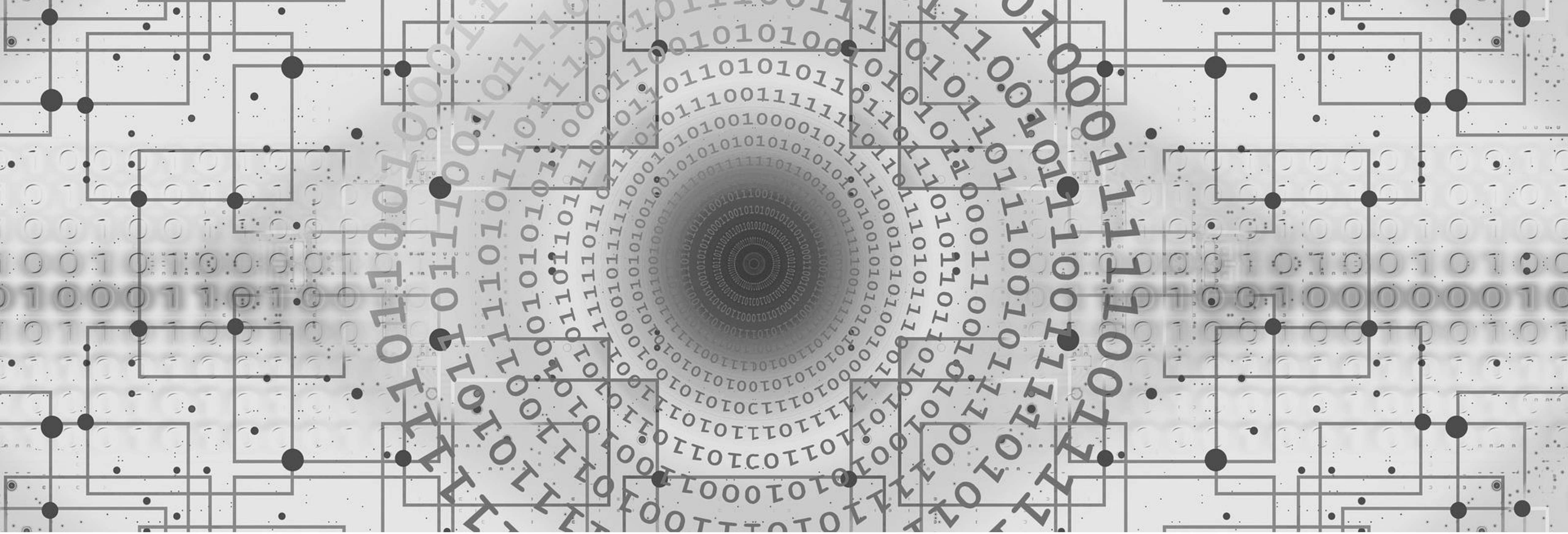


Lecture-5 : SystemVerilog Data Types, Continuous Assignment Statement and Conditional Operator

ECE-111 Advanced Digital Design Project

Vishal Karna

Summer 2025

UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

SystemVerilog Data Types

- SystemVerilog data can be specified using two properties :
  - Data kind and data type.
  - Data kind refers to usage as net type or variable type.
  - Data type refers to possible values a data kind can take.

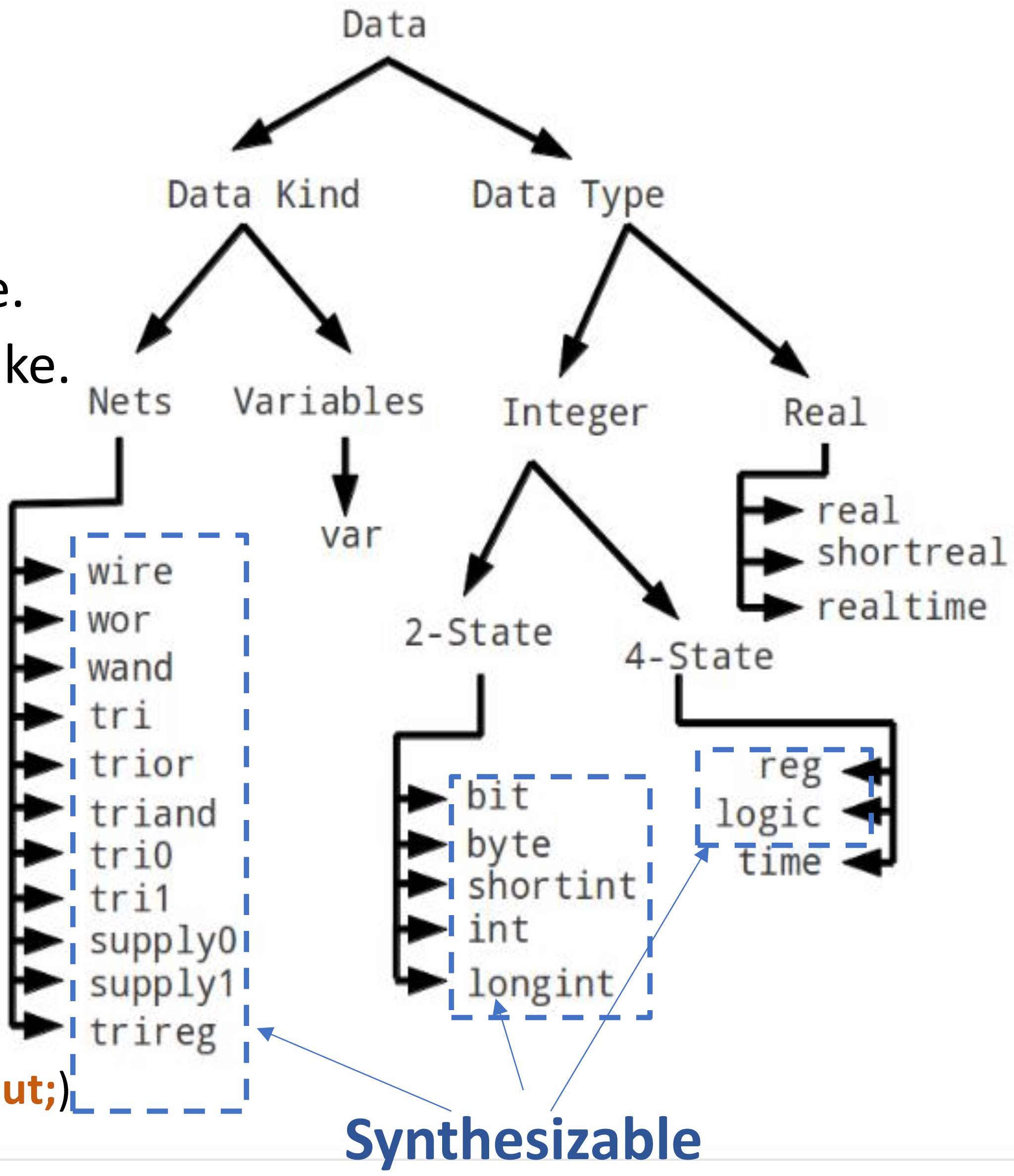
- Any data can be declared using the syntax:

  <data_kind> <data_type> <literal_name>

  example, var logic carry_out;

  wire logic [2:0] sum;

  Note :
  - **carry_out** is *declared as a* ***variable*** *which can store 4-state values*
  - **var** specification before data type is optional (logic carry_out;)
  - **sum** is declared as a **net** which can store 4-state values
  - **logic** specification after **wire** is optional (wire [2:0] sum;)



SystemVerilog Data Types

- SystemVerilog data types are divided into two main groups:
  - **Nets** and **Variables**
  - Distinction comes from how they are intended to represent different hardware elements

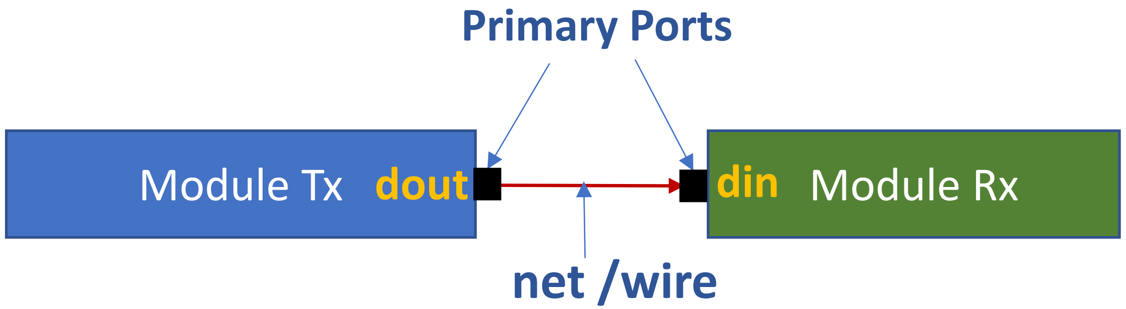
- **Nets** and **Variables** both have a data type
  - Data Type indicates value system of the net or the variable,
    - A **net** or **variable** is either 2-state (0,1) or 4-state (0,1,X,Z)
    - Example : **wire** is a 4-state **net**, **reg** and **logic** are 4-state **variables**, **bit** is a 2-state **variable**

- Data types are used by simulators and synthesis compilers to determine how to store and process changes on that data
  - Store as 4-state or 2-state value

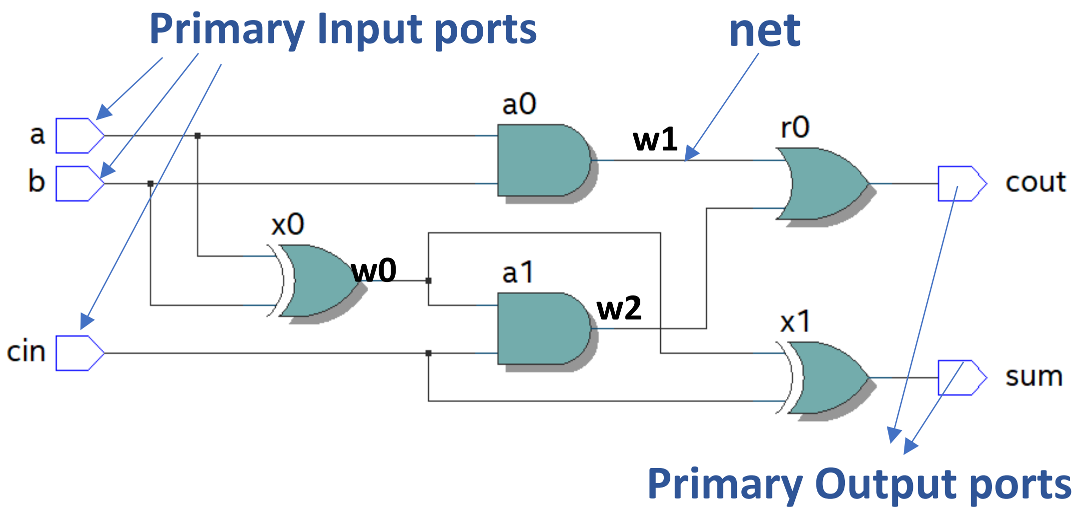
- Data types are used in RTL modeling to indicate desired silicon behavior, such as
  - **Example,** ALU should be integer based or floating-point based

Net

- **Net represents a physical connection between structural entities, such as between gates or between modules.**
  - Think like a plain wire connecting two elements in a circuit.



Module Tx dout port is connected to din port of Module Rx through a net (wire)



- **Net does not store value**
- **Net is continuously driven.**
  - Its value is derived from what is being driven from its driver(s)
- **wire is probably the most common type of a Net**
  - some other net data kinds are **tri, wand, wor, supply0, supply1, triand, trior.**

Rules for Wire Data Kind

```
module example_wire_reg
  ( input    wire  A, B, C,  // 1-bit input wire
    output  reg d,   // 1-bit reg variable
    output  wire result);

 always@(B or C) begin
    d = B | C;  // LHS has to be either reg or logic. "d" cannot be a wire data kind !
 end

 assign result = A & B;  // LHS of assign should be either wire or logic. "result" can be either a wire or logic data kind !
endmodule: example_wire_reg
```

- ❑ **wire elements are used as inputs and outputs within an actual module declaration.**
- ❑ **wire elements must be driven by something, and cannot store a value without being driven.**
- ❑ **wire elements cannot be used as the left-hand side of an "=" or "<=" statement within an always block.**
- ❑ **wire and logic elements are the only legal type on the left-hand side of an assign statement.**
- ❑ **wire elements can only be used to model combinational logic !**

Variable

- **Represents data storage elements in a circuit**
  - Provides temporary storage for simulation, does not mean actual storage in silicon
  - It holds last value assigned to it until the next assignment

- **reg is probably the most common variable data type**
  - **reg** is generally used to model hardware registers
    - Although **reg** can also represent combinatorial logic, like inside an **always@(*)** block).
  - **reg** default value is **'X'**
  - **Must be used when modeling sequential elements such as shift registers, etc !**

- **Synthesizable variables in SystemVerilog are :**
  - **logic, reg, bit, byte, integer, shortint, int, longint -- Can be used in design code and in testbench code !**

- **Non-synthesizable variables in SystemVerilog are :**
  - **real, shortreal, time, realtime -- Can only be used in testbench code not in design code !**

Rules for Reg Data Type

```
module example_wire_reg
  ( input   wire  A, B, clock, d,  // 1-bit input wire
    output  reg q,   // 1 bit reg variable
    output  wire result);

 always@(posedge clock) begin
    q = d;  // LHS has to be either reg or logic. "q" can be either reg or logic and it cannot be a wire data kind !
 end

 assign result = A & B;  // LHS of assign should be either wire or logic. "result" cannot be a reg data kind in Verilog !
endmodule: example_wire_reg
```

- **reg elements can be used as outputs within an actual module declaration.**
- **reg elements cannot be used as inputs within an actual module declaration.**
- **reg and logic is the only legal type on the left-hand side of "=" or "<=" statement with an always block**
- **reg cannot be used on the left-hand side of an continuous assign statement.**
- **reg can be used to create registers when used in conjunction with always@(posedge clock) blocks. Therefore, reg be used to create both combinational and sequential logic !**

Replace wire and reg data types with logic in SystemVerilog !

```
module example_logic
  ( input    logic  A, B, clock, d,  // language will automatically infer these inputs as a wire !
    output  logic q, // language will automatically infer output “q” as a reg !
    output logic result // language will automatically infer output “result” as a wire !
  );
  always@(posedge clock) begin
    q = d;  // language will automatically infer logic “q” as a reg !
 end

 assign result = A & B;  // language will automatically infer output logic “q” as a wire !
endmodule: example_logic
```

- In SystemVerilog design modeling use logic everywhere in place of a wire and a reg
  - **Pass the burden to language compiler/simulator to infer correct data type internally !**
- logic elements can be used as inputs, outputs, inouts within an actual module declaration.
- logic can be used on left-hand side of “=“ or “<=“ statement with an always block
- logic can be used on the left-hand side of a continuous assign statement.
- logic can be used to model both combinational and sequential hardware logic elements

Summary on Wire, Reg and Logic Data Types

wire

- **wire is** used for connecting different modules and other logic elements within module
- Wire elements must be continuously driven by something and it cannot store value. It can be both driven and read.
- **wire** data type is used on left hand side (LHS) in the continuous assignments and can be used for all types of ports.

reg

- **reg** is a data storage element. Just declaring variable as a reg, does not create an actual register but it can store values
- **reg** variables retains value until next assignment statement.
- **reg** data type variable is used on left hand side (LHS) of blocking/non-blocking assignment statement inside in am always blocks and in output port types

logic

- **logic** is an extension of **reg** data type. It can be driven by both continuous assignment or blocking/non blocking assignment (**=** and **<=**).
- **logic** can also be used in all type of port declarations (**input, output and inout**)
- **logic** was introduced in **SystemVerilog** and not support in older **Verilog!**

SystemVerilog and Verilog Data Types

Type	Mode	State	Size	Sign	SV/Verilog	Representation
reg	integer	4-state	user-defined	unsigned	Verilog	Equivalent **to var logic**
logic	integer	4-state	user-defined	unsigned	SystemVerilog	Infers a **var logic** except for input/inout ports **wire logic** is inferred
bit	integer	2-state	user-defined	unsigned	SystemVerilog	default 1-bit size
byte	integer	2-state	8-bit	signed	SystemVerilog	Equivalent to **var logic**[7:0]
integer	integer	4-state	32-bit	signed	Verilog	Equivalent to **var logic**[31:0]
shortint	integer	2-state	16-bit	signed	SystemVerilog	Equivalent to **var bit**[15:0]
int	integer	2-state	32-bit	signed	SystemVerilog	Equivalent to **var bit**[31:0]. Synthesis compilers treats as 4-state **integer** type
longint	integer	2-state	64-bit	signed	SystemVerilog	Equivalent to **var logic**[63:0]
real	floatingpoint	2-state	-	-	Verilog	Cannot be synthesized
shortreal	integer	2-state	-	-	SystemVerilog	Cannot be synthesized
realtime	floatingpoint	2-state	-	-	Verilog	Cannot be synthesized
time	Integer	4-state	64-bit	unsigned	Verilog	Cannot be synthesized

Continuous Assignment

- **Continuous assignment statement drives a right-hand side (RHS) expression onto a net or a variable in left-hand side (LHS)**
  - Continuous assignment statements RHS expression evaluation starts from simulation time 0 and continues until end of the simulation !

Syntax :

```
assign #(delay) net or a variable = expression;
```

Example :

```
wire a, b, c, d;
assign c = a + b; // assignment to 'd' happens immediately at current simulation time
assign #2 d = a - b;  // assignment to 'd' is delayed by two-time units
```

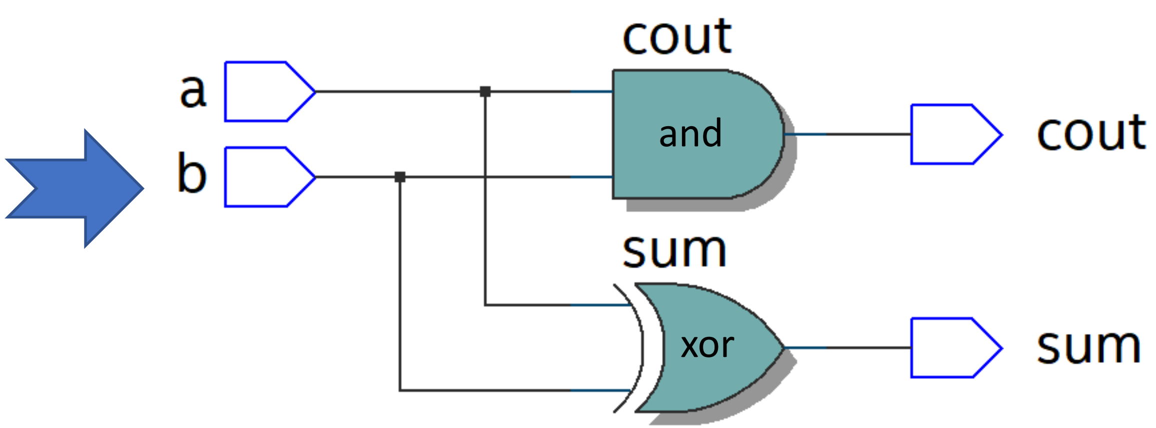
- It is called **continuous assignment** because in example above, wire "**c**" is **continuously** updated whenever **a** or **b** changes. Any change in **a** or **b** will result in change in **c**.
- This can be used for modeling combinational logic
- The assignment may be delayed by the specified amount
  - Synthesis compiler ignores the delay if specified, since it expects zero-delay RTL models
  - If no delay is specified, the assignment happens at the current simulation time
- Verilog required LHS of an **assign** to be a **net** and not a **variable**. SystemVerilog allows both in LHS !

Continuous Assignment

- **Module can contain any number of continuous assign statements and each assign statement runs concurrently.**
  - These multiple assign statements are not executed in any specific order with respect to each other
  - Changing order of multiple continuous statement within module has not implication in synthesis results
  - **Example** : Half adder with multiple continuous assignment statements

```
module half_adder(
  input logic a, b,
  output logic sum, cout
);
  // multiple continuous assign statements
  assign sum = a ^ b;
  assign cout = a & b;

endmodule: half_adder
```

Both assign statements will run in parallel



Half Adder

Continuous Assignment

- There are two types of continuous assignment statement :
  - **Explicit continuous assignments**
    - Example : **assign** sum = a + b; // **assign** keyword is explicitly specified
    - Supports both net and variable on **LHS**
  - **Implicit net declaration continuous assignments**
    - Example : **wire**[2:0] sum = a + b; // continuous nature is inferred even though **assign** is not explicitly specified
    - Implicit continuous assignments can only have nets on **LHS**

- Continuous assignment statement cannot be used in initial block and always procedural block
  - **initial** block runs only once during simulation, it exits once "end" statement is hit whereas **always** block can runs continuously (or multiple times)

```
always@(a,b) begin
  assign sum = a + b;
end
```

✗ assign statements within always procedural block is not allowed

```
Initial begin
  assign sum = 0;
end
```

✗ assign statements within initial procedural block is not allowed

- Continuous assignment however it can be inferred if used in always procedural block.

```
always@(a or b) begin
  sum = a + b;
end
```

} Behaves like continuous assignment statement

=

```
// if a or b value changes then result of a + b is assigned to sum
assign sum = a + b;
```

Continuous Assignment

- Synthesis compiler will give error when same variable is driven in both always@ procedural block and driven by continuous assignment statement.

```
module illegal_usage(
  input logic a, b,
  output logic c
);
  assign c = a ^ b;
  always@(a,b) begin
   c = a + b;
  end
endmodule: mux
```

❌ logic ‘c’ cannot be assigned from both always block and through assign statement since this will result in multiple driver on net ‘c’

- Synthesis compiler will give error when same variable is driven from multiple continuous assignment statements

```
module illegal_usage(
  input logic a, b, q,
  output logic c
);
  assign c = a ^ b;
  assign c = b | q;
endmodule: mux
```

❌ logic ‘c’ cannot be assigned from multiple continuous assignment statements this will result in multiple driver on net ‘c’

Continuous Assignment

- **LHS** of continuous assignment statement can be :
  - Scalar, 1-bit, net or a variable or a user defined data type
  - Vector net or a variable
    - If **LHS** is a smaller vector size than RHS, then MSB's of the vector on RHS will be truncated to the size of vector on LHS.
    - If **LHS** is a larger vector size than RHS, then RHS vector will be extend with zero's in its MSB's.

Example :

```
wire[4:0] A, B; // packed array
wire[5:0] C, D; // packed array
wire E [4:0]; // unpacked array
// LHS is smaller width than RHS
assign A = C;  // MSB of wire C[4] will be truncated

// LHS is larger width to larger width
assign D = B; // '0' will get assigned to MSB of wire D[4]

assign E[0] = A[0]; // LHS cannot have unpacked array
```

- **LHS** of continuous assignment statement cannot be an unpacked structure or unpacked array
- **RHS** of continuous assignment statement can be an expression comprising of :
  - Nets, Variables (registers), Function call, Concatenation operations, Bit or Part selects

Continuous Assignment Statement

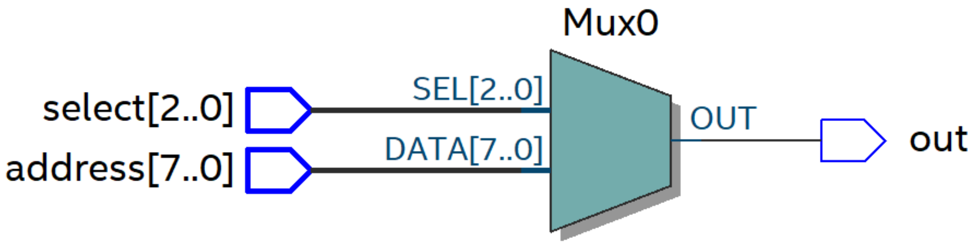
- RHS of continuous assignment statement can have function call

```
module ex_add(
  input logic[1:0] a, b, c,
  output logic[1:0] q
);

// Function add3
function logic[1:0] add3(input logic [1:0] x, y, z)
begin
  add3 = x + y + z;
end
endfunction

// Function add3 called on RHS of assign statement
assign q = add3(a, b, c);
endmodule: ex_add
```

Continuous Assignment

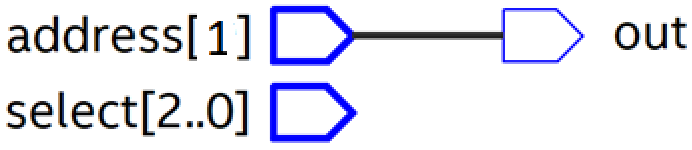
- If **RHS** expression has an array reference with **variable** index then synthesis compiler will generate a **mux**.

```
module mux(
  input logic[7:0] address,
  input logic[2:0] select,
  output logic out
);
 // non-constant index in address will result in a mux
  assign out = address[select];
endmodule: mux
```



- If **RHS** expression has an array reference with a **constant** index then synthesis compiler will generate just a **wire** and not a mux.

```
module simple_wire(
  input logic[7:0] address,
  input logic[2:0] select,
  output logic out
);
 // constant index in address will result in a wire
  assign out = address[1];
endmodule: simple_wire
```



Conditional Operator

- Widely used operator in RTL modeling. Also known as **Ternary** operator !!
- Similar to **if-else** statement
- Conditional operator often behaves like a hardware multiplexer
- Can be used in continuous **assign** statement and also within **always** procedural blocks

conditional operator	Syntax	Example Usage
? :	conditional expression **?** **true expression** **:** **false expression**	assign out= p **?** a **:** b

If **"p"** is true then assign value of **"a"** to **"out"** otherwise assign value of **"b"** to **"out"**

- Conditional expression listed before **"?"** is evaluated first as true or false
  - If evaluation result is true, then **true expression** is evaluated
  - If evaluation result is false, then **false expression** is evaluated
  - If evaluation result is unknown "x", then conditional operator performs bit by bit comparison of the two possible return values
    - If corresponding bits are both 0, a 0 is returned for that bit position
    - If corresponding bits are both 1, a 1 is returned for that bit position
    - If corresponding bits differ or if either has "x" or "z" value, an "x" is return for that bit position

Conditional Operator

- **Example :**

  logic sel, mode;

  logic [3:0] a, b, mux_out;

  assign mux_out = (sel **&** mode) **?** a **:** b;

Scenario	Value of "sel"	Value of "mode"	Value of "a"	Value of "b"	Result of conditional expr (sel & mode)	Final value assigned to "mux_out"
1	1'b1	1'b1	4'b0101	4'b1110	True (1)	4'b0101
2	1'b0	1'b1	4'b0101	4'b1110	False (0)	4'b1110
3	1'b1	1'bx	4'b0101	4'b1110	Unknown (x)	4'bx1xx
4	1'b1	1'bx	4'b011x	4'b0z10	Unknown (x)	4'b0x1x

Note : bitwise and ("**&**") will return value "x" if one of the operand is "x" and another operand is "1"

Conditional Operator

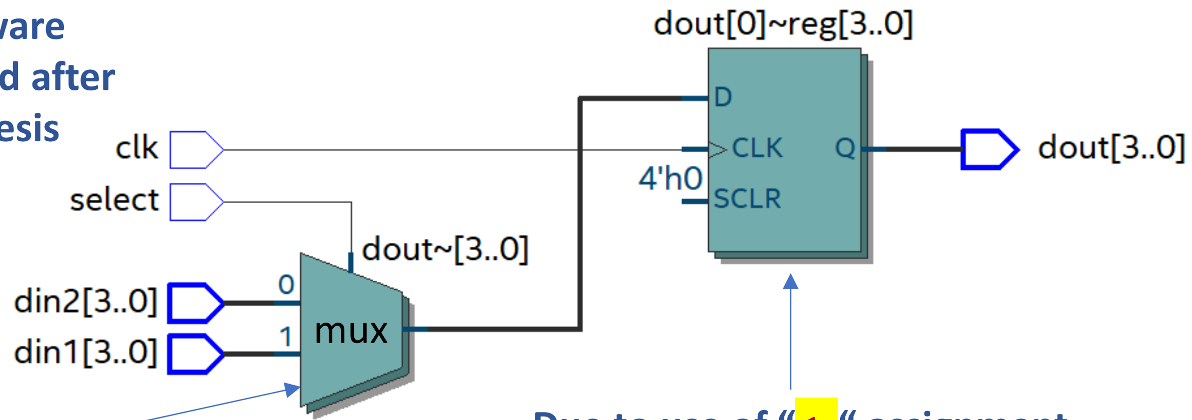
❑ Example : Conditional operator mapped to a multiplexor and a registered output

- Conditional operator to choose between two inputs for input to a register
- Conditional operator inside always block
- Synthesis compiler will map conditional operator to four multiplexers, one for each bit of din1 and din2 input and there will be 1 flipflop for each bit

```
module muxed_register
 #(parameter WIDTH=4)
 ( input   logic  clk,
   input   logic  select,
   input   logic [WIDTH-1:0]  din1, din2,
   output logic [WIDTH-1:0]  dout
 );

 // store din1 or din2 based on select value
 always@(posedge clk) begin
   dout <= select ? din1 : din2;
 end
endmodule: muxed_register
```

After synthesis of conditional operator, a multiplexer logic will be inferred

Hardware generated after synthesis



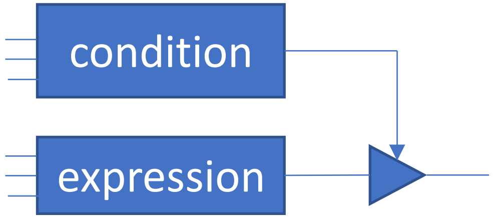
Due to use of "<=" assignment within always block and having posedge event on a signal "clk" in sensitivity list, synthesis tool inferred a FlipFlop

Conditional Operator

❑ Example : Conditional operator mapped to tri-state buffer

- Synthesis compiler will not always map conditional operator to a multiplexor
- Conditional operator can also be mapped to tri-state buffer based on how conditional operator is used and its operand data type and its values.

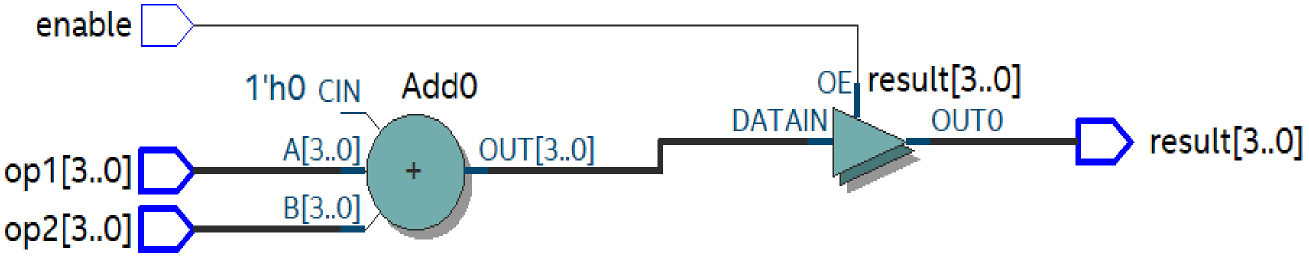
assign target = condition **?** expression **:** 1'bz;



Tri-state buffer inferred based on 1'bz assignment

```
module adder_with_tri_state_buffer
 #(parameter WIDTH=4)
  ( input   logic  enable,
    input   logic[WIDTH-1:0]  op1, op2,
    output logic[WIDTH-1:0]  result
  );

  // tri state buffer
  assign result = enable ? (op1 + op2) : 4'bz;
endmodule: adder_with_tri_state_buffer
```

Hardware generated after synthesis



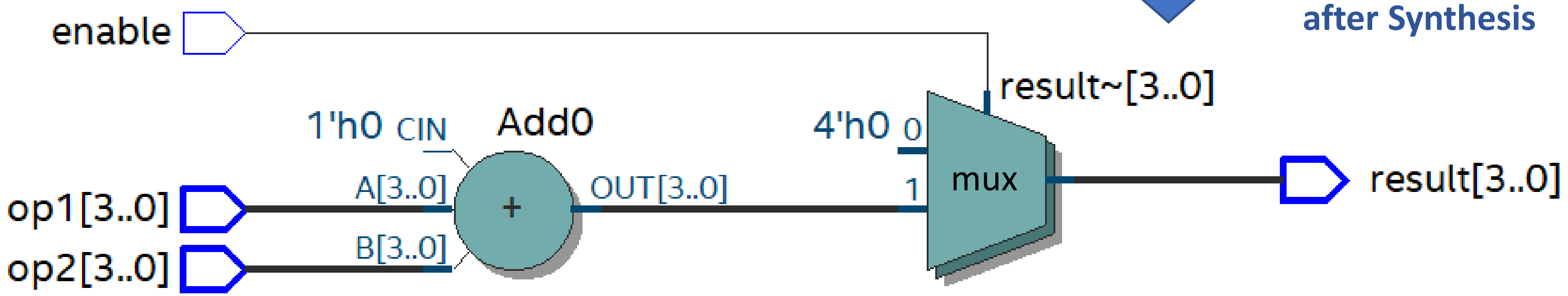
Conditional Operator

❑ Example : Conditional operator mapped to a mux

- Changing false expression value from 4’bz to 4’b0 synthesis compiler infers multiplexor instead of tri-state buffer

```
module adder_with_mux
 #(parameter WIDTH=4)
  ( input    logic  enable,
    input    logic[WIDTH-1:0]  op1, op2,
    output logic[WIDTH-1:0]  result
  );

  // multiplexor with adder output
  assign result = enable ? (op1 + op2) : 4’b0;  // changing 4’bz to 4’b0 will infer a mux
endmodule: adder_with_mux
```

Hardware generated after Synthesis



Self-Reading
Nets, Variables, Integer, Time, Real Data Types and Wand Data Kind

Verilog vs SystemVerilog Data Types

Verilog	System Verilog
❑ Strict about usage of wire and reg data type. (example : **wire** has to be used on **LHS** of continuous assignment statement and **reg** cannot be used for input and inout port declarations)	❑ Simplified usage by introducing **logic** data type which can be used for port and signal declaration. Replacing **reg** and **wire** usage.
❑ For synthesizable variables supports only **4-state** (**0,1,X,Z**) variables	❑ Synthesizable **2-state** (**0,1**) data type added
	❑ **2-state** variable can be used in testbench code where **X,Z** are not required
	❑ **2-state** variable in RTL Model improves simulation performance and 50% reduced memory usage as compared to **4-state** variables

Use of 2-State Variables with Caution !

- **Avoid all 2-state data types in RTL modeling**
  - **2-state data types can hide design bugs !**

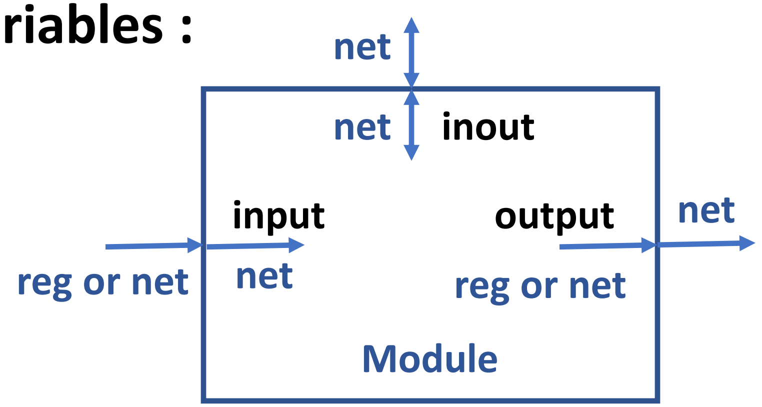
  - It can lead to **simulation** vs **synthesis** mismatches
    - Synthesis treats bit, byte, shortint, int and longint 2-state data types as a 4-state reg variable. Simulation treats as 2-State variable.
    - Simulation might start with a value 0 in each bit whereas synthesized implementation might power-up with each bit a 0 or 1.
    - Any x or z driven values on 2-state variables, are converted to 0. These data types are initialized to 0 at the start of simulation and may not trigger an event for active low signals.

  - **Exception** : use 2-state **int** data type variable for the iterator variable in for-loops where **X** and **Z** is not required.

Nets and Variables Inference

❑ **General Inference rules in SystemVerilog for nets and variables :**

- bit is 2-state variable
- wire is a 4-state net
- reg is a 4-state variable
- logic is a 4-state net and/or variable
- logic infers a net if used in input or inout port declaration
- logic infers a variable if used in output port declaration
- logic infers a variable if used to declare internal signals within module and does not have wire specified before logic
- reg cannot be used in input and inout port declaration. It can be used in output port declaration.
- default port datatype is wire if not declared explicitly when declaring ports
- default datatype of signals declared within module is a logic type variable if not explicitly defined
- var keyword before logic, bit and reg data type for internal signal declaration is optional since by default these three are variables.
- For nets declared as wire for internal signals within module, it is treated as a logic type net even though it is not explicitly specified.



Nets and Variables Inference

```
module top(

 // module ports with inferred types
  input in1,             // infers a 4-state net. Infers wire type net since net type is not explicitly declared
  input logic in2,       // infers a 4-state net. The logic infers a net if used in input or inout type port declaration
  input bit in3,         // infers a 2-state variable. The bit always infers a variable
  output out1,           // infers a 4-state net. Infers wire type net since net type is not explicitly declared
  output logic out2,     // infers a 4-state variable. The logic infers a variable if used in output type port declaration
  output bit out3        // infers a 2-state variable. The bit always infers a variable
  output reg out4        // infers a 4-state variable. The reg always infers a variable and it can only be used for output port types
);

  // internal signals with inferred and explicit types
  bit fault;             // infers a 2-state variable. The bit always infers a variable.
  logic d1;              // infers a 4-state variable. The logic infers a variable if used in signal declaration if not qualified with wire
  logic [3:0] d2;        // infers a 4-state variable.
  reg [7:0] d2;          // explicitly declares a 4-state variable.
  wire [2:0] w1;         // explicitly declares a net, infers 4-state logic
  wire logic [2:0] w2;   // explicitly declares a 4-state net
  var [3:0] d3;          // explicitly declares a variable, infers logic.
  var logic [3:0] d4;    // explicitly declares a 4-state variable. var specification is optional.

endmodule: top
```

Integer

- **Integer is a general purpose 4-state variable of register data type**
  - For synthesis it is used mainly for loops-indices, parameters, and constants.
  - They are of implicitly of type **reg**.

- Declared with keyword **integer**

- **Integer store data as signed numbers whereas explicitly declared reg types store them as unsigned**
  - Negative numbers are stored as 2’s complement

- **Size of integer is implementation specific (at least 32 bits)**
  - If they hold numbers which are not defined at compile time, their size will default to 32-bits

- **If they hold constants, the synthesizer adjusts them to the minimum width needed at compilation.**

- Example :

  integer data; //32-bit integer
  assign b = 31; //synthesizer will treat b as a 5-bit integer

Time

- **time** is a special 64-bit data type that can be used in conjunction with the **$time** system task to hold simulation time.

- Declared with keyword **time**

- **time** is not supported for synthesis and hence is used only for simulation purposes

- Syntax :
  time variable_name;

- Example :
  time start_t;
  initial
  start_t = **$time();**

Real

- SystemVerilog supports real data type constants and variables
  - Declared with keyword **real**
  - Real numbers are rounded off to the nearest integer when assigning to an integer.
  - Real Numbers can not contain 'Z' and ‘X’
  - Not supported for synthesis.

- Real numbers may be specified in either decimal or scientific notation
  - < value >.< value > : e.g 125.6 which is equivalent to decimal 125.6
  - < mantissa >E< exponent > : 2.5e4 which is equivalent to decimal 25000

- Syntax :

  real variable_name;

- Example :

real height;

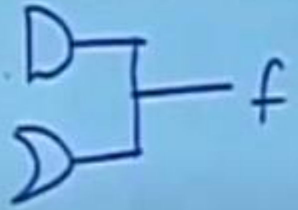
height = 50.6**;**

height = 2.4e6**;**

Wand Data Type Example

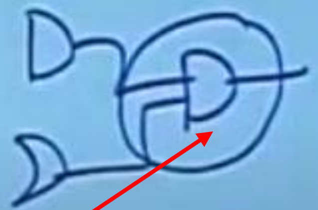
Synthesis tool will give design Error where wire 'f' is assigned from two continuous assign statements !

```
module  using_wire  (A, B, C, D, f);
    input    A, B, C, D;
    output  f;
    wire      f;             //  net f declared as 'wire'

    assign  f = A & B;
    assign  f = C | D;
endmodule
```



If wire 'f' is replaced below in code with wand 'f' now there will be no error from Synthesis tool.

```
module  using_wired_and  (A, B, C, D, f);
    input    A, B, C, D;
    output  f;
    wand    f;          // net f declared as 'wand'

    assign  f = A & B;
    assign  f = C | D;
endmodule
```



Synthesis tool will connect output of OR and AND gate and explicitly add AND gate at the output.