# CS 20 Lab 11 SystemVerilog Primer

Department of Computer Science
College of Engineering
University of the Philippines - Diliman

## 1 Introduction

This document provides an introduction to a subset of *SystemVerilog* constructs relevant to CS 20 and 21.

Note that the amount of pages of this document is mostly due to sample code blocks; the explanations are intended to be brief. This document is to be read in full.

### 1.1 SystemVerilog

*SystemVerilog* is a language used to precisely specify hardware designs and their verification. *Modules* are the basic building blocks in SystemVerilog from which hardware designs can be created and tested.

SystemVerilog programs that you will make in CS 20 and 21 consist of two types of modules: *designs* and *testbenches*. A *design* is a module that can be synthesized into a hardware element while a *testbench* is a module that feeds inputs into *designs* and verifies their outputs and behavior.

### 1.2 Online simulator

You may use `https://www.edaplayground.com/` with the *Aldec Riviera Pro 2017.02* simulator to run SystemVerilog code.

While this requires the creation of a free account, account registration and logging in are relatively painless if you choose to register with your Google or Facebook account (two clicks for login; three for registration).

Instructions on how to use the simulator are found at the bottom of this document. You may answer *all* of the lab report questions using EDA Playground instead of Xilinx Vivado. Things that the online simulator may not be able to hep you with include computation of LUTs used and automatic schematic diagram generation. Note that we require that you use a single tool for the entirety of a lab report - that is, you use either Xilinx Vivado for the entire laboratory report, or you use Aldec Riviera (no switching in the middle).

# 2 Basic Constructs

## 2.1 Modules and ports

Module DEFINITION

A *module* is a *template* or *type* from which a circuit element can be made. The syntax for defining a module is as follows:

```
module <name>(<port list>);
  <body>
endmodule
```

Note that while indentation is not required in SystemVerilog, it is recommended for readability.

*Ports* are circuit nodes that are analogous to input and output ports of circuit elements. *Port directions* can be `input`, `output`, or `inout` (both). Two equivalent ways of specifying ports are shown below.

```
// Both modules have three input ports and two output ports
// Note that these do not do anything for now

module mymodule1(a, b, c, z1, z2);
  input a, b, c;
  output z1, z2;

  // Body here
endmodule

module mymodule2(input a, b, c, output z1, z2);
  // Body here
endmodule
```

A module may have no defined ports; this is commonly done for testbenches. `input` is assumed for ports without a specified direction.

## 2.2 Bit values

A node is equivalent to a variable.

The `logic` data type is used to model circuit nodes. A node with type `logic` assumes one of the following values: `0`, `1`, `Z` (floating), or `X` (unknown).

Note that `logic` is assumed by default if the data type is left unspecified (as with the examples above). While other SystemVerilog data types do exist, we will be using `logic` for most cases in CS 20 and 21.

## 2.3 Multibit values

A multibit `logic` node may be declared with the notation `[<MSb>:<LSb>]` as seen below.

```
module mymodule1(a, b, z);
  input logic [3:0] a;  // Single 4-bit input port; a[3] as MSb; a[0] as LSb
  input logic [0:3] b;  // b has b[0] as MSb and b[3] as LSb
  output logic [3] z;   // [3] is short for [0:3]

  // Body here
endmodule

// Equivalent to above
module mymodule2(input logic [3:0] a,
                 input logic [0:3] b,
                 output logic [3] z);

  // Body here
endmodule
```

Begin indexing bits at 0.
Note that bits follow the natural convention [MSB:LSB].

Note that we could specify if we want a multibit variable's MSB to be [0] or [n-1].
[n-1] appears to be preferred.

# 3 Circuit Design

Note that the example modules shown so far do not do anything. The intended functionality of a circuit must be defined; it can be done so in two ways: *structurally* and *behaviorally.*

## 3.1 Structural design

In CS 20, you have learned that a circuit element may be defined as a network of logic gates; outputs of gates are fed as inputs to other gates. Describing a circuit element in terms of how their internal elements are connected is called *structural design.*

### 3.1.1 Module instantiation

Module INSTANTIATION

Remember that a module is a *template* or *type* from which a circuit element can be made. To use a module as a circuit element, it must be *instantiated* and assigned a label. The syntax of creating a module instance is as follows:

```
<module name> <label>(<port assignments>);
```

A concrete example of module instantiation is shown in the subsection below.

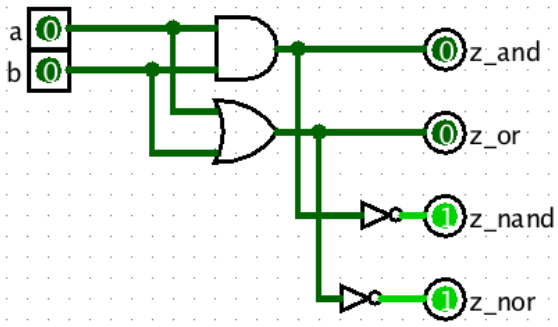
### 3.1.2 Built-in gate modules

Basic 1-bit logic gates (`not`, `and`, `or`) are provided by SystemVerilog as separate *modules.* A module that takes in two 1-bit inputs and returns their 1-bit `AND`, `OR`, `NAND`, and `NOR` values is given below.

```
module combo_gate(input  a,
                         b,
                  output z_and,
                         z_or,
                         z_nand,
                         z_nor);

  // All default gates have the output port as the first argument

  and  and_gate (z_and, a, b);
  or   or_gate  (z_or, a, b);
  not  not_and  (z_nand, z_and);
  not  not_or   (z_nor, z_or);

endmodule
```

The `not` module was instantiated twice, each with a unique label. Note that the extra spacing above is not required, but helps with readability.

Notice that the output nodes `z_and` and `z_or` are *also* connected as inputs into separate `NOT` gates. The figure below depicts how the the above code connects the nodes and gates.




Other built-in gates such as `NAND`, `NOR`, `XOR`, and `XNOR` are provided by SystemVerilog as well. All built-in logic gates accept *two or more* operands; an example will be shown in the next subsection.

### 3.1.3 Internal nodes

Nodes that are not declared as a module port are called *internal nodes*. These nodes (or variables) are used to represent intermediate outputs of internal circuit elements. As the name implies, internal nodes are not accessible outside the module they are defined in.

The example below shows how **single-bit** $a \oplus b = a'b + ab'$ is described *structurally* with the help of internal nodes.

```
module xor_1bit(a, b, z);
  input a, b;
  output z;
  logic not_a, not_b, left, right;

  not not_gate_1(not_a, a);
  not not_gate_2(not_b, b);

  and and_gate_1(left, not_a, b);
  and and_gate_2(right, a, not_b);

  or or_gate(z, left, right);
endmodule
```

The internal nodes `not_a`, `not_b`, `left`, and `right` were introduced to represent and connect the nodes of intermediate results. Note that they do *not* appear in the port list of the module.

In general, nodes that need not be exposed as input or output ports should be kept internal. Below is a sample full-adder implementation described *structurally* using internal nodes and built-in three-input `xor` and `and` gates.

```
module full_adder(a, b, c_in, sum, c_out);
  input a, b, c_in;
  output sum, c_out;
  logic a_or_b, a_or_cin, b_or_cin;

  xor  sum_gate  (sum, a, b, c_in);
  or   or_gate_1 (a_or_b, a, b);
  or   or_gate_2 (a_or_cin, a, c_in);
  or   or_gate_3 (b_or_cin, b, c_in);
  and  and_gate  (c_out, a_or_b, a_or_cin, b_or_cin);
endmodule
```

This is a structural circuit design.

Note that this full_adder is wrong. Should've been c_out = ab + ac_in + bc_in

Whenever the value of a component input changes, the outputs of the affected component are recomputed; outputs of components that use these as inputs are recomputed as well.

### 3.1.4 User-defined modules

*User-defined modules* are instantiated in the same manner as with built-in gates. The example below shows how two instances of a user-defined half adder can be used to construct a full adder.

```
module half_adder(a, b, s, c);
  input a, b;
  output s, c;

  xor xor_gate(s, a, b);                // Output port as first argument
  and and_gate(c, a, b);
endmodule

// Output ports as tail arguments
module full_adder(a, b, c_in, sum, c_out);
  input    a, b, c_in;
  output   sum, c_out;

  logic    s1, c1, c2;

  // Module instantiation
  half_adder  ha1(a, b, s1, c1);        // Output ports as tail arguments
  half_adder  ha2(c_in, s1, s, c2);
  or          or_gate(c_out, c1, c2);   // Output port as first argument

endmodule
```

This set-up surprisingly works.

Similar to a C structure.

You may think of module instantiation as similar to the concept of classes and objects in languages such as Python; a class *(module)* is a *template* or *blueprint* from which the structure and behavior of a *concrete* object *(instance)* is based on. SystemVerilog does have proper classes and objects, but these will not be used in CS 20 and 21.

Do not confuse module instantiation with other programming concepts such as function calling and return values.

## 3.2 Behavioral design

Apart from defining a circuit based on how its internal components are *structured*, SystemVerilog also provides support for defining a circuit based on its *behavior*. Before this can be done, a number of constructs need to be introduced first.

### 3.2.1 Bit literals and construction

Bit literals in SystemVerilog are written as follows:

```
// <N>'<prefix><value>

4'b0110;  // 4-bit value specified in binary
4'ha;     // 4-bit value specified in hex
4'hA;     // Both uppercase and lowercase are valid

// Integers are in two's complement

1;        // Equal to binary ...00001; adjusts with container width
-1;       // Equal to binary ...11111; adjusts with container width
```

Note that integers (int) are a separate data type than booleans (logic)

Observe that N always refer to the binary-bit width.

Note that concatenation operator can concatenate multiple values.

Bits may be concatenated using the {,} operator and replicated using the {N{}}. operator. Individual bits of a multibit value may be accessed using the [] operator while bit slices may be accessed using the [:] operator.

```
// Assume a is logic [3:0] with value 4'b0110
// Assume b is logic [3:0] with value 4'b1001
{a, b};                 // 8'b01101001
{4'b1111, a};           // 8'b11111001
{8{1'b1}};              // 8'b11111111
{4{8'hBE}};             // 32'hBEBEBEBE
b[3];                   // 1'b1 (MSb is 3 for [3:0])
{4{b[3]}};              // 4'b1111
{{2{b[3]}}, b, 2'b0};   // 8'b11100100
b[2:0];                 // 3'b001
```

//bit width adjusts automatically.

//replicate (multiply) N times.

Again, begin indexing bits at 0.
Note that bits follow the natural convention [MSB:LSB].

### 3.2.2 Bitwise operators

SystemVerilog provides standard bitwise operators such as & (AND), | (OR), ~ (NOT), and ^ (XOR). In this case, nodes may be treated as operand variables that can be used on these operators. Examples are shown below.

```
// Assume a and b are logic with equal width
a & b;                  // Bitwise AND
a | b;                  // Bitwise OR
a ^ b;                  // Bitwise XOR
~a;                     // Bitwise NOT
(~a | b) & (a | ~b);    // Computed XOR
```

Standard bitshifting operators are also available: << (logical left shift), >> (logical right shift), and >>> (arithmetic right shift).

```
// Assume n and k are logic [31:0]
n << 2;    // Shift n 2 bits to the left
n >> 2;    // Shift n 2 bits to the right with 0 sign bit
n >>> 2;   // Shift n 2 bits to the right, retaining sign bit
n << k;    // Shift n k bits to the left
```

Special *reduction* operators also exist for the binary logic operators for which the logic operator is applied to all bits of a given value.

```
// Assume n is logic [3:0] with value 4'b0110
&n;    // Results to 0; performs AND reduction (0 & 1 & 1 & 0)
~&n;   // Results to 1; performs NAND reduction ~(0 & 1 & 1 & 0)
|n;    // Results to 1; performs OR reduction (0 | 1 | 1 | 0)
~|n;   // Results to 0; performs NOR reduction ~(0 | 1 | 1 | 0)
^n;    // Results to 0; performs XOR reduction (0 ^ 1 ^ 1 ^ 0)
~^n;   // Results to 1; performs XNOR reduction ~(0 ^ 1 ^ 1 ^ 0)
```



Reduction operators reduces a multi-bit value to its logical 1-bit equivalent.

### 3.2.3 Continuous assignment

Signals continually propagate throughout the wires of a circuit. The *continuous assignment* keyword `assign` models this behavior with the following syntax:

```
// Signals flow from RHS to LHS
assign <node> = <expression>
```

*Continuous assignment* means whenever `expression` changes, the value of `node` changes as well; they *describe* how signals flow from node to node.

Operators can be used in expressions that are continually assigned to nodes. Below is an example of a two-input 32-bit `AND` gate using the said constructs.

```
module custom_and(a, b, z);
  input [31:0] a, b;
  output [31:0] z;

  assign z = a & b;  // Whenever a or b changes, z is recomputed
endmodule
```

Continuous assignments *continuously* ensure that the left-hand side value is recomputed using the *latest values* of the right-hand side. Do not confuse them with sequential assignment statements found in regular programming languages.

Use continuous assignments for simple combinational logic. Below is an example of a full adder that is described *behaviorally* using continuous assignment.

```
module full_adder(a, b, c_in, sum, c_out);
  input a, b, c_in;                                    This is a behavioral circuit design.
  output sum, c_out;

  assign sum = a ^ b ^ c_in;
  assign c_out = (a | b) & (a | c_in) & (b | c_in);
endmodule
```

SystemVerilog is able to *implicitly* translate this into a circuit with the proper internal components even without the user specifying the said components.

When using several `assign` keywords, ensure that no variable appears more than once in the left-hand side of the equal sign.

### 3.2.4 Combining structure and behavior

Circuits can be simultaneously described using both structural and behavioral constructs. The example below defines a half adder behaviorally, then structurally relates half adder instances to a behavioral expression.

```
module half_adder(a, b, s, c);
  input a, b;
  output s, c;

  s = a ^ b;
  c = a & b;
endmodule

module full_adder(a, b, c_in, sum, c_out);
  input    a, b, c_in;
  output   sum, c_out;

  logic    s1, c1, c2;

  half_adder  ha1(a, b, s1, c1);
  half_adder  ha2(c_in, s1, s, c2);

  c_out = c1 | c2;
endmodule
```

Half-adder using behavioral,
Full-adder using structural.

### 3.3 Procedural blocks

Consider the following full adder implementation that uses `P` and `G` (propagate and generate) carry-lookahead adder signals.

```
module full_adder(a, b, c_in, sum, c_out);
  input a, b, c_in;
  output sum, c_out;
  logic p, g;

  assign p = a ^ b;
  assign g = a & b;
  assign sum = p ^ c_in;
  assign c_out = g | (p & c_in);
endmodule
```

Set-up using continuous assignment.

This set-up is inefficient because it takes 2 passes for the updated p and g values to be used by sum and c_out.

Remember that continuous assignment statements are executed *concurrently*. Assuming that all input signals start as `0`, the following are computed *virtually at the same time* when `a` changes to `1` (updated results are not immediately used as inputs):

```
// Pass #1
// Only the value of a is updated

p = 1 ^ 0;            (new p is 1)
g = 1 & 0;            (new g is still 0)
sum = 0 ^ 0;          (old p (0) is used; new sum is still 0)
c_out = 0 ^ (0 & 0);  (old p (0) is used; new c_out is still 0)

// Above statements may be reordered in any way without changing results
```

Notice that `sum` and `c_out` were computed with the *old* value of `p`. When we change input signals of a circuit, we must wait for the *total propagation delay* before we can be assured that the output is consistent with respect to the updated input signals. In the same manner, we expect the simulation above to be consistent with the use of updated input signals.

The updated `p` was not used as input to `sum` and `c_out`. For the outputs to be consistent with this updated signal, the statements are computed once more using the new intermediate values:

```
// Pass #2
// The updated p needs to be propagated

p = 1 ^ 0;            (newer p is still 1)
g = 1 & 0;            (newer g is still 0)
sum = 1 ^ 0;          (new p (1) is used; newer sum is now 1)
c_out = 0 ^ (1 & 0);  (new p (1) is used; newer c_out is still 0)
```

Notice that all outputs are now computed using updated values; ensuring this took *two* simulation passes. A more efficient alternative is discussed below.

### 3.3.1 `always` block and blocking assignment

The inefficiency in *concurrent execution* is that updated values are not being immediately used as inputs. To avoid this, SystemVerilog has *procedural blocks* in which instructions are executed one line at a time by the simulator.

The most general procedural block is the `always` block which has the following syntax:

```
always @(<sensitivity list>) begin
  <statements>;
end
```

The `always` block is executed whenever **any** node in the *sensitivity list* changes its value. A sample implementation of the full adder with `P` and `G` signals using an `always` block is shown below:

The sensitivity list tells the procedural block which nodes to look out for.

```
module full_adder(a, b, c_in, sum, c_out);
  input a, b, c_in;
  output sum, c_out;
  logic p, g;

  always @(a, b, c_in, p, g)
    begin
      p = a ^ b;
      g = a & b;
      sum = p ^ c_in;
      c_out = g | (p & c_in);
    end
endmodule
```

Note that the statements with `=` inside the `always` block are called *blocking assignment* statements; they are different from continuous assignment statements that begin with `assign`. Continuous assignment statements must be placed **outside** any `always` block.

Blocking assignments *block* or prevent the execution of the statement below it until it finishes execution itself; this ensures that assignments are done line-by-line.

Note that for the blocking version of the full adder, a change in `a` requires only a single simulation pass as illustrated below (all signals start at `0`; `a` becomes `1`).

```
1) p = a ^ b;
   p = 1 ^ 0;
   p = 1;

2) g = a & b;
   g = 1 & 0;
   g = 0;

3) sum = p ^ c_in;
   sum = 1 ^ 0;              // The updated p is used
   sum = 1;

4) c_out = g | (p & c_in);
   c_out = 0 | (1 & 0);      // The updated p is used
   c_out = 0;
```

In general, the use of blocking assigments is preferred over that of continuous assignments for more efficient simulation of *combinational circuits* with update dependencies.

### 3.3.2 `always_comb` block

Setting the *sensitivity list* of an `always` block is error-prone. As such, SystemVerilog provides the following notation:



Reserve continuous assignment for simple combinational circuits.

```
always @(*) begin
  <statements>;
end
```

Setting the sensitivity list to `*` automatically includes all nodes at the right-hand side of all assignment statements inside the `always` block.

If the circuit is purely combinational, `always @(*)` can be replaced with `always_comb`. SystemVerilog raises a warning for `always_comb` blocks that are not combinational (e.g., the circuit is sequential; discussed below).

Use continuous assignments for simple combinational logic. Use `always_comb` whenever blocking assignments are relevant.

always_ff mimics flip-flops

### 3.3.3 `always_ff` block and nonblocking assignment

For SystemVerilog to properly recognize and synthesize sequential circuits, `always_ff` must be used with *nonblocking assignment* statements.

The default notation for the sensitivity list assumes *level-triggered* behavior. *Edge-triggered* behavior is modeled using the `posedge` and `negedge` keywords. A positive-edge-triggered D flip-flop is shown below.

positive-edge-triggered D flip-flop

```
module d_flipflop(input logic clk,
                  input logic d,
                  output logic q);
  always_ff @(posedge clk)
    begin
      q <= d;
    end
endmodule
```

In SystemVerilog, if a `begin-end` block contains a single statement (as seen above), the `begin` and `end` keywords may be omitted.

`<=` is called the *nonblocking assignment* operator. As its name implies, it does *not* block execution of statements below it (as with blocking assignment); it therefore is executed concurrently (similar to continuous assignment).

### 3.3.4 Initial flip-flop state and reset

Note that on startup, flip-flops have an unknown value `X`. As such, it is good practice to manually set its value by way of a *reset port*. Below are examples of D flip-flops with *synchronous* (trigger on clock) and *asynchronous* (trigger immediately) reset ports.

```
module d_flipflop_sync_reset(input logic  clk,
                             input logic  reset,
                             input logic  d,
                             output logic q);

  always_ff @(posedge clk)
    if (reset)  q <= 1'b0;
    else        q <= d;
endmodule

module d_flipflop_async_reset(input logic  clk,
                              input logic  reset,
                              input logic  d,
                              output logic q);
  always_ff @(posedge clk, reset)
    if (reset)  q <= 1'b0;
    else        q <= d;
endmodule
```

reset signal is read at posedge clk

on a posedge, if reset == 1, q <= 1'b0;

reset signal is directly passed into flipflop's sensitivity list

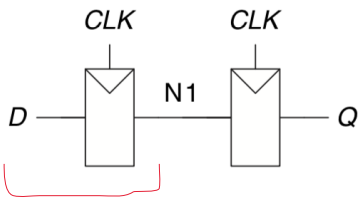
Note that the only difference between the two is that for asynchronous reset, changing `reset` triggers execution of the `always_ff` block as well (i.e., `reset` is level triggered).

The `if-else` statement is no different from C-style `if-else` statements; nonzero values are considered true.

### 3.3.5 Nonblocking assignment and pipelines

Concurrent assignment naturally models pipelined circuits. Below is a schematic of a *synchronizer circuit*.

For a shift register.




D flipflop

This circuit may be divided into three stages with the registers as separators; the left stage (`D`), the middle stage (`N1`), and the right stage (`Q`).

When a triggering edge occurs, the resulting state is that the value of `D` is propagated to `N1`; concurrently, the value of `N1` is propagated to `Q`. A sample scenario of this is illustrated below:

```
Before the triggering edge:
- D  is 1
- N1 is 0
- Q  is 1

After the triggering edge:
- D  is 1
- N1 is 1 (copies old D)
- Q  is 0 (copies old N1)
```

This pipelined flow of signals is naturally modeled by nonblocking assignment as shown below.

```
module synchronizer(input logic  d,
                    input logic  clk,
                    output logic q);
  logic n1;

  always_ff @(posedge clk)
    begin
      n1 <= d;
      q <= n1;
    end
endmodule
```

This synchronizer is similar to a 1-bit serial-in, serial-out shift register where the registered bit is stored in n1.

Modeling this behavior is possible with blocking assignments using clever tricks, but this does not guarantee that SystemVerilog will recognize the module as a sequential component.

In general, always use nonblocking assignment for sequential circuits.

## 3.4 Other relevant design constructs

The constructs described above and including this subsection are *synthesizable*; they can be translated into hardware (as long as the stated guidelines are followed).

The following SystemVerilog constructs will be used in CS 20 and 21.

### 3.4.1 Other operators

Standard arithmetic (`+`, `-`, `*`, `/`, `%`), relational (`==`, `!=`, `<`, `<=`, `>`, `>=`), and logical (`&&`, `||`, `!`) operators are also available in SystemVerilog.

Of note is the difference of `==` and `===` (and their negations `!=` and `!==`); both operators perform bitwise equality and return `1b'1` if equal and `1b'0` otherwise, but `==` and `!=` immediately output `X` if any of their operands contains a `Z` or `X`. See the illustration below for more information.

Recall that Z is floating or high impedance while X is unknown (i.e. 1 and 0 concurrently at the same wire). Note that 1 and 0 always wins over Z.

In our case we treat X as a Don't Care (can also be Error).

```
==    0   1   Z   X
0     1   0   X   X
1     0   1   X   X
Z     X   X   X   X
X     X   X   X   X

===   0   1   Z   X
0     1   0   0   0
1     0   1   0   0
Z     0   0   1   0
X     0   0   0   1

// 4'b01ZX == 4'b01ZX is X
// 4'b01ZX === 4'b01ZX is 1
```

<1/0/Z/X> == <Z/X> → X

multibit logic node == packed arrays
regular array == unpacked array

### 3.4.2 Arrays

In the following text, `logic [3:0] a` is introduced as a multibit `logic` node. More specifically, this is called a *packed array*. In contrast, `logic b[3:0]` is called an *unpacked array*.

A packed array implies that the bits are treated as a single entity. An unpacked array implies that the bits are treated as separate entities. The difference of the two may be illustrated as follows:

```
logic [3:0] a;          // There is one value of type (logic [3:0])
logic b[3:0];           // There are four values with type logic
logic [7:0] c[19:0];    // There are 20 8-bit values

c[19];                  // Refers to first 8-bit element of c
c[19][0];               // Refers to the LSb of the first 8-bit element of c
```

one 4-bit value

### 3.4.3 `case` statement

Using just continuous assignment to model behaviors is not the best option for certain cases. In particular, describing the operation of a *BCD-to-7-segment decoder* is very tedious and error-prone with just continuous assignments (or worse, via structural description).

SystemVerilog provides the `case` statement for this specific use case; the value of a node is taken and executes the statement corresponding to a matching case (or `default` if none). An example is given below.

```
module dec_74ls47(input logic [3:0]  data,
                  output logic [6:0] segments);
always_comb
  case (data)
    //                      abc_defg
    0:       segments = 7'b111_1110;  // _ makes bitstrings more readable
    1:       segments = 7'b011_0000;
    2:       segments = 7'b110_1101;
    3:       segments = 7'b111_1001;
    4:       segments = 7'b011_0011;
    5:       segments = 7'b101_1011;
    6:       segments = 7'b101_1111;
    7:       segments = 7'b111_0000;
    8:       segments = 7'b111_1111;
    9:       segments = 7'b111_0011;
    default: segments = 7'b000_0000;
  endcase
endmodule
```

In this example, 'case' keyword (similar to 'switch' in C) is used to construct a BCD look-up table (LUT).

Instead of `0` to `9` for the case matches, `4'b0000` to `4'b1001` may be used instead for clarity. Note that `case` statements must always be inside an `always`-type block.

`case` statements are usually synthesized into read-only memory components.

## 3.5 Parameterization

*Parameters* are special input values that can dictate port width (among others); their use can make designs more flexible.

The traditional ternary operator `?` `:` with syntax `<condition> ?` `<value if true> :` `<value if false>` is also present in SystemVerilog for modeling 2-way multiplexers. Below is an example of a *parameterized* 2-way multiplexer that specifies its width as a parameter.

2-way, 8-bit mux

Just replace WIDTH to change bit width of mux

```
module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic s,
              output logic [WIDTH-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

parameter #(N) can be used to create a default value. It precedes the port list of the module.

Instantiating with a supplied parameter is done using the `#(N)` operator. Below is a 4-bit, 4-way multiplexer implementation that instantiates the parameterized 2-way multiplexer above with a value of `4` for the width.

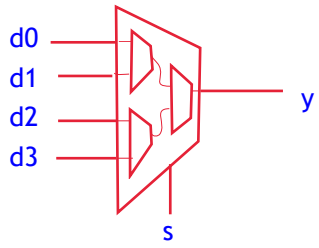
When module is declared with a parameter, this parameter can also be instantiated in later instantiations of the module.

Larger multiplexers can be constructed from smaller ones.

Interesting, d0 = d00, d1 = d01, d2 = d10, d3 = d11

```
module mux4 (input  logic [3:0] d0, d1, d2, d3,
             input  logic [1:0] s,
             output logic [3:0] y);
  logic [3:0] y_topleft, y_botleft;

  mux2 #(4)   topleft (d0, d1, s[0], y_topleft);
  mux2 #(4)   botleft (d2, d3, s[0], y_botleft);
  mux2 #(4)   right   (y_topleft, y_botleft, s[1], y);
endmodule
```

4-bit, 4-way multiplexer.




Observe how nodes can be treated as wires.

# 4 Testbench Constructs

As mentioned, a *testbench* is a module used to verify the correctness of hardware designs. This module has no declared ports and is **not** synthesizable; it cannot be translated into hardware.

## 4.1 `initial` block

The `initial` block is a procedural block as with the `always` block; its main difference is that it is executed *exactly once* during the start of the simulation. Most of the testing logic should be placed here.

An example showing its syntax and usage is shown below.

```
module testbench;
  logic clk;

  initial begin  // Syntax is initial begin <statements>; end
    clk = 0;
  end
endmodule
```

Remember that `begin` and `end` may be omitted for a single statement; since there is only one statement in `initial`, the above code may be rewritten as follows:

```
module testbench;
  logic clk;

  initial
    clk = 0;
endmodule
```

## 4.2 Simulation timing

The following testbench sets up a clock that switches every five (5) delay units.

```
`timescale 1ns/1ps
                                    `timescale <delay unit> / <delay resolution>
module testbench;
  logic clk;

  initial begin                     delay resolution is essentially the time precision,
    clk = 0;                        don't think too much about it, just set it to a really small
  end                               value. Delay unit is far more critical.

  always begin
    #5 clk = ~clk;                  Setting delay unit is similar to setting Logisim tick frequency
  end                               (note that Logisim environments are hard set with a delay
endmodule                           resolution of 1 tick/second)
```

### 4.2.1 Timescale

The `timescale` directive dictates how long a *delay unit* is and what the *delay resolution* is.

The *delay unit* is the unit amount of time a single delay is. In the example above, the delay unit is 1ns; this means that the clock signal switches every 5ns for a total of 10ns per clock cycle.

The *delay resolution* determines how much time passes per time tick. In the example above, the delay resolution is 1ps; this means that the time steps are as follows: 1ps, 2ps, 3ps, etc. Any delay amount is rounded to the nearest stepped resolution.

Valid values are `1`, `10`, and `100` while valid time units are `s`, `ms`, `us`, `ns`, `ps`, and `fs`. The syntax of `timescale` is as follows:

```
`timescale <delay unit value><time unit>/<delay resolution value><time unit>

// Note that the backtick character (`) is used (not the apostrophe ('))
// EDA Playground sets the timescale as a compiler flag
```

### 4.2.2 Delay

Each statement can be given a delay for which a certain amount of time is to pass before its execution. In the example above, the `clk = ~clk;` statement is preceded with `#5`; this means that 5 delay units must pass before the clock switches.

Statements that contain only a given delay are allowed. In the context of a procedural block (e.g., `always`, `initial`), the delay must pass before the succeeding statements are executed. An example that is equivalent to the above given is shown below.

```
module testbench;
  logic clk;

  initial
    clk = 0;

  always begin
    #5;
    clk = ~clk;
  end
endmodule
```

This is the clock driver, an always block with an empty sensitivity list (no "@ ()"), hence we need to add delays to it. So a clock cycle here is 10ns long (period of 10ns: high dur. of 5ns, low dur. of 5ns).

An `always` block with an empty sensitivity list will execute repeatedly and as fast as possible; placing delays will likely be needed.

Note that simulation is **not** executed in real time; setting a delay of 10 seconds will still have the simulation finish as fast as possible.

### 4.2.3 Terminating the simulation

Simulations terminate if there are no more instructions to be executed. Since the `always` block executes instructions, the example testbenches above will run indefinitely (unless terminated by EDA Playground for running too long or too many instructions).

For these instances, the `$finish` task can be used to manually terminate the simulation. An example that terminates after 20 delay units is shown below.

```
module testbench;
  logic clk;

  initial begin
    clk = 0;
    #20;
    $finish;
  end

  always begin
    #5;
    clk = ~clk;
  end
endmodule
```

In this case, there will only be 4 clock cycles.

## 4.3 Concurrency of procedural blocks

In the previous example, both the `initial` and `always` blocks both have delays. Note that while the statements *inside* a procedural block are executed line-by-line, procedural blocks themselves are executed *concurrently*; the `initial` and `always` blocks run independent of each other, so their delays do not add up.

### 4.3.1 Race conditions

If a node is updated (i.e., in the left-hand side of an update) from two or more locations during the same time step, its value will be indeterminate (i.e., dependent on how the simulator was coded); this is called a *race condition* and must be avoided. Ensuring that each node appears at most once in the left-hand side of an assignment statement across multiple blocks helps in avoiding race conditions.

Notice that the testbench examples above assign `clk` in two different blocks. This is *not* a race condition since the `clk` update in `initial` executes immediately while the one in `always` is delayed by 5 delay units. If `#5;` is removed, then the value of `clk` will exhibit a race condition.

To avoid having to watch out for this, we can refactor the code above to localize the update to a single block as seen below.

better testbench

```
module testbench;
  logic clk;

  initial begin
    #20;
    $finish;
  end

  always begin
    clk = 0;
    #5;
    clk = 1;
    #5;
  end
endmodule
```

## 4.4 Output functions

Apart from examining waveforms, another way of verifying program correctness is by printing out the values of input and output signals of a module to see if they are consistent with its intended behavior.

### 4.4.1 `$display`

The `$display` function prints out a string in a manner similar to `printf` of C. In particular, it also accepts a format string as its first argument, followed by variables to be printed out. Note that `$display` prints newlines (`$write` does not).

Its syntax and usage is shown below.

```
module testbench;
  logic [9:0] x;

  initial begin
    x = 49;

    // %<leading zeros><specifier>
    // Leading zeros are spaces for decimal

    $display("x in binary: %0b", x); // x in binary: 110001
    $display("x in hex: %0h", x);    // x in hex: 31
    $display("x as ASCII: %0c", x);  // x as ASCII: 1
    $display("%d %d %6d", x, x, x);  //   49   49     49
    $display("%b %b %6b", x, x, x);  // 0000110001 0000110001 110001
    $display("%x %x %6x", x, x, x);  // 031 031    031
    $display("Percent sign: %%");    // Percent sign: %
  end
endmodule
```

### 4.4.2 `$monitor`

The `$monitor` function is similar to `$display`, but instead of printing immediately, it checks if any nodes have changed values *whenever a delay occurs* and prints if a change has been detected. As the name implies, `$monitor` is useful for printing only when interesting values change.

The example testbench below instantiates the `mux4` module implemented earlier and uses `$monitor` to print out changes in inputs and outputs between each 1-unit delay.

```
module testbench();

  logic [3:0] a, b, c, d, y;
  logic [1:0] s;

  mux4 name(a, b, c, d, s, y);

  initial begin
    $monitor("%0b %0b", s, y);

    a = 4'b1111;
    b = 4'b0111;
    c = 4'b0011;
    d = 4'b0001;

    s = 2'b00; #1;
    s = 2'b01; #1;
    s = 2'b10; #1;
    s = 2'b11; #1;

    $finish;
  end

endmodule
```

## 4.5 Automating verification with `assert`

The above testbench entails having to manually verify that the output of `$monitor` is as expected. `assert` allows automation of this; it checks that a condition is satisfied or not and executes a statement based on the result. Its syntax is shown below:

```
assert (<condition>) <statement if true>
else <statement if false>
```

Note that both statements are optional. An example using this instead of `$monitor` to print test results for `mux4` is shown below.

```
module testbench();
  logic [3:0] a, b, c, d, y;
  logic [1:0] s;

  mux4 name(a, b, c, d, s, y);

  initial begin
    a = 4'b1111;
    b = 4'b0111;
    c = 4'b0011;
    d = 4'b0001;

    s = 2'b00; #1
    assert(y == 4'b1111)
    else $error("00 has wrong output: %0b", y);

    s = 2'b01; #1
    assert(y == 4'b0111)
    else $error("01 has wrong output: %0b", y);

    s = 2'b10; #1
    assert(y == 4'b0011)
    else $error("10 has wrong output: %0b", y);

    s = 2'b11; #1
    assert(y == 4'b0001)
    else $error("11 has wrong output: %0b", y);

    $display("All tests passed!");

    $finish;
  end
endmodule
```

The `$error` function works exactly like `$display`, but implies that an error has occurred. The `$fatal` function prints in the same manner, but terminates the simulation.

## 4.6 Other constructs

Note that the following constructs, when used, must be inside a procedural block.

### 4.6.1 Conditional statements

The syntax of conditional statements is shown below.

```
if <expression> begin
  <statements>;
end

if <expression> begin
  <statements>;
end else begin
  <statements>;
end

if <expression> begin
  <statements>;
end else if <expression> begin
  <statements>;
end else begin
  <statements>;
end
```

Remember `begin` and `end` statements can be omitted for single statements.

### 4.6.2 Looping statements

The syntax of looping statements is shown below.

```
while <expression> begin
  <statements>;
end

do begin
  <statements>;
end
while (<condition>);

for (<start>; <condition>; <after>) begin
  <statements>;
end
```

Note that the following code is an infinite loop due to *arithmetic overflow* on the 5-bit value `i`.

```
module testbench;
  logic [4:0] i;

  initial begin
    for (i = 0; i < 32; i=i+1) begin
      $display("%0d", i);
    end
  end
endmodule
```

One way to address this is to use the 32-bit `int` data type instead as shown below.

```
module testbench;
  initial begin
    for (int i = 0; i < 32; i=i+1) begin
      $display("%0d", i);
    end
  end
endmodule
```

# 5 EDA Playground

## 5.1 Prerequisites

Open `https://www.edaplayground.com/` and register an account. Ensure you are logged in before moving forward.

## 5.2 Running SystemVerilog code

Two files will be immediately created for you: `testbench.sv` and `design.sv`. Modify them as needed.

*Aldec Riviera Pro 2017.02* must be selected under **Tools & Simulators**. The simulation time may be changed with the **Run Time** textbox.

Compilation and simulation is done by clicking the **Run** button at the top row of the page. Alternatively, you may hit `Ctrl+Enter` or `Cmd+Enter` to compile and execute. Pay attention to the **Log** portion at the bottom of the page for compile errors and testbench output.

## 5.3 Displaying waveforms

To display waveforms, ensure that **(1)** your testbench has `$dumpfile("dump.vcd");` followed by `$dumpvars(1);` in its *initial* block and **(2)** the checkbox **Open EPWave after run** is ticked.

Upon running your code, a popup window should be displayed with the signals of your design.

Note that the popup window can be configured to open in a new tab instead by setting it in your account profile page.