ECE385
DIGITAL SYSTEMS LABORATORY
Introduction to SystemVerilog

What is SystemVerilog:

- First adopted in 2005 (IEEE 1800-2005) [1] as an IEEE standard (current version IEEE 1800-2012) and an extension and successor of the standard Verilog (IEEE 1364-2005)
- Both a hardware description language and a hardware verification language (HDVL)
- Synthesizable (able to convert to actual hardware circuit) extensions from Accelera's Superlog
- Verification (not necessary synthesizable) extensions from Synopsys® OpenVera™
- Higher level of abstraction than Verilog and VHDL. Features inherited from Verilog, VHDL, C, C++

The hardware description language can be broken into two different categories: the weakly-typed Verilog/SystemVerilog [2] and the strongly-typed VHDL [3], whose differences and pros/cons will be discussed in the following paragraphs. The industry embraces Verilog/SystemVerilog these days due to their advantages described below, but this doesn't make VHDL obsolete. In fact, many intellectual property cores (source codes) are still written in VHDL, so a good hardware designer most likely needs to be familiar with both languages.

VHDL is a strongly-typed language. That is, its syntax and semantics explicitly represent all the operations and connections in the circuit, i.e., a designer can easily dictate how his code synthesizes, with all the data flow, type conversions, and circuit behavior written in an explicit, straightforward, and self-documenting way. All of the above features make VHDL closer to the actual circuit implementation and less error-prone. But as a result, its drawbacks are the lengthiness of the code and the difficulty of implementing very complicated circuits due to its verbosity. Lacking the higher level abstractions found in weakly-typed languages, the verification aspect of VHDL is also limited compared to Verilog/SystemVerilog.

Verilog and SystemVerilog on the other hand, are weakly-typed languages, that is, the coding reflects more of a functionality of the circuit instead of the actual bit-level wiring. For example, data type conversion in Verilog/SystemVerilog is done implicitly since all data types are inherently interchangeable through their natural bit-level representations. Contrary to the bit-level operation for the VHDL signals, Verilog/SystemVerilog has the capability to manipulate vectors like in a software language. It also supports complicated verifications such as assertions and the built-in random stimulus generator. Since much of the design intentions are implicitly interpreted by the compiler, the coding of Verilog/SystemVerilog is much more concise, but this also adds a degree of uncertainty since the designer may not always predict how his code synthesizes precisely.

Example: VHDL vs. Verilog

```
// Tri-state buffer in VHDL, strongly-typed
library IEEE;                              // Needs to invoke libraries
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity tristate is                         // Interface declaration
port ( inarr : in std_logic_vector (15 downto 0);
        enable: in std_logic;
        outarr: out std_logic_vector (15 downto 0));
end entity;

architecture Behavioral of tristate is     // Behavioral architecture declaration
signal out_signal : std_logic_vector (15 downto 0);  // Ports and signals are separated
begin
        process (enable)                   // Behavioral description process
        begin
          if (enable = '0') then           // Long behavioral statement
            out_signal <= "ZZZZZZZZZZZZZZZZ";
          else
            out_signal <= inarr;
          end if;
        end process;
outarr <= out_signal;
end Behavioral;

// Tri-state buffer in Verilog, weakly typed
module tristate (inarr, enable, outarr);    // Combined Interface and behavioral
        input [15:0] inarr;                                      // declaration
        input enable;
        output [15:0] outarr;

        reg [15:0] outarr;

        assign outarr = (enable) ? inarr : 16'bz;  // C-style conditional statement
endmodule
```

SystemVerilog specifically, is a superset and a newer standard of Verilog intended to both more efficiently model and verify complex designs comparing to Verilog. To achieve this, it incorporates some of the strongly-typed features from VHDL, such as data type casting, as well as higher abstractions from software languages, such as classes and structs. SystemVerilog also improves upon the various constructs and syntax to make the coding more concise. We will focus on SystemVerilog and its syntax, but we will compare its features against VHDL and Verilog so you can better understand the differences. Below is an example showing how SystemVerilog greatly reduces the port and data type declarations comparing to Verilog.

<u>Example</u>: Verilog vs. SystemVerilog

```
// Verilog module construct
module Verilog (clk, enable, data, out);
        // Input port declaration
        input           clk, enable;
        input   [2:0]   data;
        // Output port declaration
        output          out;
        // Optional data type
        //              declaration
        wire            clk, reset, enable;
        reg             out;

        always @ (posedge clk)
        begin
          if (enable)
            out <= /*do something*/;
        end
endmodule
```

```
// SystemVerilog module construct
module SysVerilog (input  clk, enable,
                   input  [2:0] data,
                   output       out);
        // Data type always default to
        //          type logic

        always @ (posedge clk)
        begin
          if (enable)
            out <= /*do something*/;
        end
endmodule
```

It is important to keep in mind that not all of the constructs in VHDL, Verilog and SystemVerilog are synthesizable. This is because the verification aspect of these languages borrows ideologies from software languages to different degrees. Many constructs (especially in SystemVerilog) are intended for the convenience of simulations and are therefore non-synthesizable. It is not easy to keep a complete list of synthesizable/non-synthesizable constructs, especially when the list is not yet standardized for SystemVerilog [4].

One general way to quickly guess if a specific construct is synthesizable or not is by looking at its relationship with its actual hardware circuit: if a construct is easily correlating to some kind of circuits, then it is most likely a synthesizable construct; if a construct is styled like a software language and it's hard to imagine it to any hardware component at all, then it is most likely a non-synthesizable construct. However, since each compiler vendor defines its own subset of synthesizable constructs, the best way to check if your code is synthesizable is to see if the compiler generates any error during the compiling process. We will use the keyword "<u>non-synthesizable</u>" in this tutorial for any non-synthesizable constructs.

Uses both 2-state and 4-state Data Types:

2-state Data Types (0,1)	
bit	User defined vector size. Default unsigned
byte	8-bit signed integer
shortint	16-bit signed integer
int	32-bit signed integer
longint	64-bit signed integer
4-state Data Types (0, 1, x, z)	
logic (**reg**)	User defined vector size. Default unsigned
integer	32-bit signed integer
time	64-bit signed integer

The above synthesizable data types can be enumerated using **enum**, where its values are defined by names. Besides the synthesizable data types, SystemVerilog also supports a few other non-synthesizable data types for verification purposes: **time** (64-bit unsigned integer as simulation time units), **shortreal** (same as float in C), **real** (same as double in C) and **string** (same as string in C).

Note: The **logic** data type is equivalent to the **reg** data type. It is introduced in SystemVerilog intended to replace the commonly used **reg** data type in Verilog to prevent misconception, where the latter is often mistaken to be an actual hardware register (it is in fact just a signal!).

Hierarchy:

- **Module and Procedural Blocks**
  A **module** is the primary design unit in SystemVerilog. Multiple modules may be combined to form a larger design. A module-procedural block pair provides design description.

  module declaration provides the external interface to the design entity. It contains pin-out description, interface description, input-output port definitions etc.

Example:

```
// Logic is contained within modules
module Example (input          clk, enable,    // Single-bit inputs, default logic type
                input    [2:0] data,           // Three-bit input, default logic type
                output         out);           // Single-bit output, default logic type

     initial                         // "initial" procedure block initializes the variables
     begin                           // Used for testbench initialization. Not synthesizable.
          out = 0;
     end
```

```
        always @ (posedge clk)  // "always" procedure block triggered by the
        begin                                              // rising edge of clk
                if (enable)
                  out <= /*do something with the input*/;
        end
endmodule
```

Procedure blocks describe the behavior of an entity or its structure (gates, wires, etc.) using SystemVerilog constructs. There are three types of procedure blocks in SystemVerilog, and they can be used in a nested fashion.

- **initial**: an initial block is used to initialize testbench assignments and is carried out only once at the beginning of the execution at time zero. It executes all the statements within the "**begin**" and the "**end**" statement without waiting. <u>Non-synthesizable</u>.

- **always**: an always block namely execute over and over again throughout the execution period. When a triggering event occurs, all of the statements within "**begin**" and "**end**" are executed once, and then waits for the next triggering event to occur. This waiting and executing of the "**always**" block is repeated over and over again throughout the circuit operation. To avoid unintended latches, SystemVerilog has three types of the "always" procedures.
  - **always_comb**: forcing combinational logic behavior; inferred sensitivity list (sensitivity list does not need to be explicitly written).
  - **always_latch**: forcing latched logic behavior; inferred sensitivity list.
  - **always_ff**: forcing synthesizable sequential logic behavior; sensitivity list needs to be explicitly written.

- **fork-join**: a fork-join block parallelizes the statements contained within themselves. It spawns multiple sequential procedures in parallel in addition to the original. It is used in testbench to divert simultaneous tasks (e.g. one parallel procedure executes the design and the other monitors any abnormalities such as an infinite loop). <u>Non-synthesizable</u>.
  - **fork-join**: waits for all parallel processes to complete before continuing the sequential process.
  - **fork-join_any**: waits for the first of all the parallel processes to complete before continuing the sequential process.
  - **fork-join_none**: does not wait for any of the parallel processes to complete before continuing the sequential process.

<u>Example</u>:

```
// Logic is contained within modules
module Example (input   a, b, z, enable, clk, reset,
                output  z);
```

```
/*=============================================================*/
        // "always_comb" procedure

        always_comb
        begin: myComb       // "Named block" for readability.  The naming does
                z = a + b;                  // nothing, so its presence is not necessary.
        end: myComb         // End the named block here

/*=============================================================*/
        // "always_latch" procedure

        always_latch
        begin
                if (enable) begin
                  z <= a + b;
                end
        end

/*=============================================================*/
        // "always_ff" procedure triggered by either the rising edge of the reset or the
        rising edge of clock

        always_ff @ (posedge clk or posedge reset)   // ordering within the parenthesis
        begin                                                //does not matter
                if (reset) begin
                  z <= 0;
                end else begin
                  z <= a + b;
                end
        end
endmodule
```

Procedural blocks execute concurrently with respect to each other. Only the statements within a sequential block (**begin-end** block) execute sequentially in the order of their appearance. If there is more than one statement in the module, you will have to either place them in separate procedural blocks so they execute in parallel or in the same procedural block with a sequential **begin-end** block. This is the fundamental difference between a hardware language like SystemVerilog and a software language.

SystemVerilog is **case sensitive**, that is, lower case letters are unique from upper case letters. All SystemVerilog keywords are lower case.

- **Ports and Interfaces**
  Ports are the interconnections between a module and the outside world. There are three types of ports in Verilog and SystemVerilog:

- **input**: Has a data type of Nets or Registers. Default to be "**wire**" in Verilog, and "**logic**" in SystemVerilog.
- **output**: Has a data type of Nets or Registers. Default to be "**reg**" in Verilog, and "**logic**" in SystemVerilog.
- **inout**: Has a data type of Nets. Default to be "**wire**" in Verilog, and "**logic**" in SystemVerilog.

Notice the differences in the port data type declarations between Verilog and SystemVerilog. While the port data types are distinguished in Verilog, all the ports in SystemVerilog share the same **logic** type. This reduces the coding complexity as shown in the first example of this tutorial.

Simple combinational logic can be assigned outside of the procedural blocks using the "**assign**" statement. The statement is carried out continuously without the need of a sensitivity list. In this case, the output of some simple logic can be connected directly to the output port of type **logic**. The below two examples are two different ways of implementing an identical AND gate:

<u>Example</u>:

```
// "assign" statement
module AND_Gate (input   a, b,
                 output  z);

     assign z = a & b;

endmodule
```

```
// Prodecure block
module AND_Gate (input   a, b,
                 output  z);

     always @ ( a or b)
     begin
          z = a & b;
     end

endmodule
```

SystemVerilog has the capability to bundle the module ports together to cut down repetitions in the module interconnections. This is especially effective in the case where a master module sends a large number of controlling signals to many slave modules. This is achieved by using the **interface** block to group the relevant ports, while using the **modport** statement to define the directional views of the ports.

<u>Example</u>:

```
// Interface declaration
interface Bus #(SIZE=2);  // Interface declaration with variable SIZE
     logic  [SIZE-1:0]  m2s;      // Master to slave connection
     logic  [SIZE-1:0]  s2m;      // Slave to master connection

     modport master (input s2m, output m2s);   // Master view of the interfaces
     modport slave (input m2s, output s2m);    // Slave view of the interfaces
```

```
endinterface

// Master module
module Master (Bus.master interface1);

        // Do something using "interface1.m2s" and "interface1.s2m"

endmodule

// Slave module
module Slave (Bus.slave interface2);

        // Do something using "interface2.m2s" and "interface2.s2m"

endmodule

// Top-level module
module Example ();

        Bus     myInterface;
        Master  myMaster (myInterface);
        Slave   mySlave (myInterface);

endmodule
```

Data Types:

Verilog uses three primary data types: "Nets", "Registers" and "Constants", while SystemVerilog unifies "Nets" and "Registers" into simple **logic**. We will only introduce the most commonly used ones here.

- Nets: Represents a physical wire in the circuit for connections between components. They do not hold any value themselves.
  - ➢ **wire**: Simple interconnecting wire between the design components.
    Syntax: **wire**; **wire** [<*MSB*>:<*LSB*>]
    - o Example: **wire** my1BitWire; **wire** [3:0] my4BitWire;
  - ➢ **supply0, supply1**: Wires that are tied to logic '0' (ground) or '1' (power).
    - o Example: **supply0** myGround; **supply1** myPower;

- Registers: Variables used to store the value assigned to them until another assignment changes their value.
  - ➢ **logic**: unsigned variable. Syntax: **logic**; **logic** [<*MSB*>:<*LSB*>]
    - o Example: **logic** my1BitVariable; **logic** [3:0] my4BitVariable;
  - ➢ **integer**: signed variable (32 bits)
    - o Example: **integer** myInteger;

- **real**: double-precision floating point variable. Non-synthesizable.
  - Example: **real** myReal;

- Constants:
  - Integer Constants: Syntax: <*sign*><*size*>'<*radix*><*value*>
    - Sign: "–" for signed negative, else unsigned or signed positive.
    - Size: Size of the number in binary bits. Default to 32 bits if unspecified.
    - Radix: *b* (binary), *o* (octal), *d* (decimal), *h* (hexadecimal)
    - Value: Value of the number in the form indicated by the radix.

      Note: If <*size*> is smaller than <*value*>, then the excess bits in <*value*> are truncated. If <*size*> is larger than <*value*>, then leftmost bits are filled based on the value of the MSB in <*value*>. Specifically, a MSB of '0' and '1' is filled with '0', 'Z' is filled with 'Z', and 'X' is filled with 'X'.

    - Examples:

Integer	Stored as	Comments
1	00000000000000000000000000000001	Default to be 32 bits
-8'hFA	00000101	2's complement of 'FA
5'b11010	11010	5 bits of binary
'hF	00000000000000000000000000001111	Default to be 32 bits
6'hF	001111	Extension to 6 bits
6'h8F	001111	Truncation to 6 bits
7'bZ	ZZZZZZZ	7 bits of high impedance

  - Real Numbers (Non-synthesizable.): <*value*>.<*value*> or <*mantissa*>E<*exponent* >
    - Examples: 1.5; 3.6E3;

Data Structures, Classes and Packages:

Designed like a software language, SystemVerilog supports simple data structures **struct** and classes **class** to package variables and functions for verification purposes. "**package**" can also hold variables and functions, but is more equivalent to "namespace" in C language for sharing common code across modules. This greatly reduces the amount of code required to model complex circuits comparing to Verilog. While structures are synthesizable, classes and packages are non-synthesizable.

To initialize the entire structure, use the syntax: <*struct_name*> = *'{*<*member1_value*>, <*member2_value*>, *…}*, where the "'" operator in front of the bracket represents data casting (from array to struct). It is used to distinguish data casting from concatenation.

To assign a specific member, use: <*struct_name*>.<*member_name*> = <*member_value*>

Example:

```
// Anonymous structure
struct {
        logic           mybit,          // Single-bit logic
        int             myint,          // 32-bit integer
        logic   [2:0]   myarray,        // Three-bit array
} mystruct_instance;

// Typed structure (defined somewhere)
typedef struct {
        logic           mybit,          // Single-bit logic
        int             myint,          // 32-bit integer
        logic   [2:0]   myarray,        // Three-bit array
} mystruct_type

mystruct_type  mystruct_instance;       // Struct instance declared elsewhere

// Class (mainly used in verification testbench)
class myclass_name;
        // Class properties
        logic           mybit,          // Single-bit logic
        int             myint,          // 32-bit integer
        logic   [2:0]   myarray,        // Three-bit array

        // Class method 1 - function that does something and returns something
        function int myfunc_get
                return myint;
        endfunction

        // Class method 2 - task that does something without returning anything
        task mytask_set (int i)
                myint = i;
        endtask

endclass

/*=====================================================*/
// File mypackage.sv
package mypackage_name;
        // Class properties
        logic           mybit;          // Single-bit logic
        int             myint;          // 32-bit integer
        logic   [2:0]   myarray;        // Three-bit array

        // Class method 1 - function that does something and returns something
        function int myfunc_get();
```

```
                return myint;
        endfunction

        // Class method 2 - task that does something without returning anything
        task mytask_set (int i);
                myint = i;
        endtask

endpackage
// End file mypackage.sv

// File mymodule.sv
`include "mypackage.sv"
import mypackage_name::*;       // import everything from the package

// Top-level module
module use_package ();

   /*access package members using mypackage_name::<member_name> */
   int     newint = 10;
   mypackage_name::mytask_set(newint);           // access member function
   logic [2:0] newarr = mypackage_name::array;   // access member variable

endmodule
// End file mymodule.sv
/*=========================================================*/
```

Data Assignment:

Two types of data assignments are available depending on whether the user wants the statements to be executed in parallel or sequentially. The two different assignments will often produce very different results, so it is important to be careful when using the two different assignments.

- Blocking Assignment: Sequential assignments made with "=" symbol. It blocks the execution of next statement until the current statement is executed, which means the assignment results are updated immediately. Analogous to variable assignment in VHDL.
- Non-blocking Assignment: Parallel assignments made with "<=" symbol. The next statement is not blocked due to the execution of the current statement, which means the assignment results are updated only until the end of the current cycle. Analogous to ":=" in VHDL.

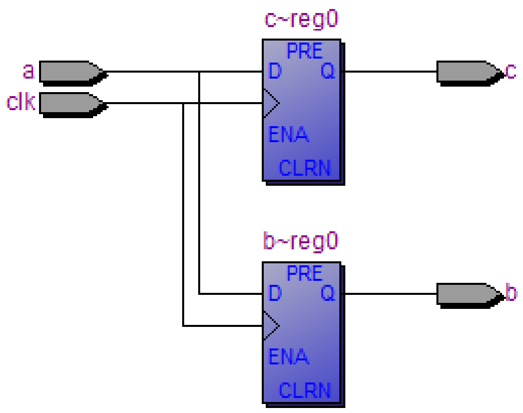
Note: Non-blocking assignment of multiple statements in a single procedural block is equivalent to single assignments in multiple procedural blocks.

A general guideline is to use non-blocking assignments for sequential always block, and blocking assignments for combinational always block.

Example: Blocking Assignment

```
module blocking (input clk, a,
                 output b, c);

        always @ (posedge clk)
        begin
// This is evaluated first serially
                b = a;
// Wait for the above evaluation to
//   complete before its own evaluation,
//   which makes the final evaluation to
//   be c=b=a.
                c = b;
        end
endmodule
```

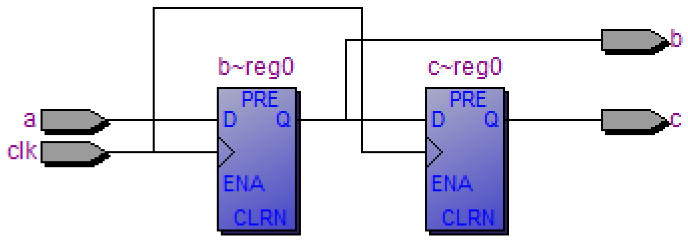
RTL Synthesis:



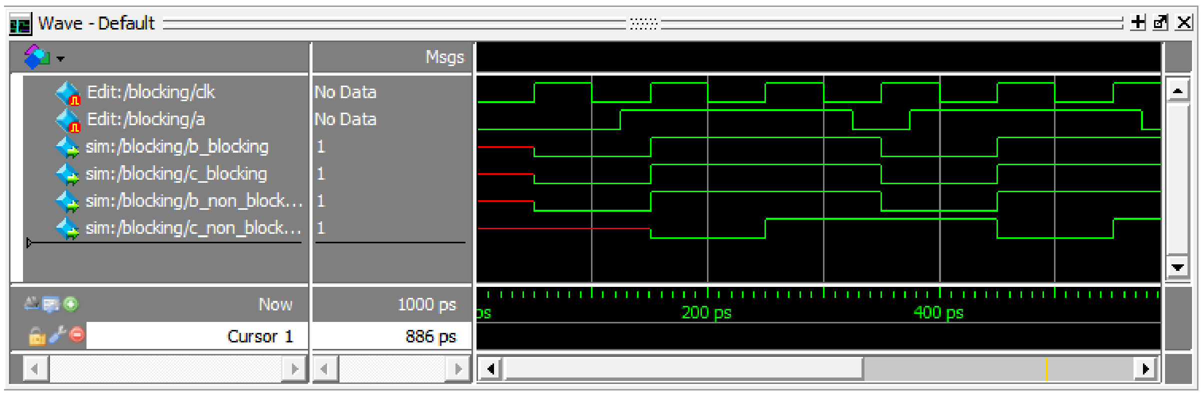
Example: Non-blocking Assignment

```
module nonblocking (input clk, a,
                    output b, c);

        always @ (posedge clk)
        begin
// These two statements are evaluated
//   in parallel disregarding the
//   dependence upon one another,
//   which means b is assigned to a
//   independently from c assigned to b
                b <= a;
                c <= b;
        end
endmodule
```

RTL Synthesis:



Simulation:



	Msgs
Edit:/blocking/clk	No Data
Edit:/blocking/a	No Data
sim:/blocking/b_blocking	1
sim:/blocking/c_blocking	1
sim:/blocking/b_non_block...	1
sim:/blocking/c_non_block...	1
Now	1000 ps
Cursor 1	886 ps

- Notice that *b_blocking* and *c_blocking* holds identical value from a blocking simulation, while *c_non_blocking* lags a clock cycle from *b_non_blocking* from a non-blocking simulation.

Data Casting:

SystemVerilog can use the "'" operator for conversion between data types.

Example: **int**'(x) (casting x to type **int**); **signed**'(x) (casting x to **signed**); 5'(x) (casting x to change to 5 bits in size)

Operators:

Operators can be used in expressions involving signals, variables, or constant object types. Here are some of the useful operators:

- Arithmetic: +, -, *, /, % (modulus)
- Relational: === (case equality – where *'X'* and *'Z'* also have to match precisely), !== (case inequality), == (logical equality – where only *'0'* and *'1'* have to match. Result is 'X' if either operand contains *'X'* or *'Z'*), != (logical inequality), <, <=, >, >=
- Logical: ! (logical not), && (logical and), || (logical or)
- Bit-wise Operators: ~ (negation), & (and), ~& (nand), | (or), ~| (nor), ^ (xor), ^~ or ~^ (xnor)
- Concatenation: {,} (used to combine bits). Ex: {a, b[2:0], 3'b010}
- Shift: << (left shift), >> (right shift)
- Replication: {<*n*>{<*m*>}} (replicate the value m, n times). Ex: {a, 2{b, c}} //equivalent to {a, b, c, b, c}
- Conditional: <*condition_expr*> ? <*true_expr*> : <*false_expr*> (select either the true or the false expression based on the condition expression)

If- Else and If- Else if Statements:

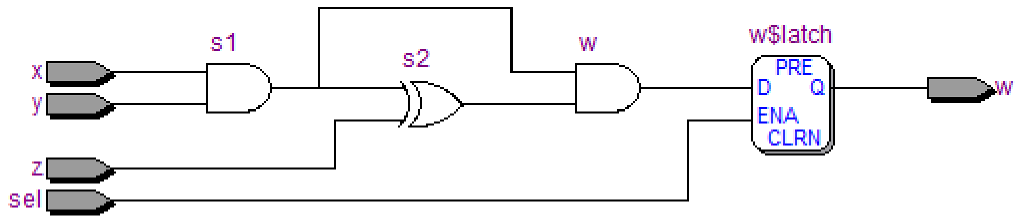
- An **if** statement executes a block of sequential statements upon matching certain condition(s).
- It can also include an **else** component or one or more **else if** components.
- *if* statements are only allowed within a procedural block. The statements are executed sequentially if a conditional match is detected.
- To avoid inferred latches, all outputs should be assigned values on all execution paths.
- Keep common statements outside of the *if- else if* statements.

Example: Inference

```
module if_example (input x, y, z, sel,
                   output w);

    always @ (x or y or z or sel)
    begin: inference
        logic   s1, s2;

        if (sel == 1'b1)
        begin
          s1 = x & y;
```

```
                s2 = s1 ^ z;
                w <= s1 & s2;   //Since w gets a value only conditionally, a latch is
              end                 // inferred.  Add an "else" statement or default w
        end                       // right after "begin" to avoid latches
endmodule
```

RTL Synthesis:



- To avoid the inferred latch, assign a default value to w outside the *if* statement. For example, you can add "w <= 0;" before the *if* statement. You can also add the same statement in the *else* part using an *if- else* statement. You will have to do the same thing even when using *always_comb*.

Case Statement:

- Equivalent to nested set of *if-else if* constructs but produces less logic since it does not force priority order.
- It identifies mutually exclusive blocks of code.
- Since all possible values of select inputs must be covered, use the *default* clause.
- **case** statements have to be inside a module. It can be used to describe multiplexors.
- **casez** statements treat 'Z' as *don't cares*, where a match is asserted when the rest of the word matches.
- **casex** statements treat both 'X' and 'Z' as *don't cares*.

Example: '*case*' Statement

```
module case_example(input a, b, sel,
                     output c);

always @ (a or b or sel)
begin: case_process

   case (sel)
      1'b0 :  c = a;
      1'b1 : c = b;
      default : c = 1'b0;
   endcase
end
endmodule
```

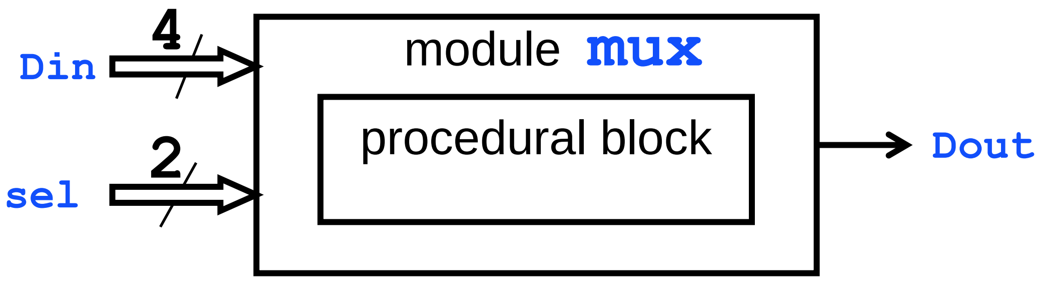
Example: '*casez*' Statement

```
module case_example(input a, b, sel,
                     output c);

always @ (a or b or sel)
begin: case_process

   casez (sel)
      1'z0 :  c = a;
      1'zx :  c = b;
      default : c = 1'b0;
   endcase
end
endmodule
```

Note: The '*case*' example to the left employs precise matching, that is, '*sel*' has to be strictly either the value '1'b0' or '1'b0' in order for '*c*' to receive '*a*' or '*b*'. As for the '*casez*' example to the right, both case items will ignore their own <*radix*> when comparing against '*sel*', since 'Z' is treated as *don't cares* here. But they still need to match their <*value*>. For example, a '*sel'* of '1'h0' will only match the first case item, and a '1'hF' will only match the second case item.

- Although *case* statement and *if-else if* statements are logically similar, SystemVerilog will actually interpret them into different physical circuits, as shown below

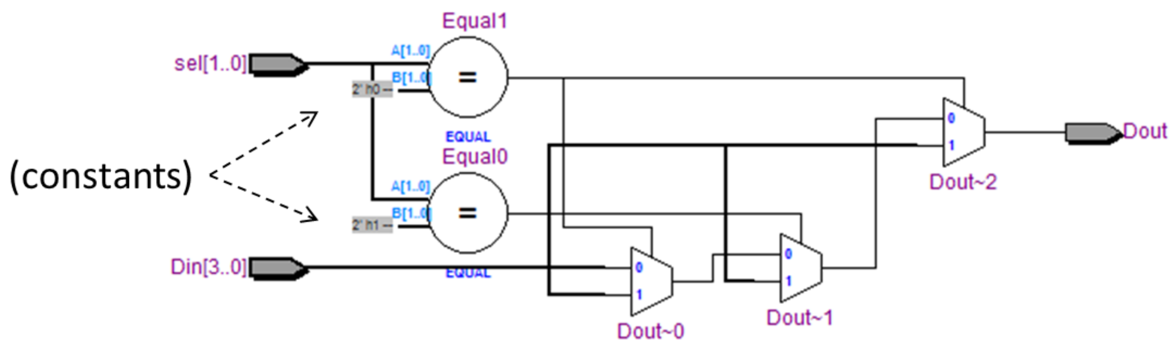
Example: 4-to-1 Multiplexer



Design 1:

```
module mux (input [3:0] Din,
            input [1:0] sel,
            output Dout);

    always @ (Din or sel)
    begin
        if (sel == 1'b00)
          Dout = Din[0];
        else if (sel == 1'b01)
          Dout = Din[1];
        else if (sel == 1'b10)
          Dout = Din[2];
        else
          Dout = Din[3];
    end
endmodule
```

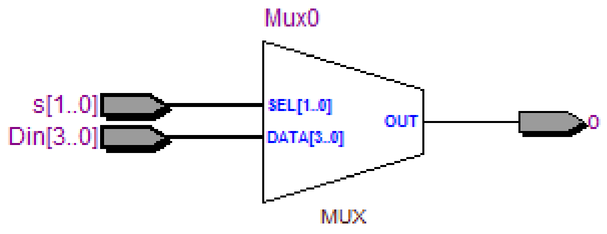
RTL Synthesis:



Design 2: (Produces less logic)

```
module mux (input [3:0] Din,
            input [1:0] sel,
            output Dout);

    always @ (Din or sel)
    begin
        case (sel)
          2'b00   : Dout = Din[0];
          2'b01   : Dout = Din[1];
          2'b10   : Dout = Din[2];
          default : Dout = Din[3];
         endcase
    end
endmodule
```

RTL Synthesis:



- Notice how the *case* statement gives a cleaner circuit with less logic. In general, although there are many different syntactic approaches to achieve the same logical functionality, most often one will be better than the other due to the translation of the circuit. Therefore it is somewhat important to try to write SystemVerilog in a nice, organized way.

Creating Hierarchical Design Using Components:

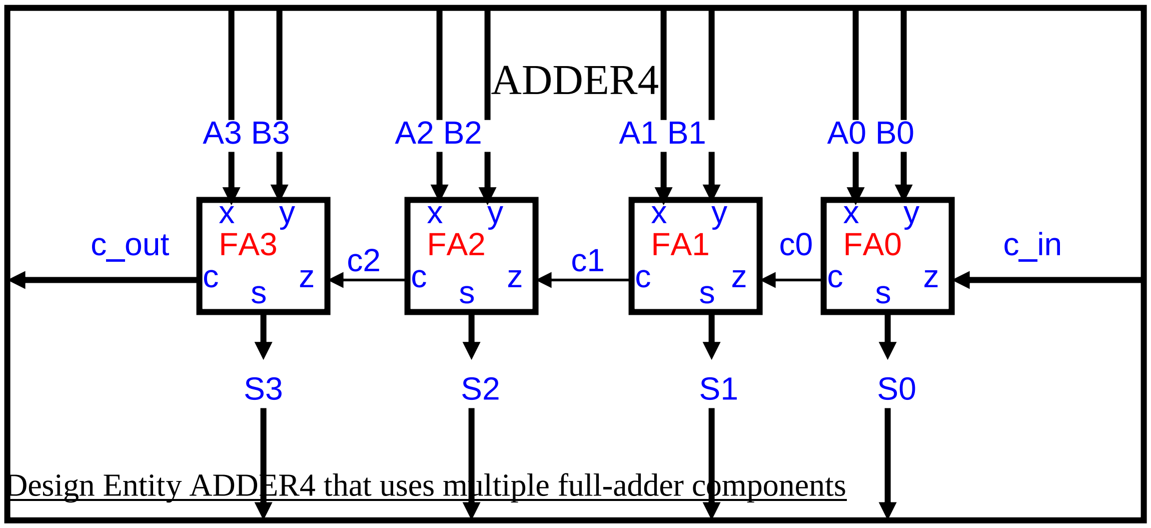
As mentioned earlier, you can create larger designs using modules as the components that make up the design. One module can instantiate other modules as components of it. One module can use different modules as components as well as instantiate multiple copies of the same module.

Example: Creating a 4-bit adder using a full-adder component

Design Entity Full-Adder

```
module full_adder  (input x, y, z,
                    output s, c);

        assign s = x^y^z;
        assign c = (x&y)|(y&z)|(x&z);

endmodule
```

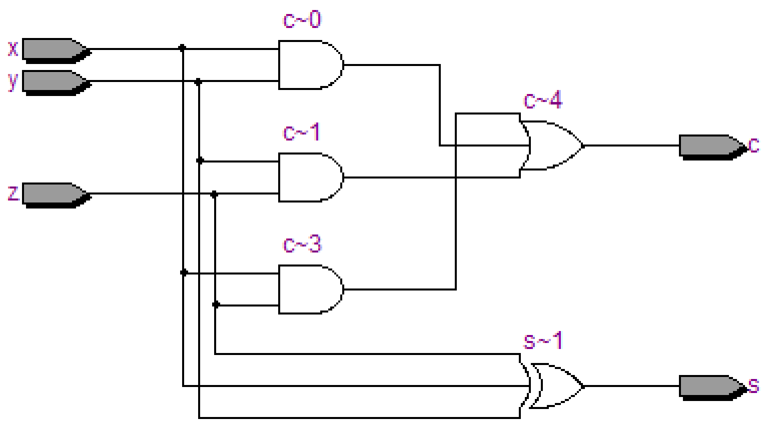
Use component full_adder to create a 4-bit adder:

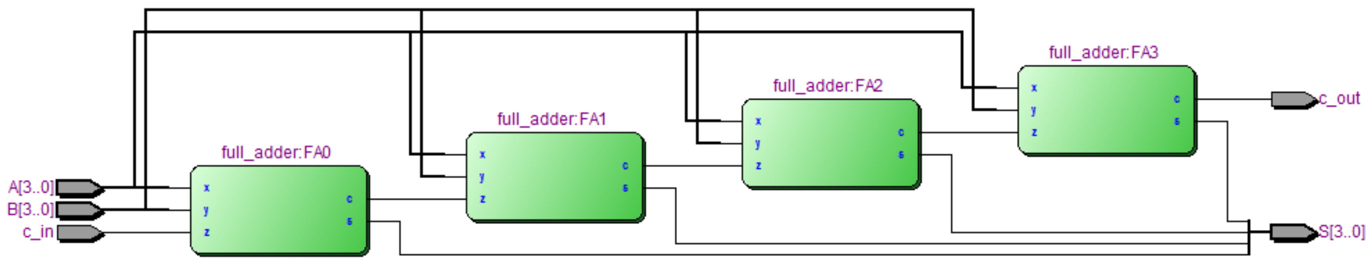


Design Entity ADDER4 that uses multiple full-adder components

```
module ADDER4 (input    [3:0]  A, B,
               input           c_in,
               output   [3:0]  S,
               output          c_out);

        // Internal carries in the 4-bit adder
        wire            c0, c1, c2;

        /*=============================================*/
        // Netlists with named (explicit) port connection
        // Syntax: <module> <name>(.<parameter_name> (<connection_name>), …)
        full_adder      FA0 (.x (A[0]), .y (B[0]), .z (c_in), .s (S[0]), .c (c0));
        full_adder      FA1 (.x (A[1]), .y (B[1]), .z (c0), .s (S[1]), .c (c1));
        full_adder      FA2 (.x (A[2]), .y (B[2]), .z (c1), .s (S[2]), .c (c2));
        full_adder      FA3 (.x (A[3]), .y (B[3]), .z (c2), .s (S[3]), .c (c_out));

endmodule
```

RTL Synthesis:





To simplify the port connections and coding redundancy, SystemVerilog introduces implicit ".name" and ".*" port connections. This is based on the fact that most of the port connections share the same port names on each end of the connection. For the ".name" implicit port connection, SystemVerilog automatically connects ports that share the same name and size with the connecting **wire**. For the ".*" port connection, SystemVerilog automatically connects ports that share the same name and size without the need of rewriting them.

Example: .name connection

```
module top_level(input clock, reset, in,
                 output out);
```

```
    wire        tmp;
    wire [2:0] data;


    // module1 myModule1(input clock, reset, in, output result, output [2:0] data)
    module1 myModule1(.clock, .reset, .in, .result(tmp), .data(data));
    // module2 myModule2(input clock, reset, feed, output [1:0] data, output out)
    module2 myModule2(.clock, .reset, .feed(tmp), .data(data[1:0]), .out);

endmodule
```

Note: The example above employs implicit port connection. If the name and size of the connections match (‘*clock*’, ‘*reset*’, ‘*in*’ and ‘*out*’ port in this case), then the matching ports will be automatically connected. However, if the name (‘*result*’/’*feed*’ ports) or the size (‘*[2:0]data*’/‘*[1:0]data*’ ports) of the connecting ports doesn’t match, we will need to write out the connection explicitly by adding a parenthesis behind the port name and indicate the specific wiring.

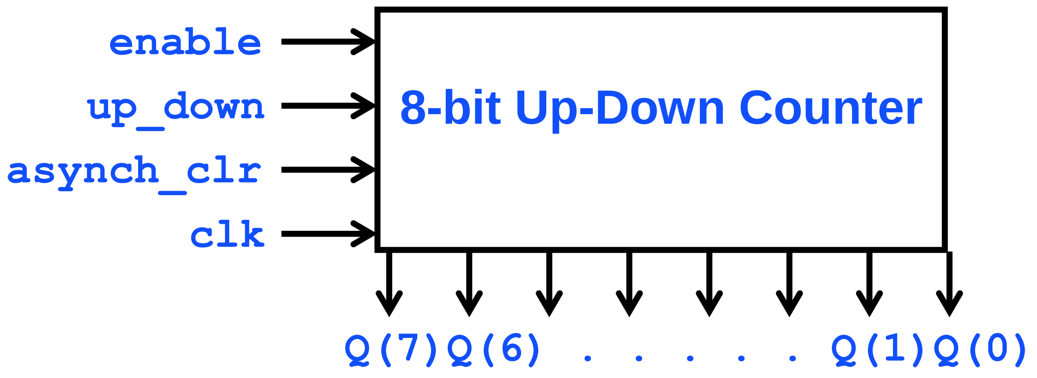
<u>Example</u>: .* connection

```
module top_level(input clock, reset, in,
                 output out);

    wire        tmp;
    wire [1:0] data;

    // .* “Wild card” port connection.  Only need to write out the connection if the
    //   name or size doesn’t match
    module1 myModule1(.*, .result(tmp));
    module2 myModule2(.*, .feed(tmp));

endmodule
```

Note: In both implicit connections, an unconnected port must be explicitly named with blank connection. Ex: .<*port_name*>()

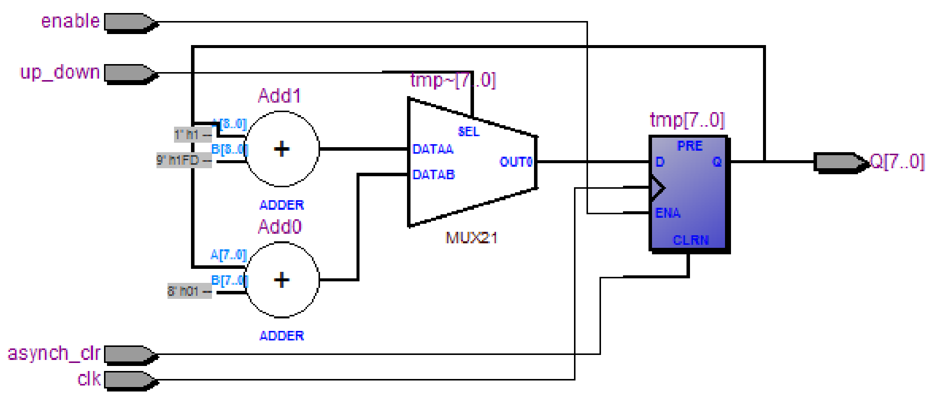
More Examples:

<u>Example</u>: Sequential Circuit

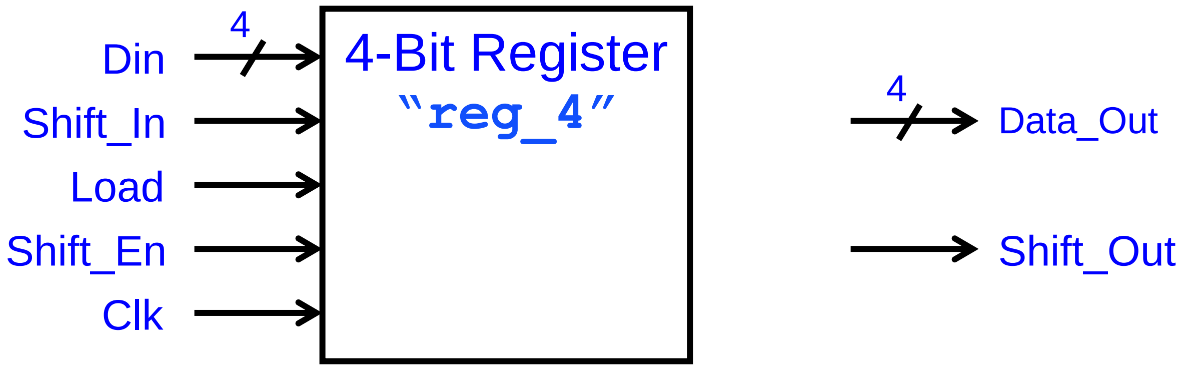


```
module counter (input clk, asynch_clr, enable, up_down,
                  output  [7:0]  Q);

        always @ (posedge clk or posedge asynch_clr)
/* Placing "posedge asynch_clr" in the sensitivity list enables clear whenever
   asynch_clr is on a rising edge.  If "posedge asynch_clr" is not in the sensitivity
   list, then the circuit will only be cleared if asynch_clr is high during a rising edge
   of the clk, i.e., the circuit will be synchronously cleared */

        begin
                if (asynch_clr)
                  Q <= 8'b0;
                else if (enable)
                  if (up_down)
                    Q <= Q + 1'b1;
                  else
                    Q <= Q - 1'b1;
        end

endmodule
```

RTL Synthesis:

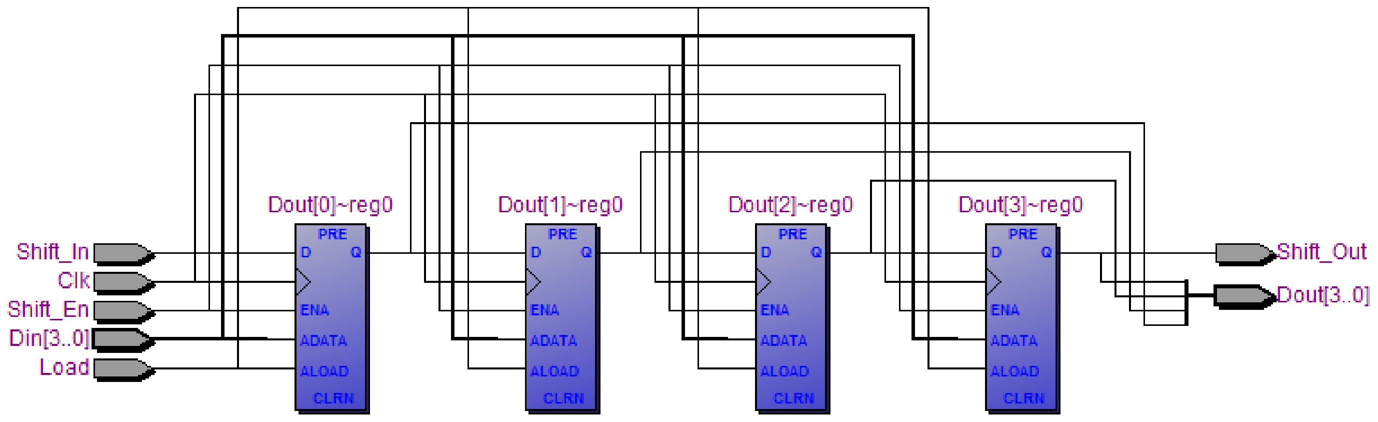


Example: N-Bit Left Shift Register

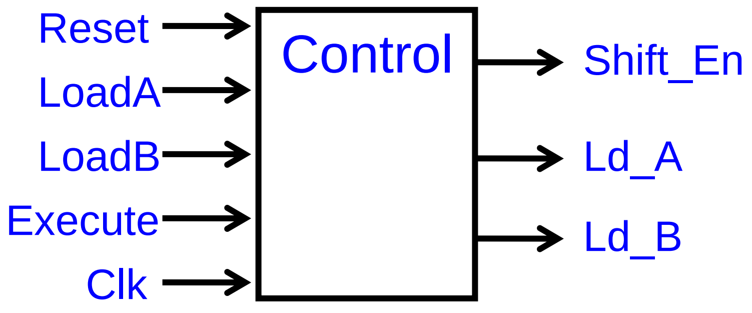


```
// #(N=4) is a variable parameter
module shiftReg #( N = 4) (input            Clk, Shift_En, Shift_In, Load,
                           input  [N-1:0]   Din,
                           output [N-1:0]   Dout,
                           output           Shift_Out);

        always_ff @ (posedge Clk or posedge Load)
        begin
                if (Load)
                  Dout <= Din;
                else if (Shift_En)
                  Dout <= { Dout[N-2:0], Shift_In };     // Concatenation
        end

        assign Shift_Out = Dout[N-1];

endmodule                      // shiftReg
```

RTL Synthesis:

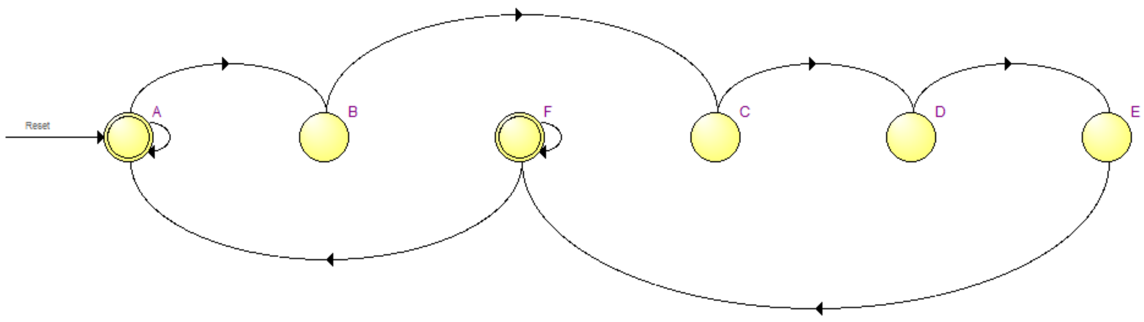


State Machine Design:
Example: Control Unit



```
module control (input  Clk, Reset, Execute, LoadA, LoadB,
                output Shift_En, Ld_A, Ld_B);

        enum logic [2:0] {A, B, C, D, E, F}   curr_state, next_state;   // Internal state logic

        // Assign 'next_state' based on 'state' and 'Execute'
        always_ff @ (posedge Clk or posedge Reset )  // Sequential logic
        begin
                if (Reset)
                  curr_state <= A;   // alternatively use the enum method
                else                 //   <enum_name>.first to point to the first enum value
                  curr_state <= next_state;
        end

        // Assign outputs based on 'state'
        always_comb
        begin
                // Default to be self-looping (stay in current state unless triggered), so
                //  therefore no potential latch will be inferred
                next_state  = curr_state;

        // The keyword "unique" asserts that there are no overlapping cases.  It will
        //    generate warnings when multiple case item expressions are true at the
        //    same time
                unique case (curr_state)
                  // use <enum_name> = <enum_value>  to assign the next state, or
                  //     use <enum_name>.next to go to the immediate next state
                  A :    if (Execute)              // conditional state transition
                            next_state = B;
                  B :       next_state = C;       // deterministic state transition
                  C :       next_state = D;
                  D :       next_state = E;
                  E :       next_state = F;
                F :      if (~Execute)
                           next_state = A;
            endcase
    end

    // Assign outputs based on 'state'
    always @ (LoadA, LoadB, curr_state)
    begin
            case (curr_state)
                A:
                  begin
                    Ld_A = LoadA;
                    Ld_B = LoadB;
                    Shift_En = 1'b0;
                  end
                F:
                  begin
                    Ld_A = 1'b0;
                    Ld_B = 1'b0;
                    Shift_En = 1'b0;
                  end
                default:
                  begin
                    Ld_A = 1'b0;
                    Ld_B = 1'b0;
                    Shift_En = 1'b1;
                  end
            endcase
    end

endmodule
```

RTL Synthesis:



Verification:

After designing and implementing your circuit in SystemVerilog, the next step is to make sure that it works according to the design requirement free from bugs through the verification process. A simulation instantiates the top level module of your circuit and does the following things:

- Generates the input vector waveform
- Apply the input waveform to the DUT (your circuit)
- Outputs the response

Once the circuit response is obtained, you can then verify it to see if it complies with the design requirement. If any response does not comply properly, you should trace back the specific faulty response all the way back to where the design flaw is generated and try to fix it. You should always verify your circuit as complete as possible to avoid unexpected flaws.

There are two ways to verify your circuit. The first one is using the SystemVerilog testbench module. SystemVerilog is a very powerful verification language, with constructs borrowed from the software languages. The compiler will generate the input signals according to the constructed testbench module, and the simulated result will be outputted. The testbench approach is especially useful when dealing with large and complicated circuits, which usually requires the generation of numerous inputs and the verification of numerous outputs.

If your circuit is simple and straightforward, then using the GUI waveform generator approach might be more convenient. Many HDL compilers come with built-in or custom-made GUI waveform generators which allow the designers to directly generate the input signals and view the output signals graphically. The Quartus II compiler we will be using employs such a custom-made GUI waveform generator. Please refer to IQT. 6-9 for a tutorial on the ModelSim-Altera GUI verification, and IQT. 29-34 for a tutorial on using testbenches in ModelSim-Altera.

Other Notes/Hints:

- A design that simulates is not guaranteed to synthesize. Simulation tools allow simulating designs that may not be physically possible to implement.
- It is always a good idea to assign values to outputs in all possible cases for any diverging statement. This way we can avoid inferred latches. If we do not specify the value of a signal for some input combination, then the design will have inferred latch(es) because it will try to hold the old value of the signal.
- Keep in mind that HDL statements are usually executed concurrently and not sequentially.
    - When a value is assigned to a signal, it does not take effect until the next simulation cycle, unless you specify it to be updated immediately, i.e., through the blocking statement.
- Include all inputs that can change the outputs in the sensitivity list of a procedure to avoid different circuit behavior from simulation and synthesis.
    - You can manipulate when the procedure executes by including/excluding certain inputs from the sensitivity list, but that will only work for the simulation model. If the synthesized design is combinational, then the logic will respond to events on any inputs, not just the inputs in the sensitivity list.

- SystemVerilog allows delay/wait statements, but that is only guaranteed during simulation. The synthesized design may not have similar delays.
    - For example: #5 x <= y; // delay x <= y by 5 time units.
        - This is allowed, but synthesis behavior is not guaranteed.
- SystemVerilog compilers generally allow looping statements (**for**, **while**, **do**…**while**, …), but synthesis tools will only be able to synthesize it if the statement can be unrolled and determined during the synthesis, i.e., the looping statements are to save repetitive statements and are not intended to execute non-determined cyclings.
- Do not specify initial values in variable declarations. Synthesis compilers cannot synthesize the **initial** procedural block. Initial values can be assigned explicitly under a controlling signal (e.g. reset).
- Assign *don't care* values to signals when possible for default cases. It may allow the synthesis compiler to produce a better design.
- Case statements produce less logic than *if-else* statements since *if-else* statements produce priority logic also.
- Do not assign value to the same signal via multiple statements that can execute simultaneously. The compiler will either handle only the last assignment or produce an error.
- Write your code for synthesis rather than just simulation, as you will need to synthesize your designs for all experiments involving SystemVerilog.

References:

[1] "SystemVerilog 3.1a Language Reference Manual, Accellera's Extensions to Verilog", Accellera, 2004. Available at: http://www.eda.org/sv/

[2] S. Sutherland, S. Davidmann and P. Flake, *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*, 2nd ed. Springer, 2006.

[3] Yalamanchili, Sudhakar. *Introductory VHDL – From Simulation to Synthesis*. New Jersey: Prentice-Hall, 2001.

[4] "A Proposal for a Standard Synthesizable Subset for SystemVerilog-2005: What the IEEE failed to Define", by Stuart Sutherland. presented at DVCon, March 2006. Available at http://www.sutherland-hdl.com/papers/2006-DVCon_SystemVerilog_synthesis_subset_paper.pdf