Lecture 6:
SystemVerilog Data Types, Continuous Assignment Statement and Conditional Operator

UCSD ECE 111

Prof. Farinaz Koushanfar

Fall 2024

Some slides are courtesy of Prof. Lin or Prof. Karna ECE111 courses

Important announcements (Details in Canvas)

- **Prof office hours this week:** Thurs 10/17 from 12-2pm or by email appointment
- **TA office hours:** are now MWF 9-11am for Fall'24
  - Zoom Meeting ID: 948 6397 0932; Passcode 004453
  - https://ucsd.zoom.us/j/94863970932?pwd=iXcQXbLYjOaAQTdOJlrmglCYMSyeir.1
- **TA Discussion session:** Wed 4-4:50PM (in person) Location: CNTRE 214
- **Oct 8:** Homework 2 was posted on Canvas
  - Due on Wednesday, **10/16/24**
  - Cloudlabs is now up and running so you should all have cloud access to SW
  -
- **Oct 15:** Homework 3 was posted on Canvas
  - Due on Wed Oct 23, **10/23/24**
  - **Late homework policy:** Max of 2 late days, with 20% grade reduction per day

Homework 3 overview

- You will learn how to:
  - Create synthesizable SystemVerilog code
  - Learn about primary ports
  - Better learn how to use testbenches
  - Design functional SystemVerilog code that can post synthesis.
- There will be two parts for this homework:
  - **Homework-3a**: Developing a Synthesizable SystemVerilog Model for a 4-bit Johnson Counter
  - **Homework-3b**: Developing a synthesizable SystemVerilog model of a 4-bit Universal Shift Register.

SystemVerilog Data Value Set

- For RTL modeling, SystemVerilog uses a four-value set to represent actual value that can occur in silicon
- These 4-Value Sets are described in the table below :

Value	Abstract State
0	▪ Represents an abstract digital low state
1	▪ Represents an abstract digital high state
Z	▪ Represents abstract digital high-impedance state. ▪ In multi-driver circuit, a value of 0 or 1 will override a Z
X	▪ Represents either uninitialized value, or uncertain value or a conflict of values in a multi-driver circuit

Note :

Values of '0', '1', and 'Z' are an abstraction of values that can exist in actual silicon
The value of 'X' is not an actual silicon value.
Simulators use 'X' value to indicate uncertainty in how physical silicon would behave under specific conditions.

System Verilog Data: Kind and Type

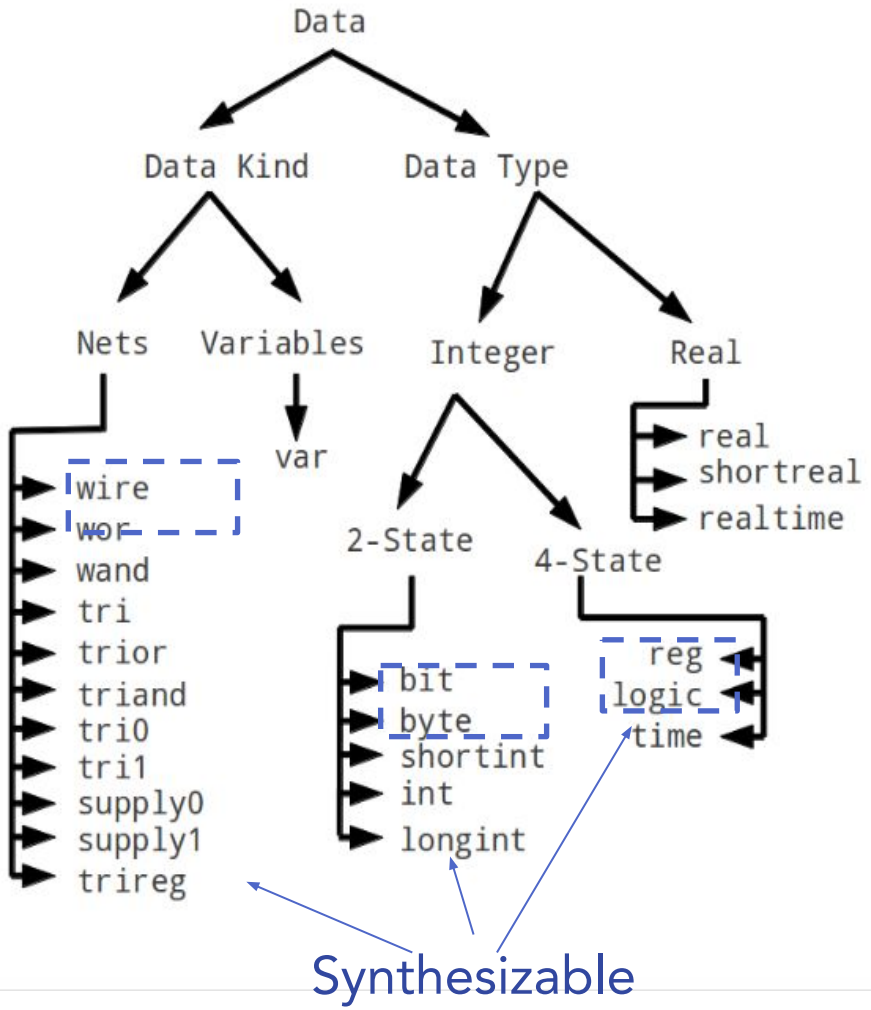
- SystemVerilog data can be specified using two properties : (i) Kind and (ii) Type
  - Data kind refers to usage as net or variable
  - Data type refers to possible values a data kind can take (e.g. integer, float etc.)
- Any data can be declared using the syntax:

  <data_kind> <data_type> <literal_name>

  example: var logic carry_out;

  wire logic [2:0] sum;

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



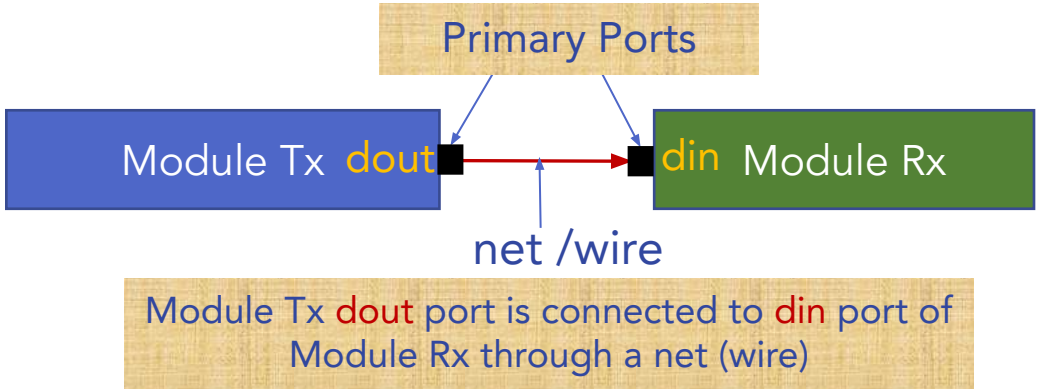
System Verilog Data Type

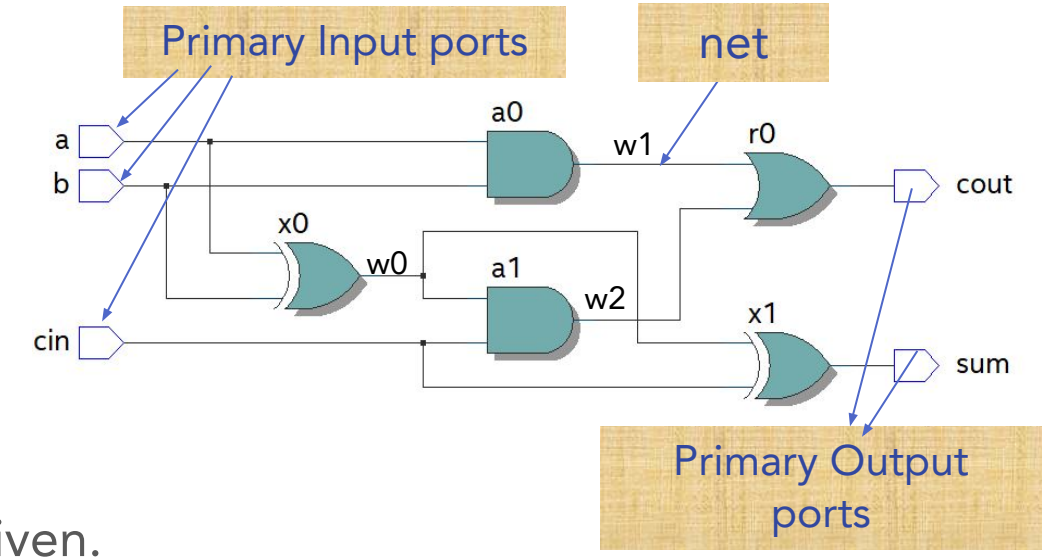
- A net or variable is either 2-state (0,1) or 4-state (0,1,X,Z)
  - Examples :
    - wire is a 4-state net,
    - reg and logic are 4-state variables,
    - bit is a 2-state variable

- Data types are used by simulators are synthesis compilers to determine how to store and process changes on that data
  - Such as, store as 4-state or 2-state value
  - Such as, integer or floating point, e.g., ALU

Nets

- 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





- Net does not store value and 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 (they can provide different value resolution when the net is driven by multiple drivers)

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@(posedge A) 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 to model combinational logic
- wire elements can be used as inputs and outputs within an actual module declaration
- wire and logic elements are the only legal type on the left-hand side of an assign statement
  - Can throw errors when used as the left-hand side of an "=" or "<=" statement within an always block.

Variables

- 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 testbench and design code!)

- Non-synthesizable **variables** in SystemVerilog are :
  - real, shortreal, time, realtime (can be used in testbench code but 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 (but not 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 and initial block
  - Will throw errors when used as the left-hand side of a continuous assign statement.
- reg can be used to create both combinational and sequential logic
  - reg can create registers, for example, when used in conjunction with always@(posedge clock) blocks.

Replace wire / reg 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
  - Let compiler or simulator to infer the 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 and it store values. They 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.
- **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 an 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!

System Verilog 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

(Synthesizability is based on Quartus)

Continuous Assignment Statement

Continuous Assignment

- Continuous assignment statement are used to drive a right-hand side (RHS) expression onto a net or a variable on the left-hand side (LHS)

Syntax :

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

Example :

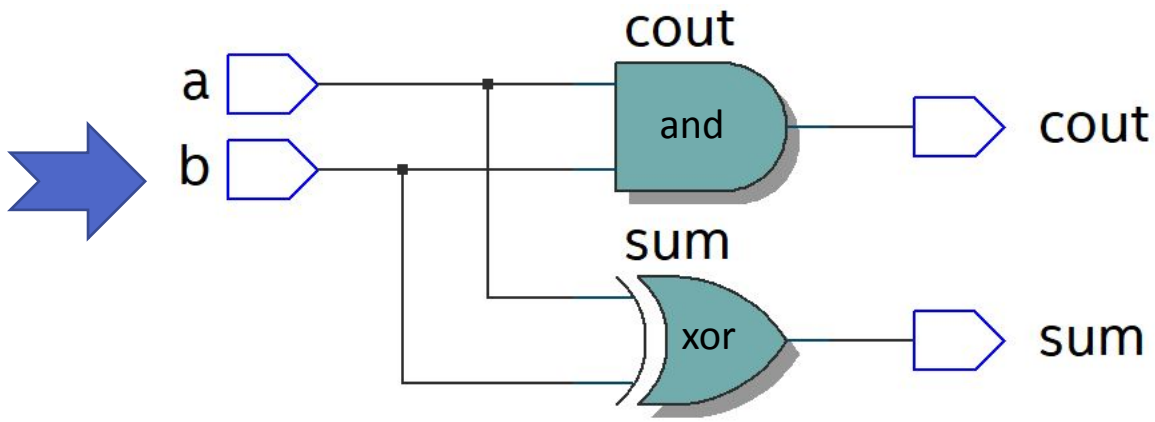
```
wire a, b, c, d;
assign c = a + b;      // assignment to 'd' happens immediately at current simulation time
assign #2 d = a - b;  // assignment to 'd' is delayed by two-time units
```

- 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
- 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
  - Changing the order of multiple continuous statement within a module has no implication on 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
```



Continuous Assignment

- There are two types of continuous assignment statement :
  1. Explicit
  2. Implicit

```
// 1. Explicit: assign is explicitly specified
assign sum = a + b;
```

```
// 2. Implicit: continuous assignment is inferred
wire [2:0] sum = a + b;
```

- 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 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 an error when same variable is driven in both always@ procedural block and driven by continuous assignment statement (though **simulator** may permit it)

```
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 an error when same variable is driven from multiple continuous assignment statements (though **simulator** may permit it)

```
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 MSBs 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 zeros in its MSBs

```
wire[4:0] A, B; // packed array
wire[5:0] C, D; // packed array
wire E [4:0]; // unpacked array

assign A = C;  // MSB of wire C[5] will be truncated

assign D = B; // '0' will get assigned to MSB of wire D[5]

assign E[0] = A[0]; // ERROR: 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 nets, variables (registers), function call, concatenation operations, bit or part selects

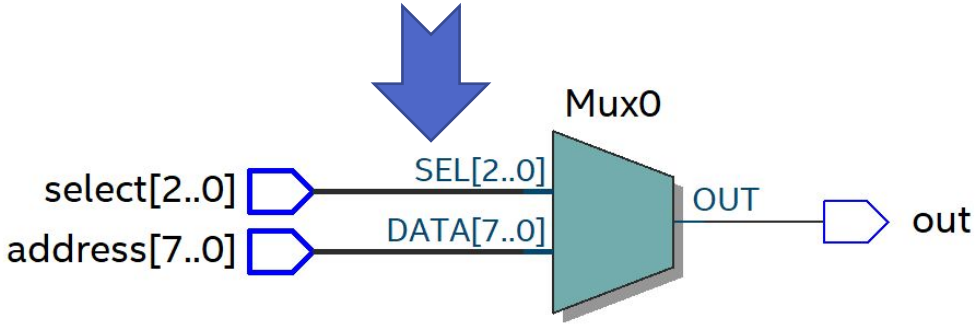
Continuous Assignment: Function Call on RHS Example

```
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: Synthesis

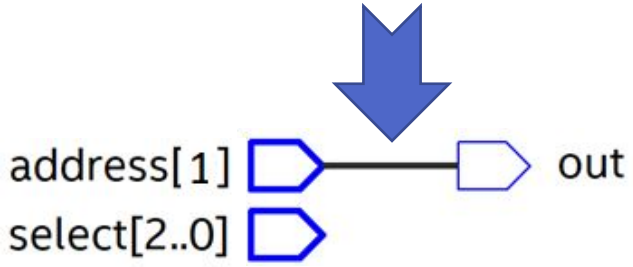
- If RHS expression has an array reference with a variable index then synthesis compiler will generate a mux.

```
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

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 Operator Evaluation

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

- 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: Evaluation Example

- Example :

```
logic sel, mode;
logic [3:0] a, b, mux_out;
assign mux_out = (sel & mode) ? a : b;
```

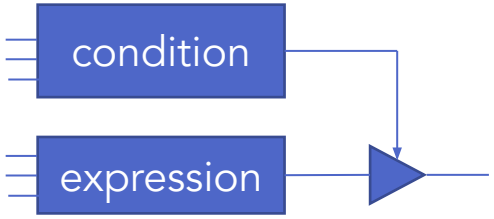
Scenario	Value of "sel"	Value of "mode"	Value of "a"	Value of "b"	Result of conditional expr (sel & mode)	Final value assigned to "mux_out"
1	1'b1	1'b1	4'b0101	4'b1110	True (1)	4'b0101
2	1'b0	1'b1	4'b0101	4'b1110	False (0)	4'b1110
3	1'b1	1'bx	4'b0101	4'b1110	Unknown (x)	4'bx1xx
4	1'b1	1'bx	4'b011x	4'b0z10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

- Conditional operator can also be mapped to tri-state buffer based on how the conditional operator is used and its operand data type and its values.

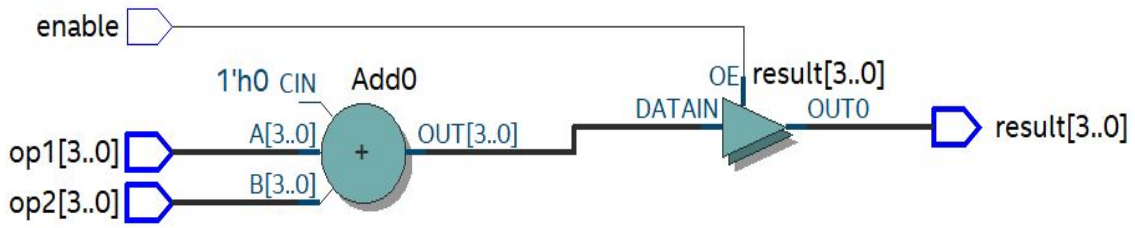
```
assign target = condition ? expression : 1'bz;
```



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
```

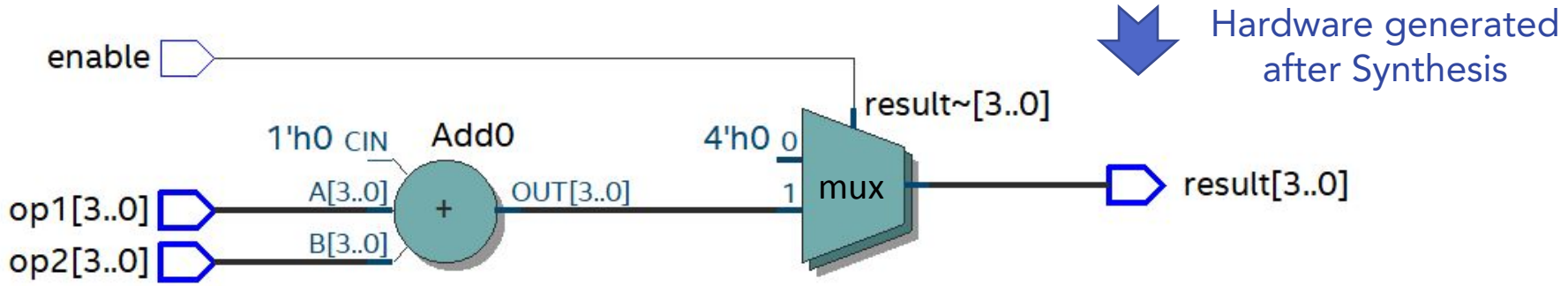
Hardware generated after synthesis



Conditional Operator to 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
```



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

Verilog vs SystemVerilog Data Types

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)
- For synthesizable variables supports only **4-state** (**0,1,X,Z**) variables

System Verilog

- Simplified usage by introducing **logic** data type which can be used for port and signal declaration. Replacing **reg** and **wire** usage.
- 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 !

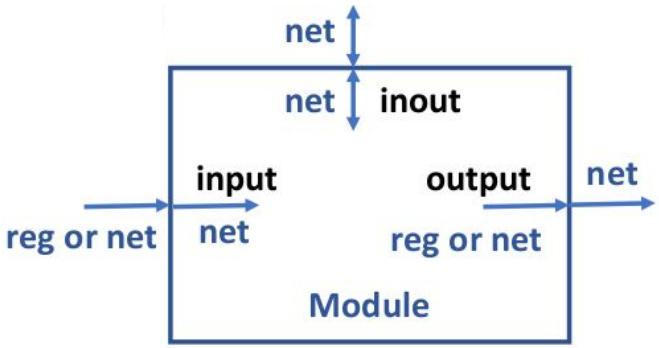
- **Avoid all 2-state data types in RTL modeling**
  - **2-state data types can hide design bugs !**

  - 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

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.

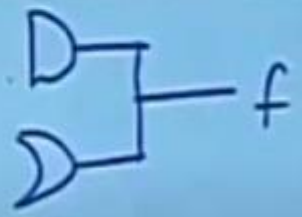
- Real numbers may be specified in either decimal or scientific notation
  - < value >.< value > : e.g 125.6 which is equivalent to decimal 125.6
  - < mantissa >E< exponent > : 2.5e4 which is equivalent to decimal 25000

- Syntax :
  - **real** variable_name;
- Example :
  - **real** height; height = 50.6**;**
    height = 2.4e6**;**

Wand Data Type Example

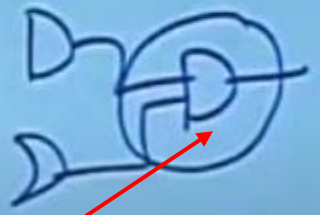
Synthesis tool will give design Error where wire ‘f’ is assigned from two continuous assign statements !

```
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.