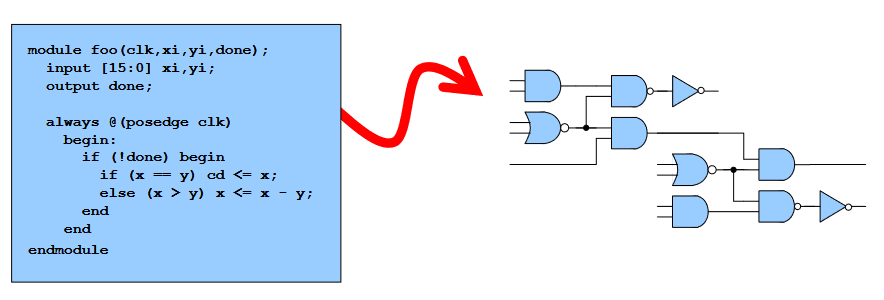
**ECE 3331 Laboratory Notes  
Henry Johnston**

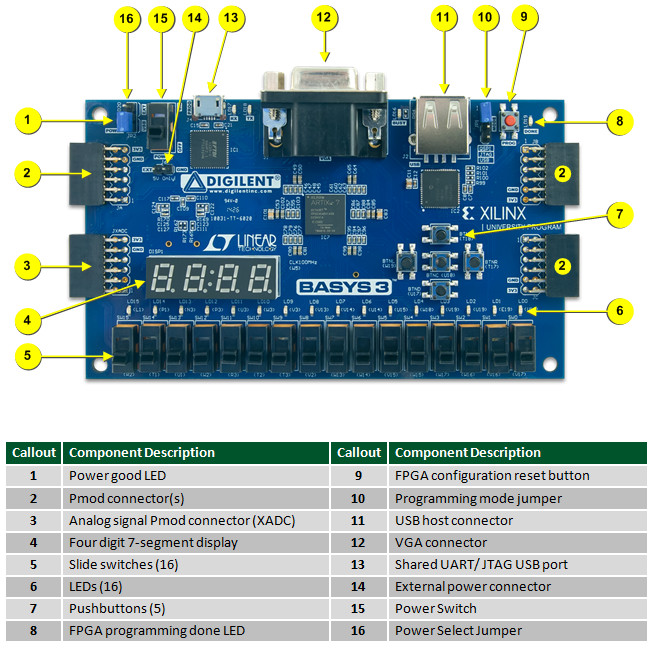
**Part I: Introduction to FPGA and Verilog**

In ECE3331, students will be required to create Verilog code for a field programmable gate array (FPGA). The FPGA is an [integrated circuit](https://en.wikipedia.org/wiki/Integrated_circuit) designed to be configured by the user. FPGAs contain an array of [programmable](https://en.wikipedia.org/wiki/Programmable_logic_device) [logic blocks](https://en.wikipedia.org/wiki/Logic_block) and reconfigurable interconnects that allow the blocks to be wired together. [Logic blocks](https://en.wikipedia.org/wiki/Logic_block) can be configured to implement simple [combinational logic circuits](https://en.wikipedia.org/wiki/Combinational_logic) (e.g. multiplexers, decoders and comparators) to far more complex sequential circuits like microprocessors. FPGAs are used to interact with peripheral such as switches, LEDs, servomotors, VGA ports, keyboards, monitors, etc.

The desired configuration of an FPGA is specified using a [hardware description language](https://en.wikipedia.org/wiki/Hardware_description_language) (HDL). Hardware description languages are used to make higher-level descriptions of the behavior of digital circuits. A higher-level description is converted into a netlist: description of connectivity in a circuit with an optimized number of logic gates. The process of converting a higher-level description of a circuit’s behavior into a netlist is called **logic synthesis**. The netlist is then converted into a bitstream (a sequence of bits) that is loaded to the FPGA. **Programming an FPGA is the process of loading the bitstream into the FPGA**. Bitstreams usually have the .bit file extension.

  
A higher-level description of a digital circuit is converted to actual hardware through a process called **logic synthesis**. Figure source: MIT OCW.

The two most popular hardware descriptions languages in industry and academia are VHDL and Verilog. VHDL was originally created by the United States Department of Defense and is widely used in the military sector. Verilog is more popular in academia and commercial industry and it is widely used because of its similarity to the C programming language. For ECE3331, Vivado will be used to write and synthesize Verilog code and to generate a bitstream for the Artix-7 FPGA of the Basys3 board. In ECE3331, students have to program the FPGA to interact with switches, pushbuttons, LEDs, servomotors, sensors and 7-segment displays among other peripherals.

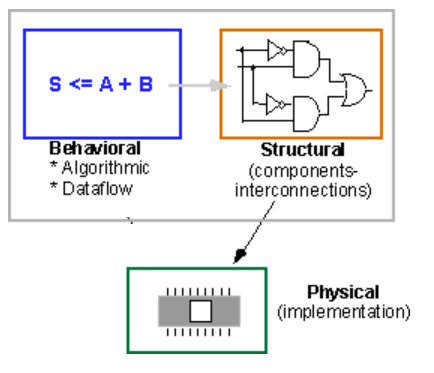
  
The Artix-7 FPGA is seen in the middle of the Basys3 board. The Artix-7 has its own on-chip analog-to-digital signal converter (XADC). The Basys3 board includes 1 USB-JTAG port for FPGA programming, 16 LEDs, 16 switches, 5 pushbuttons, Pmod connectors, 1 four-digit seven-segment display, 1 VGA port (for image and video projects) among other peripherals.

Verilog has a syntax similar to the [C programming language](https://en.wikipedia.org/wiki/C_(programming_language)). However, it is important to emphasize that Verilog is a **hardware description language.** The C programming language describes a set of actions to be done sequentially by the CPU. C is a **software description language**. In other words, C is good for telling a CPU what to do while Verilog is good for designing and implementing digital circuits that can perform a desired set of actions.

There are different ways to design a digital circuit using Verilog: structural (gate-level) modeling, behavioral modeling and dataflow modeling:

* **Structural or gate-level modeling** describes the structure within a module by explicitly describing its gates and how they connect with one another as well as to the input/output ports of the module. Structural modeling is a textual replacement for a schematic.
* **Behavioral modeling** is a higher-level description of a digital circuit. Behavioral modeling is used to describe the function of a digital circuit in an algorithmic manner. Verilog has conditional statements (if-else statement, case statement) and control flow statements (for-loops, while-loops) similar to the C programming language. These statements can be used to describe the behavior of a circuit instead of designing the circuit itself. The synthesis tool in the Vivado software maps a behavioral model to a physical circuit with logic gates.
* **Dataflow modeling** describes the flow of data between input and output. It is used for implementation of the Boolean functions of combinational logic. Bit-wise operators (~, &, |, ^, ~^) are used for describing dataflow models of combinational logic. The synthesis tool in the Vivado software maps a dataflow model to a physical circuit with logic gates.

Behavioral modeling and dataflow modeling are different from structural or gate-level modeling. They allow designers to make a description of what the hardware should do without designing the hardware itself. The process of automatically generating a gate-level model from either a dataflow or a behavioral model is called **logic synthesis**.

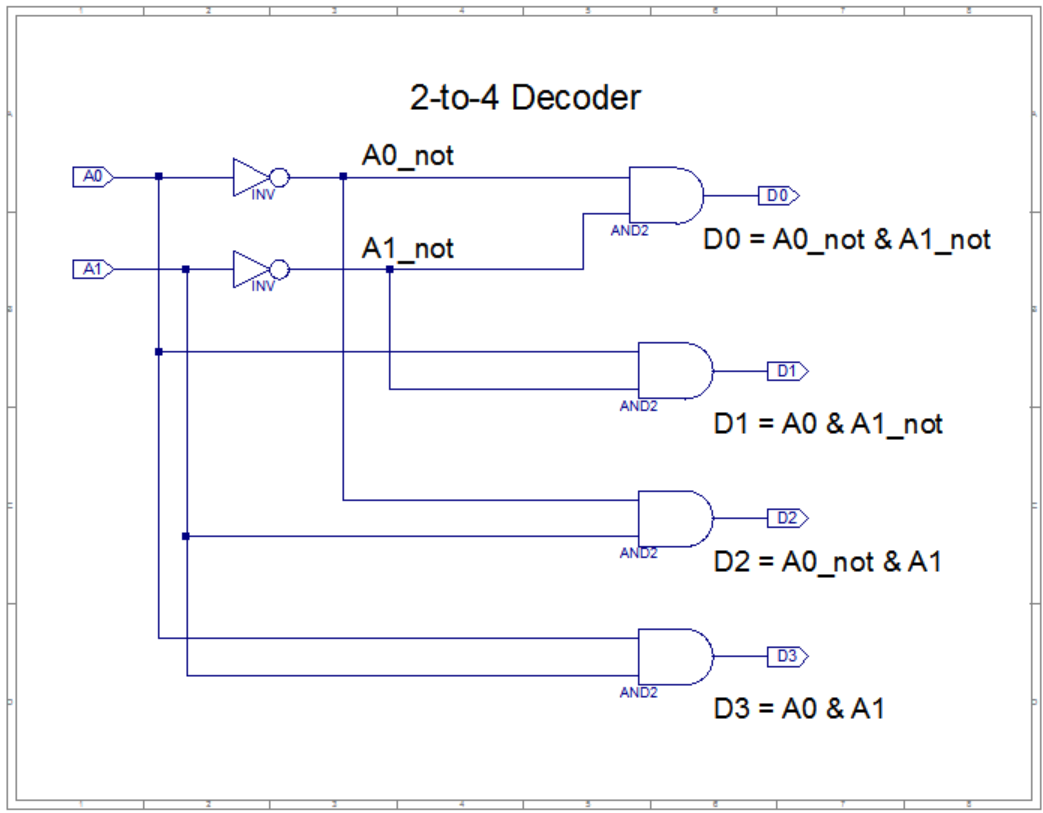
  
Logic synthesis and FPGA implementation.   
Figure source: Jan Van der Spiegel, University of Pennsylvania.

**Implementation of combinational circuits with Verilog**

Combinational logic circuits are memoryless circuits. Their outputs depend only on the present inputs and not on the past states of the outputs. Decoders, multiplexers, comparators, priority encoders, adders, subtractors and multipliers are examples of combinational logic circuits. There are more combinational logic circuits than the previously mentioned, but only Verilog code for the decoder, multiplexer, comparator and priority encoder will be shown.

1. **2-to-4 Decoder:**

A decoder is a multiple-input, multiple-output combinational logic circuit that converts coded inputs into coded outputs. The ratio of input to output codes is n-to-2n. Decoders are necessary in applications such as data multiplexing and 7-segment displays.

  
Figure 1.1: 2-to-4 decoder schematic. The ampersand (&) stands for bit-wise AND.

![](data:None;base64,iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAYAAAAfFcSJAAAABHNCSVQICAgIfAhkiAAAAAtJREFUCJljYAACAAAFAAFiVTKIAAAAAElFTkSuQmCC)  
Figure 1.2: 2-to-4 decoder truth table.

The structural or gate-level model of the decoder will be shown first, followed by the behavioral and dataflow models. The structural model (example 1.1) implements the decoder using NOT and AND gates as shown in figure 1.1. The behavioral model (example 1.2) implements the decoder using conditional statements (if-else statements). The dataflow model (example 1.3) implements the decoder in terms of the Boolean functions shown next to the output of the AND gates in figure 1.1.

Finally, example 1.4 shows the test bench used to simulate the structural model of the decoder. To simulate a design, you need both the **unit under test** (the file for the structural model of the decoder) and to provide a **stimulus** (input) in the test bench. A test bench is code that allows you to provide a set of stimuli to the design. Notice that only the test bench and simulation of the first decoder is shown. All decoders (structural, behavioral and dataflow) should exhibit the exact same behavior when given the same set of inputs.  
// -------------------------------------------------------------// --------- Example 1.1: Decoder Structural Model -------------  
// -------------------------------------------------------------

// Structural or gate-level model. Module name is “decoder2to4” followed by the ports, port directions and port names

**module dec2\_4\_gate**( **input** A0,  **input** A1,  **output** D0,  **output** D1,  **output** D2,  **output** D3);

// The wires are the outputs of the NOT gates shown in figure

**wire** A0\_not; **wire** A1\_not;

**not**(A0\_not, A0); // Logic gate primitives. Output is always the first terminal

**not**(A1\_not, A1); **and**(D0, A0\_not, A1\_not); **and**(D1, A0, A1\_not); **and**(D2, A0\_not, A1); **and**(D3, A0, A1);

**endmodule**

// -------------------------------------------------------------// ------------------- Example 1.1: End ------------------------  
// -------------------------------------------------------------

A Verilog program always begins with the word **module** and ends with the word **endmodule**. The initial word **module** is followed by the **name of the module** chosen by the programmer and the **module’s ports** and **port directions** (input, output or inout) inside parenthesis. The **module’s ports** are the pins of the hardware component. Verilog offers **primitives** for the basic logic gates. These **primitives** must have one output and at least one input. The first terminal is the output and the following terminals are the inputs. Verilog also offers a **net type** called **wire**. A **wire** is used for connecting different elements and to transmit data (bits) from one element to the other. They do not store the data. The **wire** net type can be seen as a physical wire. Verilog files have the .v file extension.

A 2-to-4 decoder in Verilog can also be generated by a behavioral model as mentioned before. In behavioral modeling, a module can be thought of as a black box. It is not necessary to know the logic gates inside the box. However, it is necessary to know the outputs for a given set of inputs. There is more than one algorithm for creating a behavioral model of the decoder in Verilog. One of these algorithms uses the truth table (figure 1.2) and if-else statements as shown below:

// -------------------------------------------------------------// ----- Example 1.2.1: Decoder Behavioral Model (v.1) ---------  
// -------------------------------------------------------------

// Behavioral model. Module name is "dec2\_4\_behavioral" followed by the ports, port directions and port names  
**module** **dec2\_4\_behavioral**( **input** A0,   
 **input** A1,   
 **output reg** D0,   
 **output reg** D1,   
 **output reg** D2,   
 **output reg** D3);

// Whenever there is a change in A0 or A1, execute the always block  
**always @**(A0 or A1) **begin**  
  
**if** ({A1,A0}==2'b00) // Concatenation operator "{}" used to combine signals into a   
{D3,D2,D1,D0} = 4'b0001; // single group. It makes code more readable  
  
**else if** ({A1,A0}==2'b01)   
{D3,D2,D1,D0} = 4'b0010;

**else if** ({A1,A0}==2'b10)  
{D3,D2,D1,D0} = 4'b0100;

**else**   
{D3,D2,D1,D0} = 4'b1000;

**end** // End the always block  
 **endmodule**

// -------------------------------------------------------------// ------------------ Example 1.2.1: End -----------------------  
// -------------------------------------------------------------

There are several differences in the behavioral model shown in example as compared to the structural model from example 1:

* The output ports are declared as **reg** (register). In Verilog, **reg** represents a data storage element. It is generally used to model hardware registers (although it can also represent combinational logic). The **reg** variables retain their value until the next value is assigned to them. The outputs of behavioral models are usually declared as **reg** variables.
* The second difference you will notice is the presence of the **always** block. The actions inside the block will be triggered every time there is a change in the sensitivity list of the block (variables inside of the parenthesis next to the “**@**” symbol).
* Finally, you will notice that the inputs and the outputs are concatenated. Concatenation is the combination of different signals into a single group. In the case of the output, the binary values for 1, 2, 4 and 8 are stored in the group {D3,D2,D1,D0} depending on the value of the group {A1,A0}. The default numeral system in Verilog is the decimal system but the binary system is used in example 2. The values 4’b0001 (four-bit binary representation for 1 in Verilog), 4’b0010 (four-bit binary representation for 2), 4’b0100 (four-bit binary representation for 4) and 4’b1000 (four-bit binary representation for 8) are the outputs of the decoder from the right-side of the truth table in figure 1.2.

It is also important to mention that the case equality operator (==) is different from the equal symbol (=). The case equality operator is used to make a comparison between two variables while the equal symbol is used to assign a value to a variable or describe combinational logic.

// -------------------------------------------------------------

// ----------- Example 1.3: Decoder Dataflow Model -------------  
// -------------------------------------------------------------

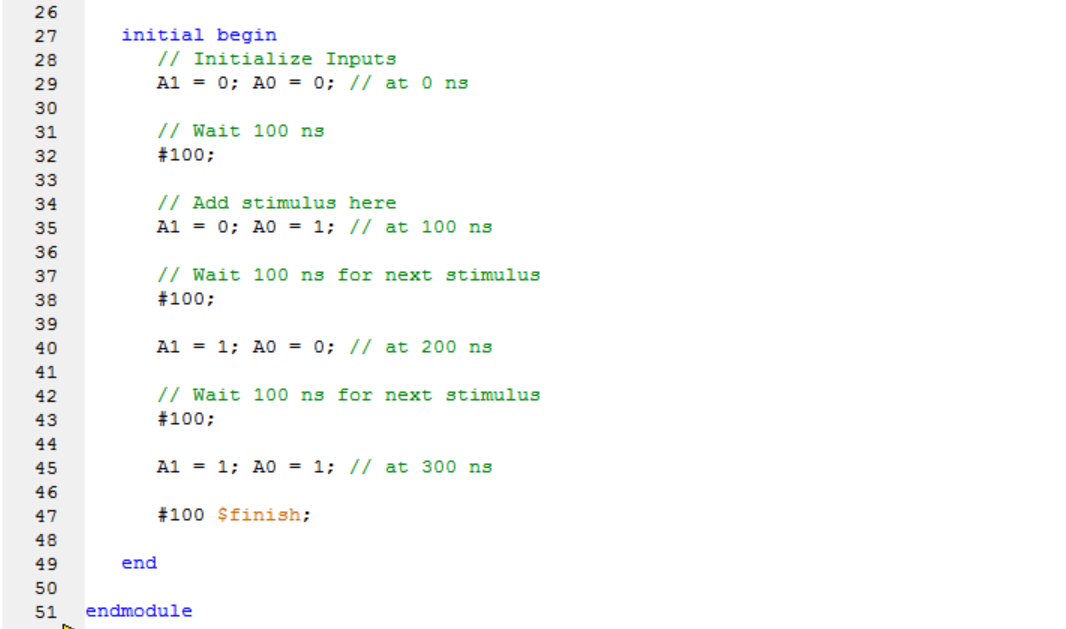
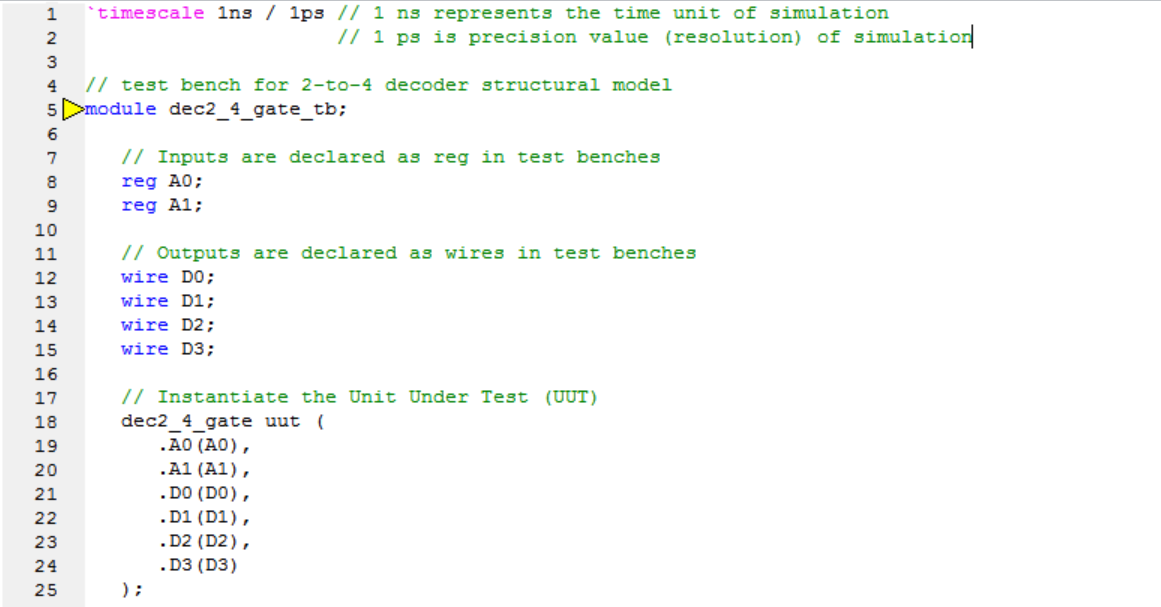
// Dataflow model. Module name is "dec2\_4\_df" followed by the ports, port directions and port names **module dec2\_4\_df**( **input** A0,   
 **input** A1,   
 **output** D0,   
 **output** D1,   
 **output** D2,   
 **output** D3);

// "assign" is used to assign a Boolean function to each output. Symbol “~” stands for binary bit-wise NOT. Symbol “&” stands for binary bit-wise AND  
**assign** D0 = ~A0 & ~A1; // Boolean function for D0  
**assign** D1 = A0 & ~A1; // Boolean function for D1  
**assign** D2 = ~A0 & A1; // Boolean function for D2  
**assign** D3 = A0 & A1; // Boolean function for D3  
  
**endmodule**

// -------------------------------------------------------------// ------------------- Example 1.3: End ------------------------  
// -------------------------------------------------------------

The dataflow model of the 2-to-4 decoder describes the circuit in terms of the Boolean functions shown in figure 1.1 (next to the outputs). The **assign** statement in Verilog is used to model combinational logic. The signals in the assign statement are continuously monitored for changes. Finally, binary bit-wise operators were used in the Boolean functions: the tilde symbol “~” for binary bit-wise NOT and the ampersand symbol “&” for binary bit-wise AND. Page [] contains a table with these symbols.

Finally, the test bench is shown in example 1.4. The test bench specifies the **unit under test** (decoder structural model), the **stimuli** (inputs) and the **time periods** when these stimuli are applied. In this set of notes, you will see different ways of applying a stimulus to a design. Test benches are not synthesizable and they cannot be loaded to the FPGA. They are only useful for testing a design.

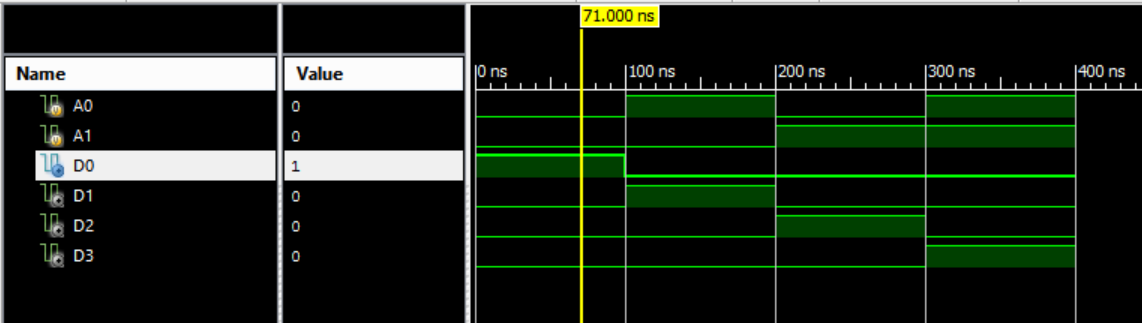
// -------------------------------------------------------------  
// ------ Example 1.4: Decoder Structural Model Test Bench -----  
// -------------------------------------------------------------  
  


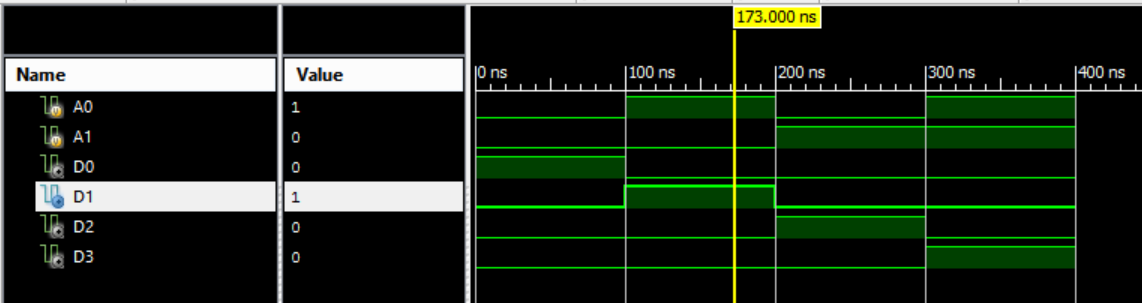
// -------------------------------------------------------------// ------------------- Example 1.4: End ------------------------  
// -------------------------------------------------------------

The first task in the test bench is to specify the time unit and resolution of the simulation. In example 1.4, the time unit is nanosecond (ns) and the resolution is picosecond (ps). Inputs are declared as registers while outputs are declared as wires.

The next step is to **instantiate** the **unit under test** (the decoder structural model for this particular simulation). Finally, the test bench applies a set of stimuli. The stimuli are just the inputs from the truth table in figure 1.2. A time period is assigned for the application of each stimulus since they cannot be applied at the same time. The simulation is terminated at 400 ns with the statement **$finish**. A Verilog test bench has the .v file extension. The test bench file name usually includes the words “test\_bench” or “tb” to avoid confusion with the **unit under test**.

**2-to-4 decoder simulation results in ISim:**

  
Figure 1.3: Output of 2-to-4 decoder structural model in ISim. The stimulus is A1 = 0 and A0 = 0. The time interval is 0 ns - 100 ns. Output is D3 = 0, D2 = 0, D1 = 0 and D0 = 1.

  
Figure 1.4: Output of 2-to-4 decoder structural model in ISim. The stimulus is A1 = 0 and A0 = 1. The time interval is 100 ns - 200 ns. Output is D3 = 0, D2 = 0, D1 = 1 and D0 = 0.

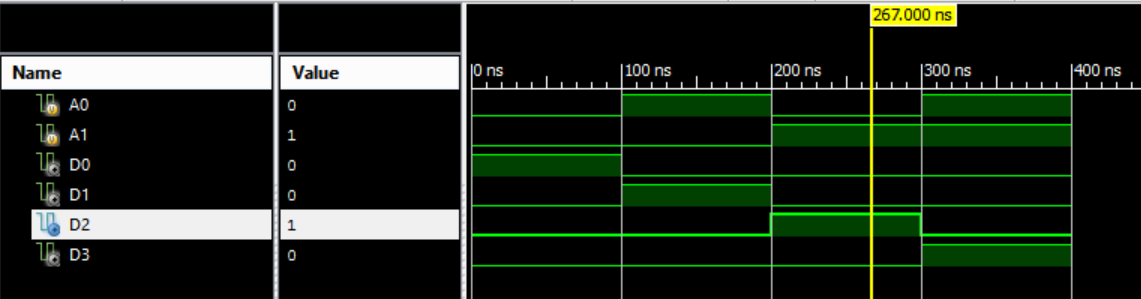
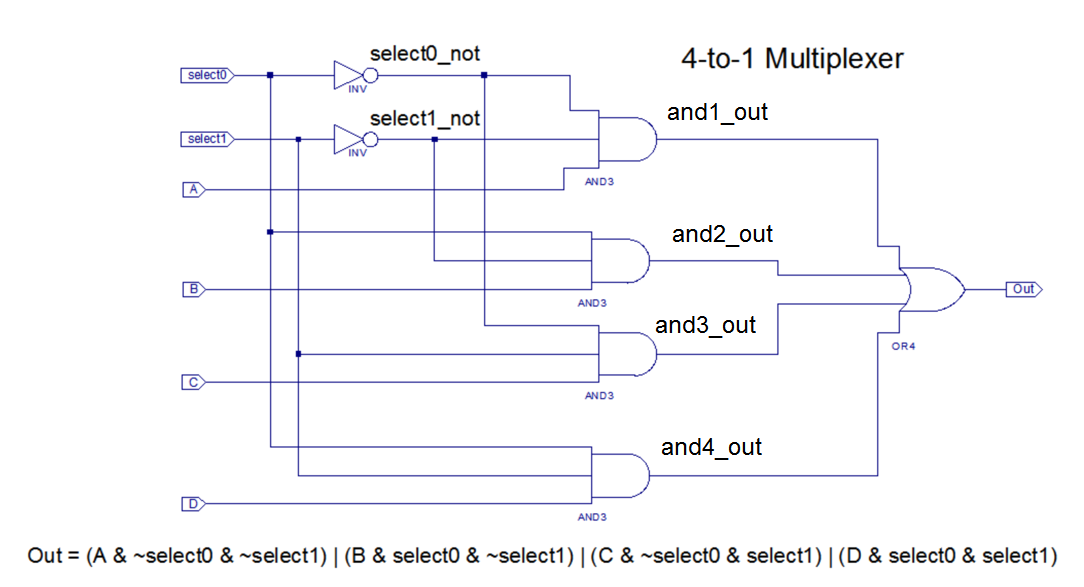
  
Figure 1.5: Output of 2-to-4 decoder structural model in ISim. The stimulus is A1 = 1 and A0 = 0. The time interval is 200 ns - 300 ns. Output is D3 = 0, D2 = 1, D1 = 0 and D0 = 0.

  
Figure 1.6: Output of 2-to-4 decoder structural model in ISim. The stimulus is A1 = 1 and A0 = 1. The time interval is 300 ns - 400 ns. Output is D3 = 1, D2 = 0, D1 = 0 and D0 = 0. Simulation is stopped at 400 ns.

1. **4-to-1 Multiplexer:**

A multiplexer is a combinational logic circuit that allows one of several input signals to be selected and sent to a single output. It is a data selector. An input is selected by controlling the states of select0 and select1 shown in figure 2.1. The selection inputs select0 and select1 can be two separate ports or one single 2-bit port (as seen in examples 2.1, 2.2 and 2.3). Multi-bit ports, wires and registers are called **vectors**.

****Figure 2.1: 4-to-1 multiplexer schematic. The Boolean function for the output (Out) is shown in the bottom. The tilde (~) stands for bit-wise NOT, the ampersand (&) stands for bit-wise AND, and the vertical bar ( | ) stands for bit-wise OR.

![](data:None;base64,iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAYAAAAfFcSJAAAABHNCSVQICAgIfAhkiAAAAAtJREFUCJljYAACAAAFAAFiVTKIAAAAAElFTkSuQmCC)Figure 2.2: 4-to-1 multiplexer truth table.

Example 2.1 shows the code for the structural or gate-level model of the 4-to-1 multiplexer shown in figure 2.1. The structural or gate-level model implements the multiplexer using NOT, AND, and OR gates. The behavioral model implements the multiplexer using conditional statements (case statements). Finally, the dataflow model implements the multiplexer in terms of the Boolean function shown in the bottom of figure 2.1.

// -------------------------------------------------------------  
// --------------- Example 2.1: Mux Structural Model -----------  
// -------------------------------------------------------------

// Behavioral model. Module name is "mux4\_1\_gate" followed by the ports, port directions and port names **module mux4\_1\_gate**( **input** A,  
 **input** B,  
 **input** C,  
 **input** D,  
 **input[1:0]** select, // 2-bit vector input  
 **output** Out);

**wire** select0\_not;  
**wire** select1\_not;  
**wire** and1\_out;  
**wire** and2\_out;  
**wire** and3\_out;  
**wire** and4\_out;

**not**(select0\_not, select[0]); // select[0] is the least-significant bit of select  
**not**(select1\_not, select[1]); // select[1] is the most-significant bit of select  
**and**(and1\_out, select0\_not, select1\_not, A);  
**and**(and2\_out, select[0], select1\_not, B);  
**and**(and3\_out, select0\_not, select[1], C);  
**and**(and4\_out, select[0], select[1], D);  
**or**(Out, and1\_out, and2\_out, and3\_out, and4\_out);

**endmodule**

// -------------------------------------------------------------// ------------------- Example 2.1: End ------------------------  
// -------------------------------------------------------------

// -------------------------------------------------------------  
// -------------- Example 2.2: Mux Behavioral Model ------------// -------------------------------------------------------------

// Behavioral model. Module name is "mux4\_1\_behavior" followed by the ports, port directions and port names **module mux4\_1\_behavior**(**input** A,   
  **input** B,   
 **input** C,   
 **input** D,  
  **input[1:0]** select, // 2-bit vector input  
  **output reg** Out);

**always@**(A or B or C or D or select) **begin** // always block sensitivity list

**case**(select) // all four outcomes will depend on the value of select

2'b00: Out = A;  
2'b01: Out = B;  
2'b10: Out = C;  
2'b11: Out = D;

**endcase**

**end**

**endmodule**

// -------------------------------------------------------------// ------------------- Example 2.2: End ------------------------  
// -------------------------------------------------------------

Notice that a behavioral model for a multiplexer can also be written using if-else statements. Conditional statements (if-else and case) are usually synthesized as multiplexers. **Any Boolean function can be implemented with multiplexers**, however, this is impractical for complex Boolean functions.

// -------------------------------------------------------------// ---------------- Example 2.3: Mux Dataflow Model ------------  
// -------------------------------------------------------------

// Dataflow model. Module name is "mux4\_1\_df" followed by the ports, port directions and port names

**module mux4\_1\_df**( **input** A,  
 **input** B, **input** C, **input** D, **input[1:0]** select, **output** Out);

// "assign" is used to assign a Boolean function to the output.

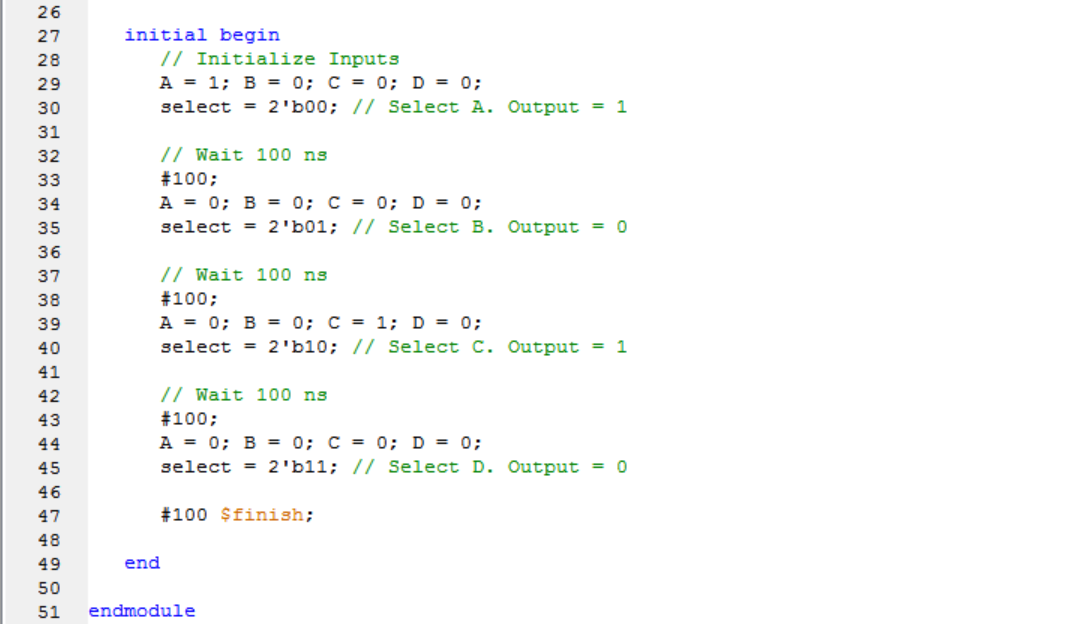
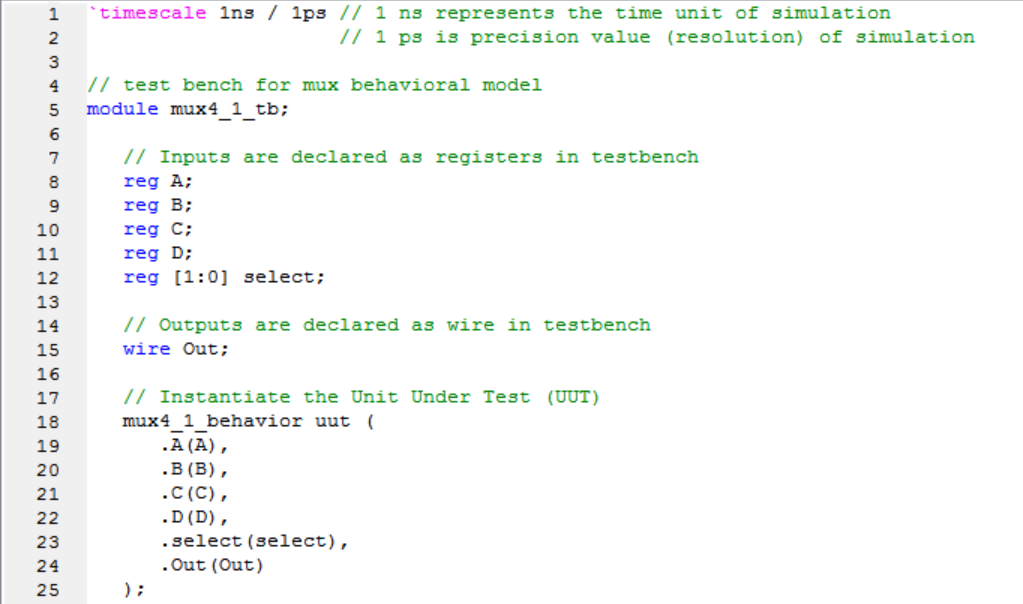
// “select[0]” is the LSB (least significant bit) of select. “select[1]” is the MSB (most significant bit) of select**.**

**assign** Out = (~select[0] & ~select[1] & A) | (select[0] & ~select[1] & B) | (~select[0] & select[1] & C) & (select[0] & select[1] & D);

**endmodule**

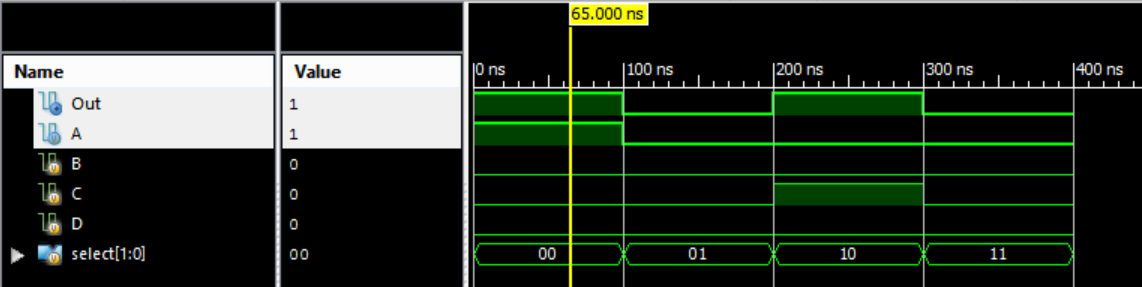
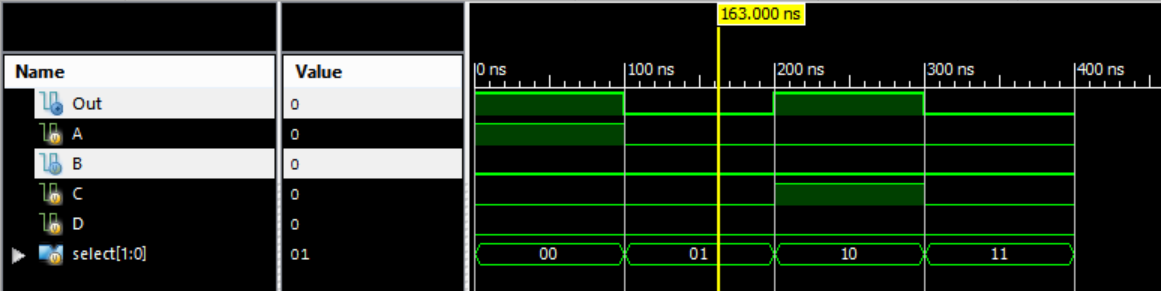
// -------------------------------------------------------------// ------------------- Example 2.3: End ------------------------  
// -------------------------------------------------------------

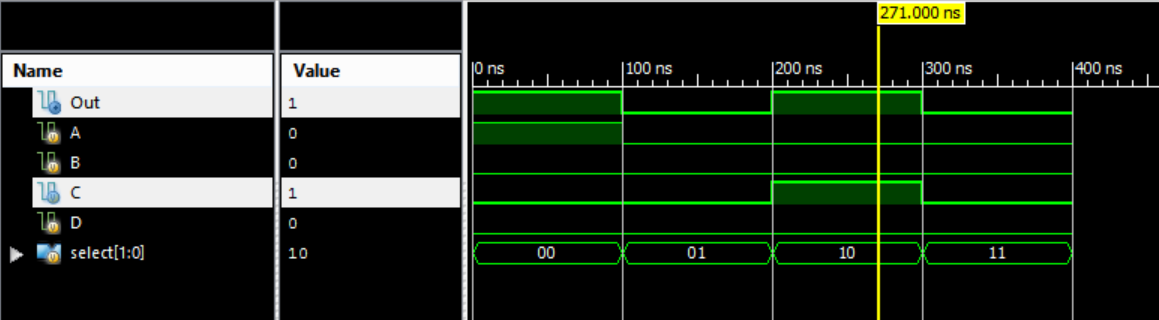
// -------------------------------------------------------------// --------- Example 2.4: Mux Behavioral Model Test Bench ------  
// -------------------------------------------------------------

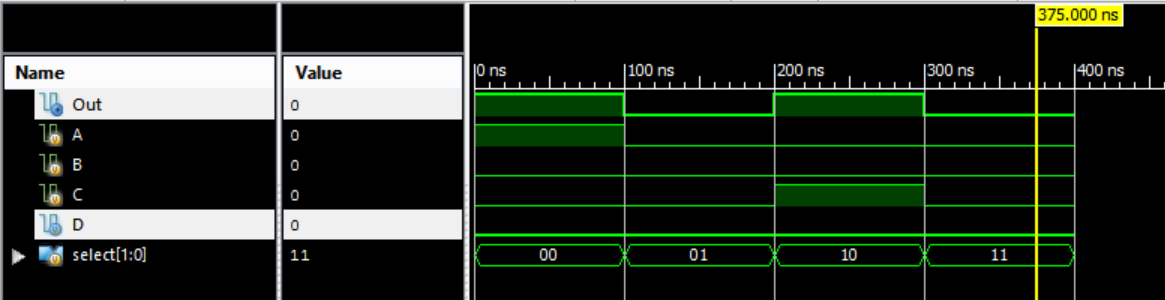


// -------------------------------------------------------------// ------------------- Example 2.4: End ------------------------  
// -------------------------------------------------------------

**4-to-1 multiplexer simulation results in ISim:**

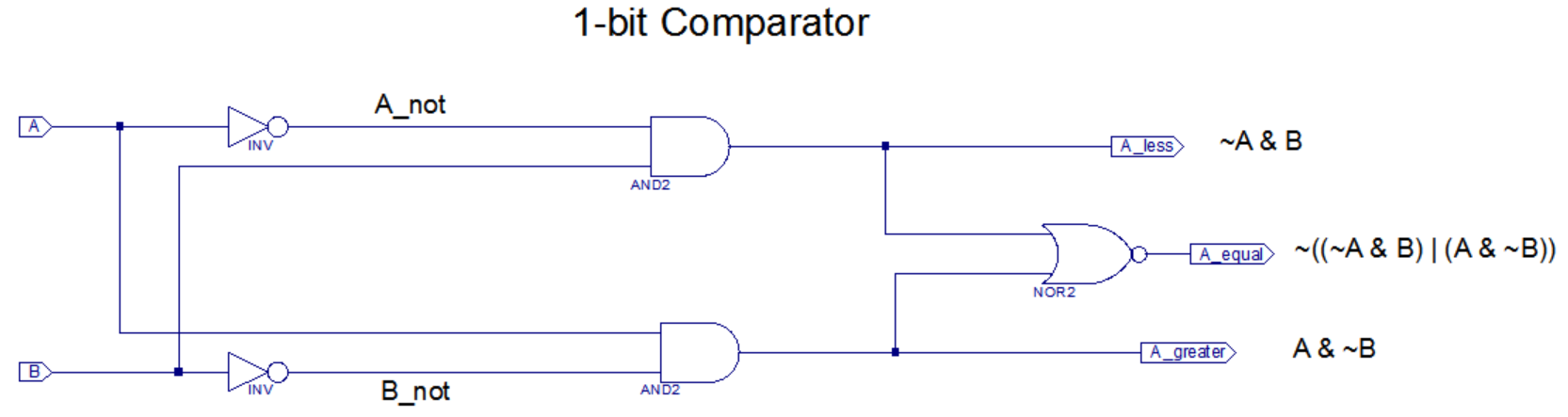
  
Figure 2.3: Output of 4-to-1 multiplexer behavioral model in ISim. The stimulus is select = 2’b00. Time interval is 0 ns – 100 ns. The output is Out = A = 1.  
  
  
Figure 2.4: Output of 4-to-1 multiplexer behavioral model in ISim. The stimulus is select = 2’b01. Time interval is 100 ns – 200 ns. The output is Out = B = 0.

  
Figure 2.5: Output of 4-to-1 multiplexer behavioral model in ISim. The stimulus is select = 2’b10. Time interval is 200 ns – 300 ns. The output is Out = C = 1.

  
Figure 2.6: Output of 4-to-1 multiplexer behavioral model in ISim. The stimulus is select = 2’b11. Time interval is 300 ns – 400 ns. The output is Out = D = 0. Simulation is stopped at 400 ns.

1. **1-bit Comparator:**

A digital comparator is a combinational logic circuit that compares two binary numbers in order to find out whether one binary number is equal, less than or greater than the other binary number. It can be used to produce an output from a counter when a certain number is reached. It also has other applications where a binary number representing a physical variable (such as temperature) is compared with a reference value.

****Figure 3.1: 1-bit comparator. The Boolean functions for the outputs are shown in the right of the figure. Notice that NOR gate is implemented with the vertical bar ( | ) used for bit-wise OR and the tilde (~) used for bit-wise NOT.

![](data:None;base64,iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAYAAAAfFcSJAAAABHNCSVQICAgIfAhkiAAAAAtJREFUCJljYAACAAAFAAFiVTKIAAAAAElFTkSuQmCC)Figure 3.2: 1-bit comparator truth table.

// -------------- Example 3.1: Comparator Structural Model ------------------

// Behavioral model. Module name is "comparator\_gate" followed by the ports, port directions and port names **module comparator\_gate**( **input** A, **input** B, **output** A\_greater, **output** A\_less,  
 **output** A\_equal**);**

**wire** A\_not;  
**wire** B\_not;

**not**(A\_not, A); **not**(B\_not, B); **and**(A\_less, A\_not, B); **and**(A\_greater, A, B\_not); **nor**(A\_equal, A\_greater, A\_less);

**endmodule**

// -------------------------------------------------------------// ------------------- Example 3.1: End ------------------------  
// -------------------------------------------------------------

// -------------------------------------------------------------// ----------- Example 3.2: Comparator Behavioral Model --------  
// -------------------------------------------------------------// Behavioral model. Module name is "comparator\_behavioral" followed by the ports, port directions and port names **module comparator\_behavioral**( **input** A, **input** B, **output reg** A\_greater, **output reg** A\_less, **output reg** A\_equal);

**always@**(A or B) **begin**

**if** (A>B)  
{A\_greater,A\_equal,A\_less} = 3'b100;

**else if** (A==B){A\_greater,A\_equal,A\_less} = 3'b010;

**else**{A\_greater,A\_equal,A\_less} = 3'b001;

**end**

**endmodule**

// -------------------------------------------------------------// ------------------- Example 3.2: End ------------------------  
// -------------------------------------------------------------

The behavioral model of the 1-bit comparator shown in example 3.2 is done with if-else statements. However, it is also possible to do it with a case statement like the one shown in example 2.2 for the 4-to-1 multiplexer. A circuit can have several different behavioral models. Different behavioral models of the same circuit may synthesize in different ways. However, they will always implement the same Boolean function. The synthesized circuits will have the same outputs given the same set of inputs.

// -------------------------------------------------------------// --------- Example 3.3: Comparator Dataflow Model ------------  
// -------------------------------------------------------------

// Behavioral model. Module name is "comparator\_df" followed by the ports, port directions and port names **module comparator\_df**( **input** A, **input** B, **output** A\_greater, **output** A\_less,  
 **output** A\_equal);

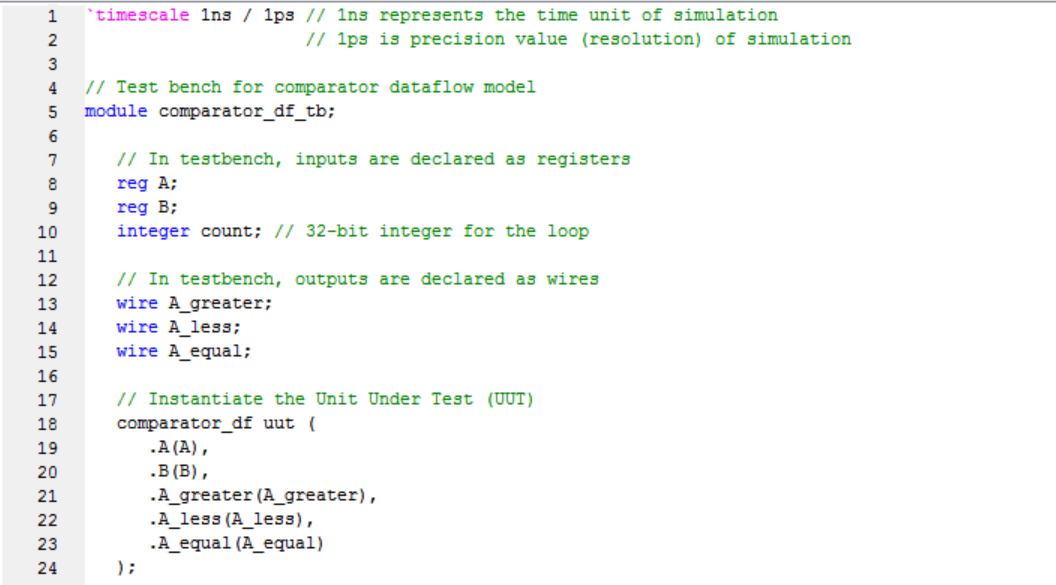
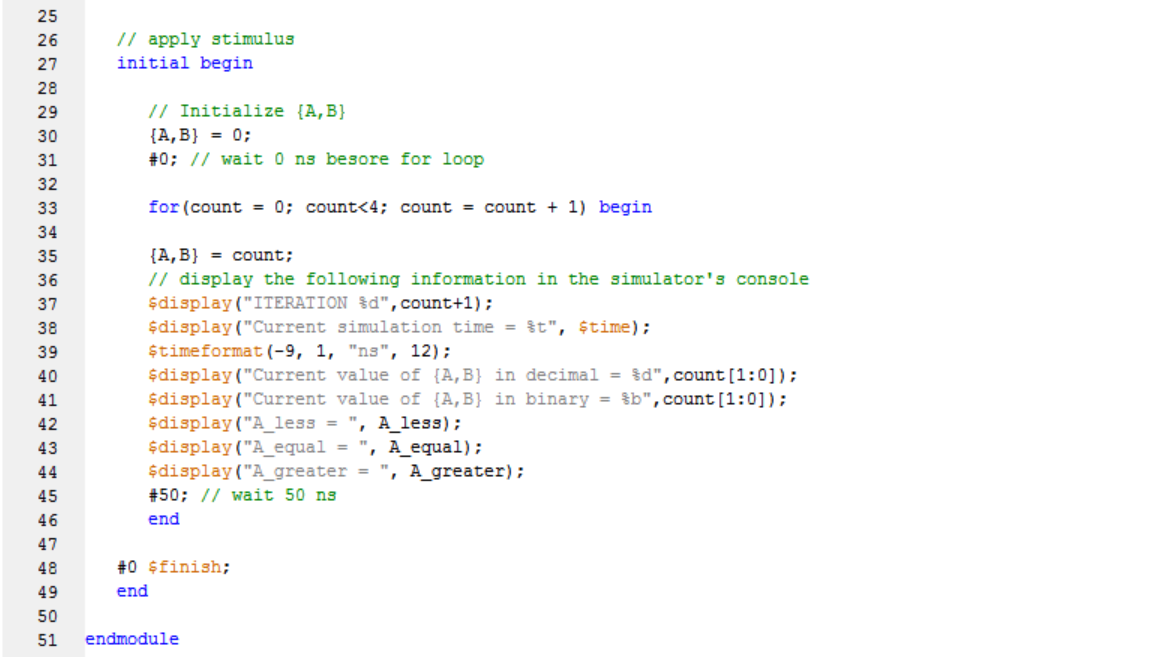
**assign** A\_less = ~A & B; **assign** A\_equal = ~((~A & B) | (A & ~B)); **assign** A\_greater = A & ~B;

**endmodule**

// -------------------------------------------------------------// ------------------- Example 3.3: End ------------------------  
// ------------------------------------------------------------- Notice that there is no bit-wise operator for the NOR operation in example 3.3. The NOR operation is implemented with the OR and NOT bit-wise operators, e.g. ~(X | Y).

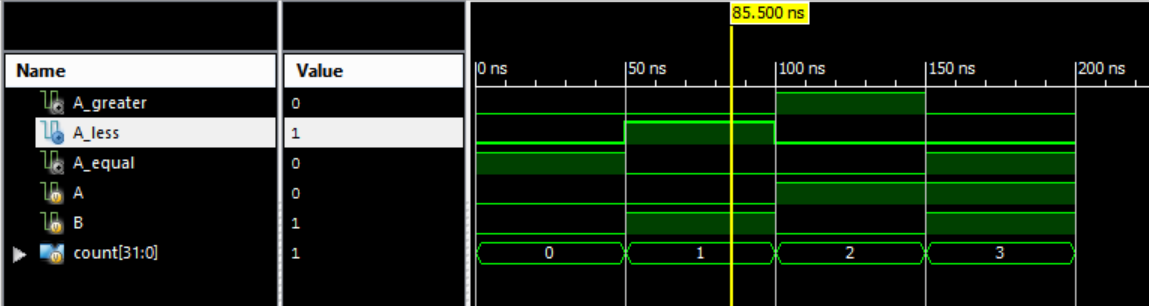
The test bench for the comparator dataflow model is shown below:

// -------------------------------------------------------------// ----- Example 3.4: Comparator Dataflow Model Test Bench -----  
// -------------------------------------------------------------

  
   
  
// -------------------------------------------------------------// ------------------- Example 3.4: End ------------------------  
// -------------------------------------------------------------

The stimuli are applied in a different way from the test benches shown in examples 1.4 and 2.4. A for-loop is used for the application of stimuli. They can also be applied in the same way as examples 1.4 and 2.4. The for-loop is used for convenience in this example. An **integer** called count is declared in the body of the module. Integers are 32-bit signed values. This integer is the **loop counter**: it starts at 0 and ends at 3. The loop goes through four iterations. In the first iteration, the value of count (0 or 2’b00) is stored in {A,B}. Therefore, A = 0 and B = 0 in the first iteration. The values of A, B, A\_less, A\_equal and A\_greater are shown in the console of the ISim simulator and a delay of 50 ns is applied. In the second iteration, the value of count (1 or 2’b01) is stored in {A,B} and the current states of inputs and outputs are shown in the ISim console. The iterations finish at count = 3. These stimuli are taken from the truth table shown in figure 3.2.

**1-bit comparator simulation results in ISim:**

  
Figure 3.3: Output of 1-bit comparator dataflow model in ISim. The stimulus is {A,B} = count = 0 (2’b00). Time interval is 0 ns – 50 ns. The output is A\_equal = 1 (A is equal to B).  
  
  
Figure 3.4: Output of 1-bit comparator dataflow model in ISim. The stimulus is {A,B} = count = 1 (2’b01). Time interval is 50 ns – 100 ns. The output is A\_less = 1 (A is less than B).

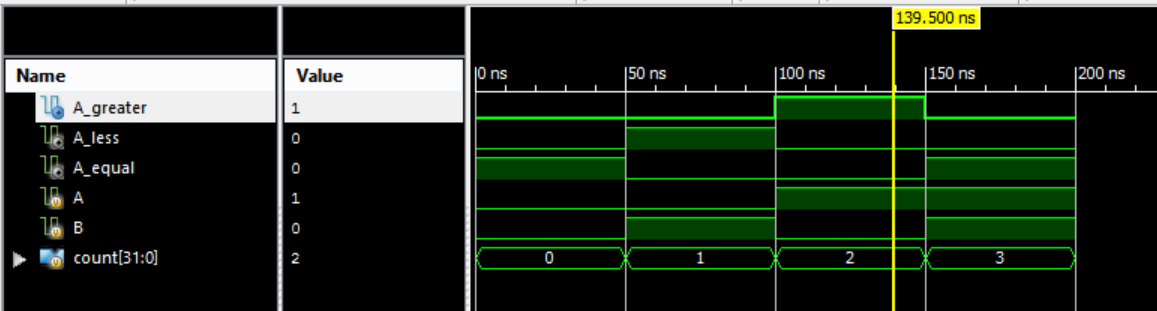
  
Figure 3.5: Output of 1-bit comparator dataflow model in ISim. The stimulus is {A,B} = count = 2 (2’b10). Time interval is 100 ns – 150 ns. The output is A\_greater = 1 (A is greater than B).

  
Figure 3.6: Output of 1-bit comparator dataflow model in ISim. The stimulus is {A,B} = count = 3 (2’b11). Time interval is 150 ns – 200 ns. The output is A\_equal = 1 (A is equal to B). The simulation is stopped at 200 ns.

**ISim Simulator Console:**

ITERATION 1

Current simulation time = 0

Current value of {A,B} in decimal = 0

Current value of {A,B} in binary = 00

A\_less = 0

A\_equal = 1

A\_greater = 0

ITERATION 2

Current simulation time = 50.0ns

Current value of {A,B} in decimal = 1

Current value of {A,B} in binary = 01

A\_less = 0

A\_equal = 1

A\_greater = 0

ITERATION 3

Current simulation time = 100.0ns

Current value of {A,B} in decimal = 2

Current value of {A,B} in binary = 10

A\_less = 1

A\_equal = 0

A\_greater = 0

ITERATION 4

Current simulation time = 150.0ns

Current value of {A,B} in decimal = 3

Current value of {A,B} in binary = 11

A\_less = 0

A\_equal = 0

A\_greater = 1

Stopped at time : 200 ns

1. **8-to-3 Priority Encoder**

The priority encoder is a digital [circuit](https://en.wikipedia.org/wiki/Electronic_circuit) or [algorithm](https://en.wikipedia.org/wiki/Algorithm) that compresses multiple [binary](https://en.wikipedia.org/wiki/Binary_code) inputs into a smaller number of outputs. It encodes the give information into a more compact form. Therefore, it performs the opposite function of the decoder. If two or more inputs are given at the same time, the input having the highest priority (highest magnitude) will take [precedence](https://en.wiktionary.org/wiki/precedence). The priority encoder is often used to control [interrupt requests](https://en.wikipedia.org/wiki/Interrupt_request). In computer systems, there are numerous input devices or peripherals which attempt to supply data simultaneously. A priority encoder enables the input device with highest priority to access the computer first.

Figure 4.1 shows the truth table for the priority encoder. The x represents a “don’t care”. A “don’t care” could be 0 or 1. The input having the highest magnitude takes [precedence](https://en.wiktionary.org/wiki/precedence) so the values of the previous inputs do not matter. If D7 = 1, the outputs will always be Q2 = 1, Q1 = 1 and Q0 = 1. The states of the preceding inputs (D6, D5, D4, D3, D2, D1 and D0) do not matter.

![](data:None;base64,iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAYAAAAfFcSJAAAABHNCSVQICAgIfAhkiAAAAAtJREFUCJljYAACAAAFAAFiVTKIAAAAAElFTkSuQmCC)  
Figure 4.1: 8-to-3 priority encoder truth table.

For this particular digital circuit, only the behavioral model will be shown:

// -------------------------------------------------------------// ------ Example 4.1: Priority Encoder Behavioral Model -------  
// -------------------------------------------------------------

**module priority\_encoder**( **input[7:0]** D,  
 **output reg[2:0]** Q);

**always @**(\*) **begin**

**casez**(D)

8'b00000001: Q = 3'b000;

8'b0000001?: Q = 3'b001;

8'b000001??: Q = 3'b010;

8'b00001???: Q = 3'b011;

8'b0001????: Q = 3'b100;

8'b001?????: Q = 3'b101;

8'b01??????: Q = 3'b110;

8'b1???????: Q = 3'b111;

**endcase**

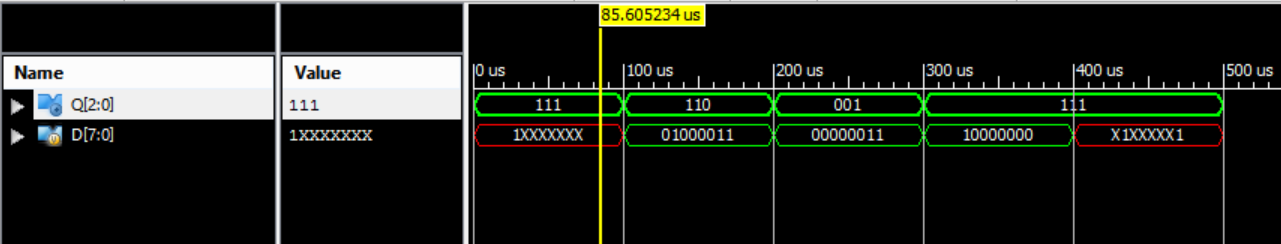
**end**

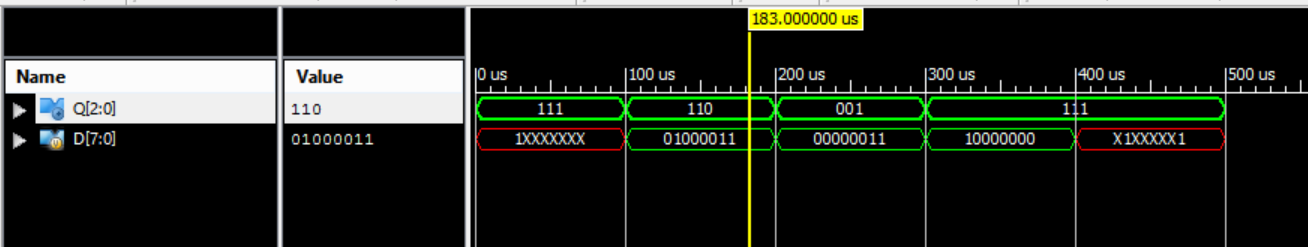
**endmodule**

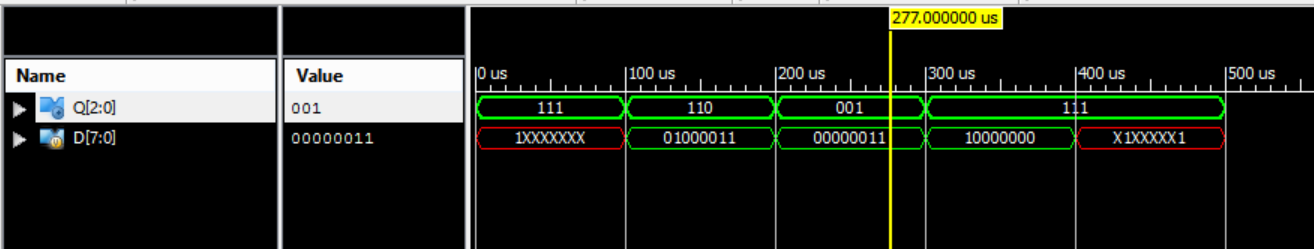
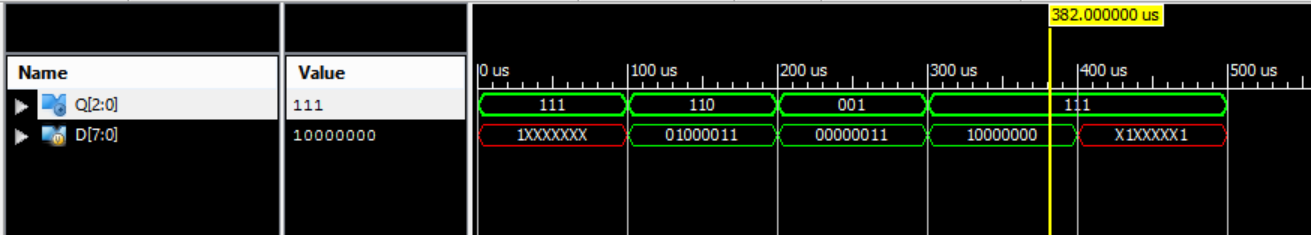
// -------------------------------------------------------------// ------------------- Example 4.1: End ------------------------  
// -------------------------------------------------------------

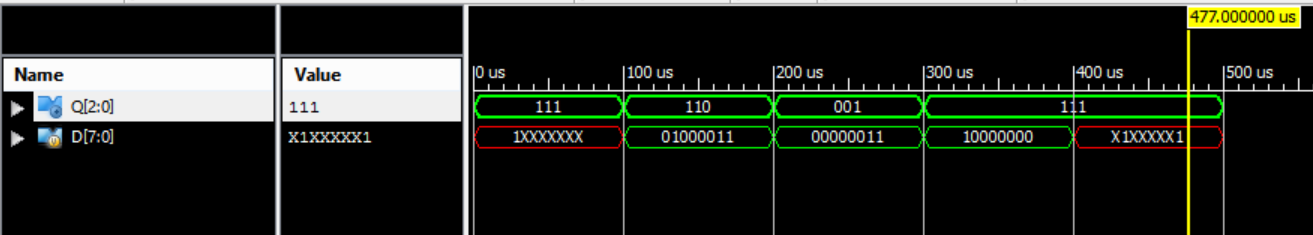
The behavioral model for the priority encoder has several differences from the Verilog codes shown so far. First, you will notice that the **always block** has an asterisk (\*) in the sensitivity list. The synthesizer will automatically include in the sensitivity list all inputs that might affect the outcome of the **always block**. Second, the **casez** statement is used in the **always block**. The **casez** statement allows programmers to use the symbol “**?**” as a “don’t care” element. The **case** statement only allows a user to use 0 or 1. The **casez** statement is necessary to produce the input-output relationship of the truth table shown in figure 4.1.

**8-to-3 priority encoder simulation results in ISim:**

  
Figure 4.2: Input D = 8’b1xxxxxxx and output Q = 3’b111. Time interval is 0 us – 100 us.

  
Figure 4.3: Input D = 8’b01xxxxxx and output Q = 3’b111. Time interval is 100 us – 200 us.

  
Figure 4.3: Input D = 8’b00000011 and output Q = 3’b001. Time interval is 200 us – 300 us.  
  
  
Figure 4.4: Input D = 8’b10000000 and output Q = 3’b111. Time interval is 200 us – 300 us.

  
Figure 4.5: Input D = 8’bx1xxxxx1 and output is still Q = 3’b111. Time interval is 300 us – 400 us. The output will remain the same because the most significant bit is x. The value x can be a 0 or a 1 so the output cannot be decided.

**Final Remarks**

There are many different coding styles in Verilog. Different behavioral models can synthesize as the same circuit or they can synthesize as different circuits that produce the same outputs given the same inputs. The following example of a behavioral model of a 2-to-4 decoder proves this statement:

// -------------------------------------------------------------// ------------ Decoder Behavioral Model (version 2) -----------  
// -------------------------------------------------------------

**module** **dec2\_4\_behavioralv2**(A, D);

**parameter** N = 4; // parameters in Verilog are constants  
**parameter** log2N = 2;  
**input [log2N-1:0]** A; // declare A as input   
**output reg [N-1:0]** D; // declare D as output register  
**integer** i; // integers are 32-bit signed numbers. They are useful in loop statements

**always @**(A) **begin**

**for**(i=0; i<N; i = i+1)

D = 0; // make D equal to 0  
D[A] = 1'b1; // now change the Ath bit of D for a 1

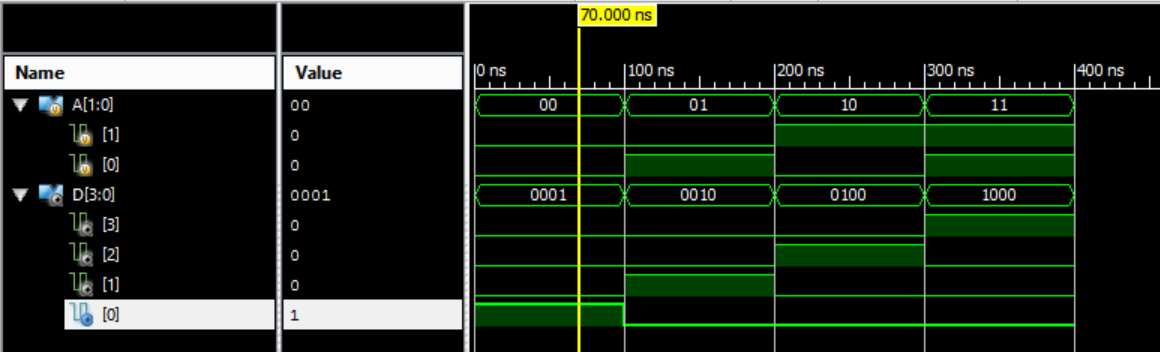
**end**

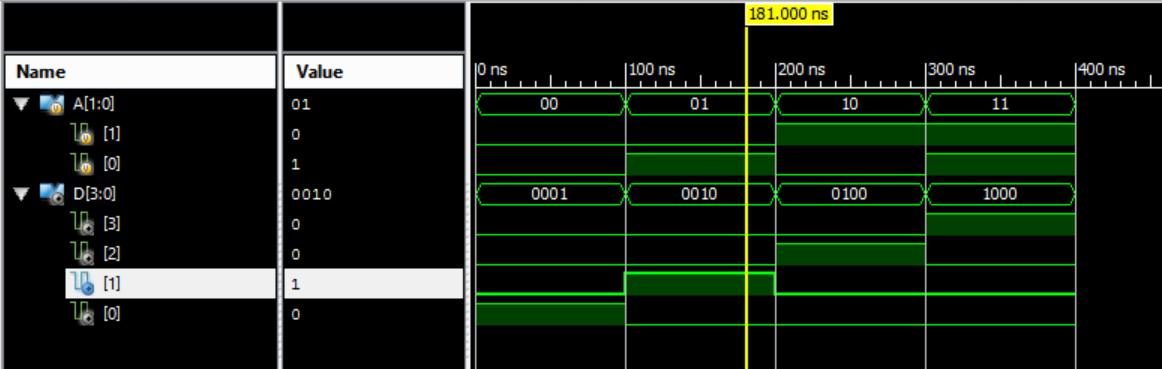
**endmodule**

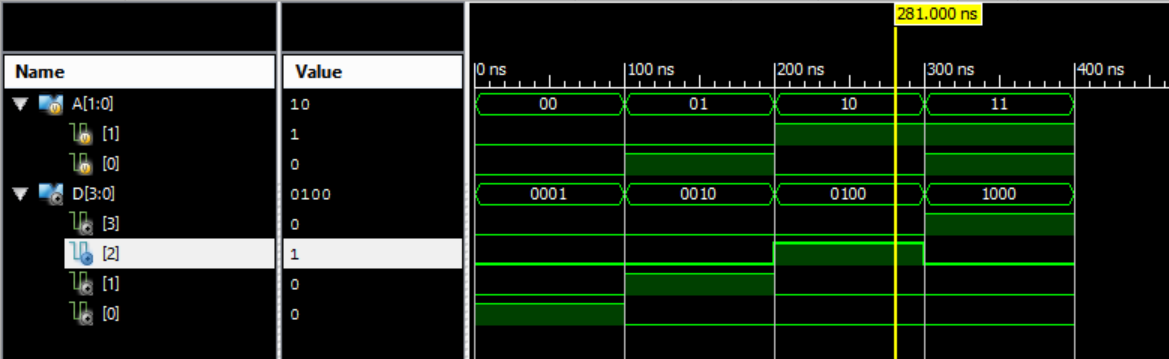
// -------------------------------------------------------------// -------------------------- End ------------------------------  
// -------------------------------------------------------------

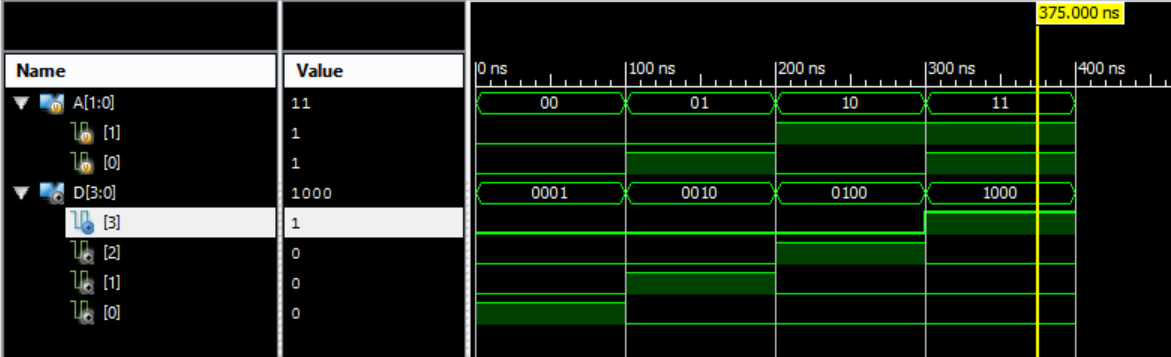
Notice that the structure of the code is different from the one shown in previous examples. The inputs and outputs are not declared next to the name of the module. They are now declared inside the body of the module. A **parameter** and an **integer** are used. The **parameter** is a constant value in Verilog. Parameters don’t belong to any other data type such as net or register types. The **integer** is a general purpose 32-bit signed number. If an integer holds a constant, the synthesizer adjusts it to the minimum width (MSB to LSB) needed at compilation. The for-loop goes through four iterations: it begins at 0 and ends before the **integer** i reaches N. In each iteration, the value of the **output register** D is set to 0 and then the Ath bit of the register is set to 1. This meets the input-output relationship shown in the truth table of figure 1.2.

The simulation results are shown below. Notice that the results are essentially identical to the ones of the structural model of the 2-to-4 decoder. The only difference is that there is a single 2-bit input (A) instead of two separate 1-bit inputs (A1 and A0). There is also a single 4-bit output register (D) instead of four different output registers (D3, D2, D1 and D0).

  
Output of 2-to-4 decoder behavioral model (v2) in ISim. The stimulus is A = 2’b00. The time interval is 0 ns - 100 ns. Output is D = 4’b0001.

 Output of 2-to-4 decoder behavioral model (v2) in ISim. The stimulus is A = 2’b01. The time interval is 100 ns - 200 ns. Output is D = 4’b0010.

  
Output of 2-to-4 decoder behavioral model (v2) in ISim. The stimulus is A = 2’b10. The time interval is 200 ns - 300 ns. Output is D = 4’b0100.

 Output of 2-to-4 decoder behavioral model (v2) in ISim. The stimulus is A = 2’b11. The time interval is 300 ns - 400 ns. Output is D = 4’b100. Simulation is stopped at 400 ns.

For more information about Verilog HDL, watch the following video by **Intel FPGA**:

<https://www.youtube.com/watch?v=PJGvZSlsLKs&t=722s>

Also watch the first tutorials on Vivado and Verilog by **Daniel Ayala (former TTU project lab tutor)**:

<https://www.youtube.com/watch?v=SelRtdOZlP4>

<https://www.youtube.com/watch?v=5BmMca8KKaE&t=4s>