif0`**, and **`notif1`**. These gates have one input, one control signal, and one output. The pull gates supported are **`pullup`** and **`pulldown`** with a single output (no input) only.

The basic syntax for each type of gates with zero delays is as follows:

`and` | **`nand`** | **`or`** | **`nor`** | **`xor`** | **`xnor`** [instance name] (out, in1, …, inN); // [] is optional and | is selection
`buf` | **`not`** [instance name] (out1, out2, …, out2, input);
`bufif0` | **`bufif1`** | **`notif0`** | **`notif1`** [instance name] (outputA, inputB, controlC);
`pullup` | **`pulldown`** [instance name] (output A);

One can also have multiple instances of the same type of gate in one construct separated by a comma such as

`and` [inst1] (out11, in11, in12), [inst2] (out21, in21, in22, in23), [inst3] (out31, in31, in32, in33);

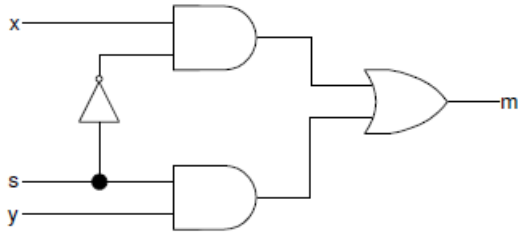
The language also allows the delays to be expressed when instantiating gates. The delay expressed is from input to output. The delays can be expressed in form of rise, fall, and turn-off delays; one, two, or all three types of delays can be expressed in a given instance expression. The turn-off delay is applicable to gates whose output can be turned OFF(.e.g. notif1).

For example,

```
and #5 A1(Out, in1, in2);       // the rise and fall delays are 5 units
and #(2,5) A2(out2, in1, in2);        // the rise delay is 2  unit and the fall delay is 5 units
notif1 #(2, 5, 4) A3(out3, in2, ctrl1); //the rise delay is 2, the fall delay is 5, and the turn-off delay is 4 unit
```

The gate-level modeling is useful when a circuit is a simple combinational, as an example a multiplexer. Multiplexer is a simple circuit which connects one of many inputs to an output. In this part, you will create a simple 2-to-1 multiplexer and extend the design to multiple bits.

1-1. Create a 2-to-1 multiplexer using gate-level modeling.



1-1-1. Open PlanAhead and create a blank project project called lab1_1_1 (refer Step 1 of the PlanAhead Tutorial)**.**

1-1-2. Create and add the Verilog module with three inputs (*x*, *y*, *s*) and one output (*m*) using gate-level modeling (refer Step 1 of the PlanAhead Tutorial).

1-1-3. 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).

1-1-4. Synthesize the design (refer Step 4 of the PlanAhead Tutorial).

1-1-5. Implement the design (refer Step 5 of the PlanAhead Tutorial)..

1-1-6. Generate the bitstream, download it into the Nexys3 board, and verify the functionality (refer Step 6 of the PlanAhead Tutorial for steps involved in creating and downloading the bitstream).

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

1-2-1. Open PlanAhead and create a blank project project called lab1_1_2**.**

1-2-2. Create and add the Verilog 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 gate-level modeling.

1-2-3. 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].*

1-2-4. Synthesize the design.

1-2-5. Implement the design.

1-2-6. Generate the bitstream, download it into the Nexys3 board, and verify the functionality.

Dataflow Modeling Part 2

Dataflow modeling style is mainly used to describe combinational circuits. The basic mechanism used is the continuous assignment. In a continuous assignment, a value is assigned to a data type called **`net`**. The syntax of a continuous assignment is

`assign` [delay] LHS_net = RHS_expression;

Where LHS_net is a destination net of one or more bit, and RHS_expression is an expression consisting of various operators. 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. The gate level modeling examples listed in Part 1 can be described in dataflow modeling using the continuous assignment. For example,

```
assign out1 = in1 & in2; // perform and function on in1 and in2 and assign the result to out1
assign out2 = not in1;
assign #2 z[0] = ~(ABAR & BBAR & EN); // perform the desired function and assign the result after 2 units
```

The target in the continuous assignment expression can be one of the following:
1. A scalar net (e.g. 1st and 2nd examples above)
2. Vector net
3. Constant bit-select of a vector (e.g. 3rd example above)
4. Constant part-select of a vector
5. Concatenation of any of the above

Let us take another set of examples in which a scalar and vector nets are declared and used

```
wire COUNT, CIN;   // scalar net declaration
wire [3:0] SUM, A, B;    // vector nets declaration
assign {COUT,SUM} = A + B + CIN; // A and B vectors are added with CIN and the result is
                                 // assigned to a concatenated vector of a scalar and vector nets
```

Note that multiple continuous assignment statements are not allowed on the same destination net.

2-1. Connect input switches to output LEDs using dataflow modeling.

2-1-1. Open PlanAhead and create a blank project project called lab1_2_1**.**

2-1-2. Create and add the Verilog module with 8 inputs (*x_in*) and 8 outputs (*y_out*) using dataflow modeling. Use vector assignment statement as we do not need to do any processing between SW inputs.

2-1-3. Create and add the UCF file to the project. Assign ***SW7-SW0*** to *x_in* and ***LED7-LED0*** to *y_out*.

2-1-4. Synthesize the design.

2-1-5. Implement the design.

2-1-6. Generate the bitstream, download it into the Nexys3 board, and verify the functionality.

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

2-2-1. Open PlanAhead and create a blank project project called lab1_2_2**.**

2-2-2. Create and add the Verilog 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. Each assignment statement should have 3 units delay.

As an example, a one-bit 2-to-1 multiplexer can be described as follows:

```
assign #3 m = (~s & x) | (s & y); // 3 units delay
```

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. Add the provided testbench (mux_2bit_2_to_1_dataflow_tb.v) to the project.

2-2-5. Simulate the design for 100 ns and analyze the output.

2-2-6. Synthesize the design.

2-2-7. Implement the design.

2-2-8. Generate the bitstream, download it into the Nexys3 board, and verify the functionality.

Behavioral Modeling Part 3

Behavioral modeling is used to describe complex circuits. It is primarily used to model sequential circuits, but can also be used to model pure combinatorial circuits. The mechanisms (statements) for modeling the behavior of a design are:
`initial` Statements
`always` Statements

A module may contain an arbitrary number of **`initial`** or **`always`** statements and may contain one or more procedural statements within them. They are executed concurrently (i.e. to model parallelism such that the order in which statements appear in the model does not matter) with respect to each other whereas the procedural statements are executed sequentially (i.e. the order in which they appear does matter). Both **`initial`** and **`always`** statements are executed at time=0 and then only **`always`** statements are executed during the rest of the time. The syntax is as follows:

```
initial [timing_control] procedural_statements;
always [timing_control] procedural_statements;
```

where a procedural_statement is one of:
procedural assignment
conditional_statement
case_statement
loop_statement
wait_statement

The **`initial`** statement is non-synthesizable and is normally used in testbenches. The **always** statement is synthesizable, and the resulting circuit can be a combinatorial or sequential circuit. In order for the model to generate a combinatorial circuit, the **`always`** block (i) should not be edge sensitive, (ii) every branch of the conditional statement should define all output, and (iii) every case of case statement should define all output and must have a default case. More detailed coverage of this topic is covered in Lab 7. The destination (LHS) should be of **`reg`** type; either scalar or vector. For example,

```
reg m; // scalar reg type
reg [7:0] switches; // vector reg type
```

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

```
reg m;

always @ (x or y or s)
begin
      if(s==0)
            m=y;
      else
            m=x;
end
```

3-1. Create a 2-to-1 multiplexer using behavioral modeling.

3-1-1. Open PlanAhead and create a blank project project called lab1_3_1**.**

3-1-2. Create and add the Verilog module with three inputs (*x*, *y*, *s*) and one output (*m*) using behavioral modeling. Use the example code given above.

3-1-3. Add the UCF file, created in 1-1, to the project. Assign ***SW0*** and ***SW1*** to *x* and *y*, ***SW7*** to *s*, and ***LED0*** to *m*.

3-1-4. Synthesize the design.

3-1-5. Implement the design.

3-1-6. Generate the bitstream, download it into the Nexys3 board, and verify the functionality.

3-2. Create a two-bit wide 2-to-1 multiplexer using behavioral modeling.

3-2-1. Open PlanAhead and create a blank project project called lab1_3_2**.**

3-2-2. Create and add the Verilog 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.

3-2-3. 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]*.

3-2-4. Synthesize the design.

3-2-5. Implement the design.

3-2-6. Generate the bitstream, download it into the Nexys3 board, and verify the functionality.

Mixed-design Style Modeling Part 4

Complex systems can be described in Verilog HDL using mixed-design style modeling. This modeling style supports hierarchical description. The design can be described using:
Build-in gate primitives (gate-level modeling covered in Part 1)
Dataflow modeling (covered in Part 2)
Behavioral modeling (covered in Part 3)

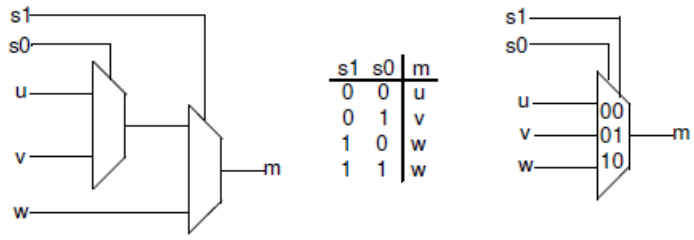
Module instantiation
Combinations of the above.

Interconnections between various objects are done through nets (of type ***wire***). Nets may be scalar or vector. For example,

```
wire y; // scalar net
wire [3:0] sum; // vector net
```

In absence of size, the net is assumed to be the scalar type.

As an example of a mixed-style modeling, following diagram shows how one can build a 3-to1 multiplexer using multiple instances of 2-to-1 multiplexer. It also shows the symbol and the truth table.



In the above diagram, u, v, w are data inputs whereas S0, S1 are select signals, and the output is m. It uses two instances of 2-to-1 multiplexer.

4-1. Model a 3-to-1 multiplexer using 2-to-1 multiplexers.

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

4-1-2. Create a top-level Verilog 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.

4-1-3. Add the used 2-to-1 model file to the project.

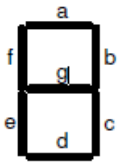
4-1-4. Create and add the UCF file to the project. Assign ***SW0, SW1***, and ***SW2*** to *u, y,*and *w* respectively, ***SW3*** to *s0*, ***SW4*** to *s1*, and ***LED0*** to *m*.

4-1-5. Synthesize and implement the design.

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

4-2. Model a BCD to 7-Segment Decoder.

A 7-segment display consists of seven segments, numbered a to g which can be used to display a character. Depending on the input type, a type conversion may be needed. If 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 on the segment and the anode of the display needs to be driven logic 0 on this board).

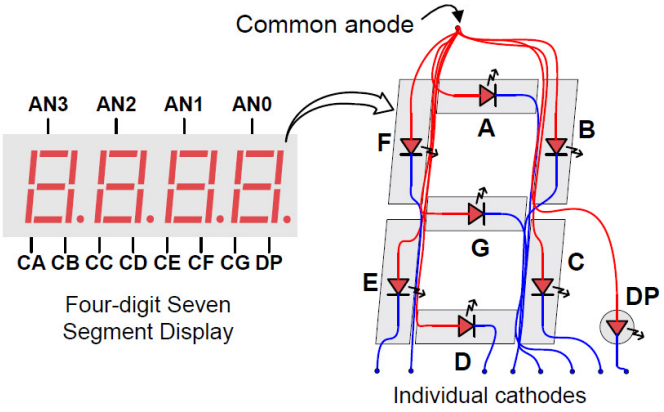


.

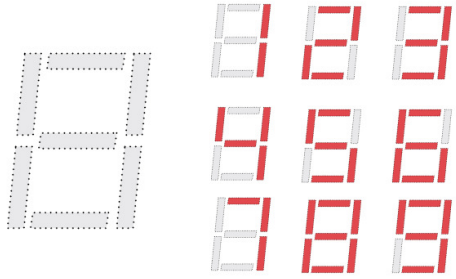
Input	a	b	c	d	e	f	g
0000	0	0	0	0	0	0	1
0001	1	0	0	1	1	1	1
0010	0	0	1	0	0	1	0
0011	0	0	0	0	1	1	0
0100	1	0	0	1	1	0	0
0101	0	1	0	0	1	0	0
0110	0	1	0	0	0	0	0
0111	0	0	0	1	1	1	1
1000	0	0	0	0	0	0	0
1001	0	0	0	0	1	0	0
1010 to 1111	X	X	X	X	X	X	x

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.



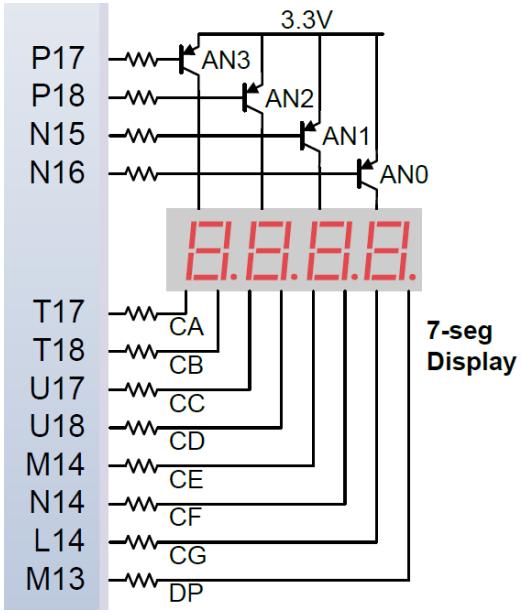
Four-digit Seven Segment Display



An un-illuminated seven-segment display, and nine illumination patterns corresponding to decimal digits

[Reference – Nexys3 Reference Manual]

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.



[Reference – Nexys3 Reference Manual]

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. 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 Lab 8 (Architecture Wizard and CoreGEN).

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

4-2-2. Create a top-level Verilog 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..

4-2-3. Create and add the UCF 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 ucf file provided in the sources directory.

4-2-4. Synthesize the design.

4-2-5. Implement the design.

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

Conclusion

In this lab, you created PlanAhead projects to develop various models. You implemented the design and verified the functionality in hardware as well as simulation. You learned three modeling styles. The gate-level and dataflow modeling are primarily used for the combinatorial circuits, whereas the behavioral modeling supports both combinatorial and sequential circuits design. In this lab you used the behavioral modeling for the combinatorial circuits design. In next few labs you will be using dataflow modeling for designing various combinatorial circuits, and starting with Lab 7, you will use the behavioral modeling to design sequential circuits.