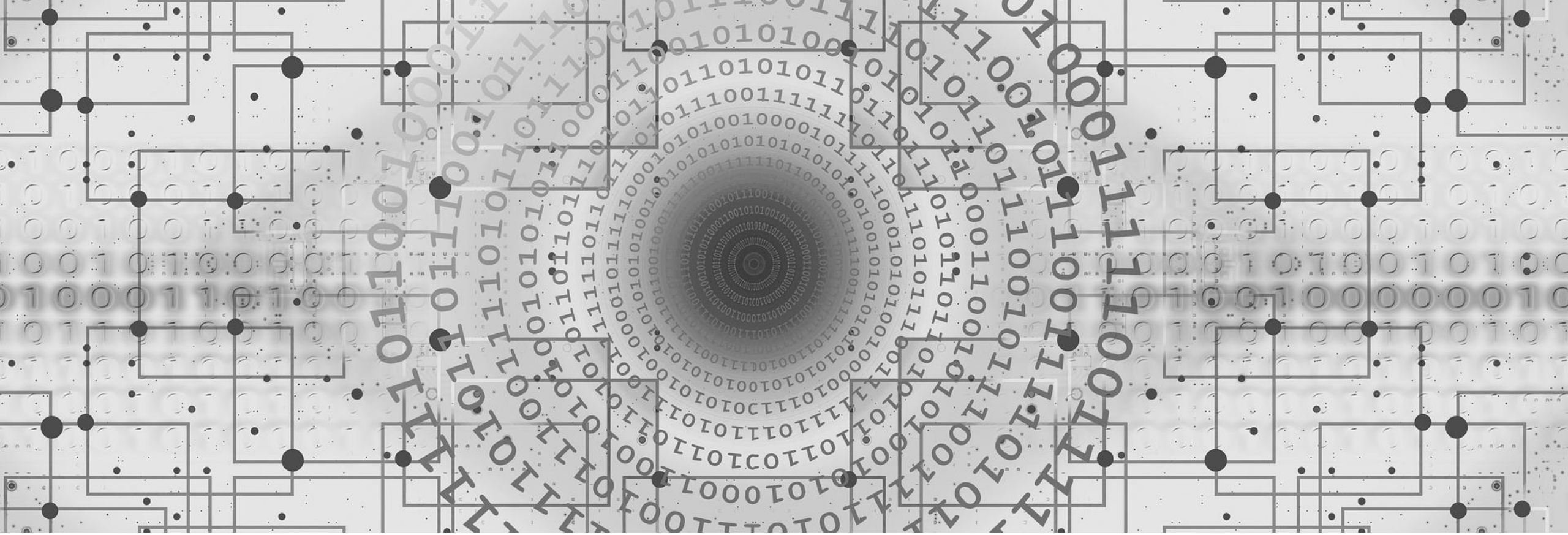


Lecture-4 : Anatomy of SystemVerilog Module

ECE-111 Advanced Digital Design Projects

Vishal Karna

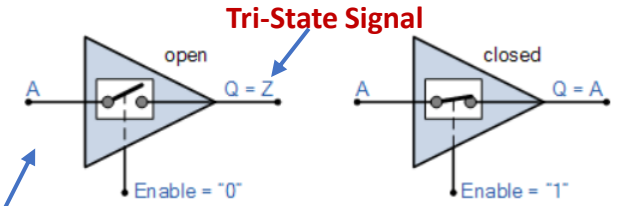
UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

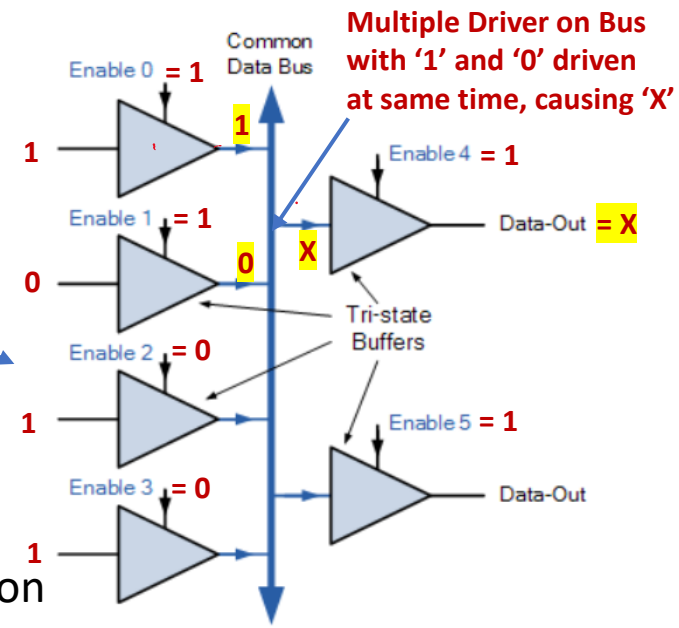
Winter 2022

SystemVerilog Data Value Set

- For hardware 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 (tri-stated signal) ▪ In multi-driver circuit, a value of 0 or 1 will override a Z
X	▪ Represents unknown value or indicates uncertainity (**Simulation Only**) ▪ Indicates a wire or a register is uninitialized, or it indicates unintentionally same wire is driven simultaneously to different logic 0 or 1 by different drivers (also known as multi-driver circuit)





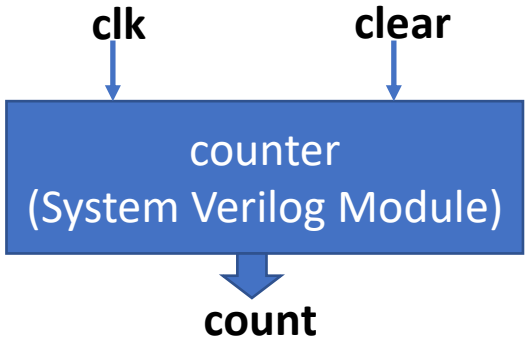
Note :

- 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. It is only has meaning in Simulation.
- Simulators use **'X'** value to indicate uncertainty in how actual hardware would behave under specific conditions if there are registers/wires not initialized or there exists unintentional multiple driver on a wire or a register

SystemVerilog Module Anatomy

- **In SystemVerilog modeling, the primary block is called a Module :**
  - A module is a container that holds the information about a design behavior
  - A module can however contain instances of another module(s), to create hierarchy of modules

- **Module Specification Format :**
  - Module start and end declaration
  - Optional parameter list
  - Primary port declarations with directions
  - Local nets and Variables declarations
  - Concurrent statements which defines functionality and timing of design
  - Instances of other modules




```
module counter    // Module start declaration with module name
 #(parameter WIDTH=4)  // Parameter declaration
  (
    input logic clk,
    input logic clear,
    output logic [WIDTH-1:0] count
  );
logic[WIDTH-1:0] cnt_value; // Local variable declaration

always @(posedge clk or posedge clear)
  begin
    if (clear == 1)
      cnt_value = 0;
    else
      cnt_value = cnt_value + 1;
  end

assign count = cnt_value;
endmodule: counter  // Module end declaration
```

Primary port declarations with directions and data type of each port specified

Functionality of design using concurrent statements

SystemVerilog Module

❑ Module Name

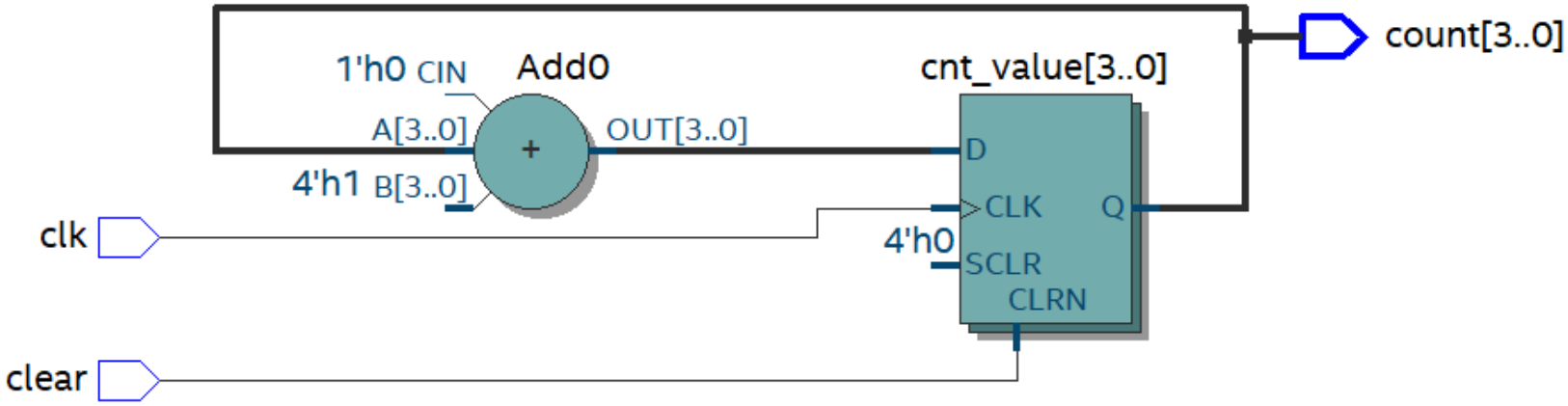
- Enclosed between the keywords **module** and **endmodule**
- Name of the module is user defined (for example *“counter”*)
- After keyword **endmodule** it is optional to specify semi-colon module name (for example **endmodule:** *counter)*
  - Good coding practice for debugging and documentation purpose especially when there might be many lines of code between module and endmodule

❑ Parameter

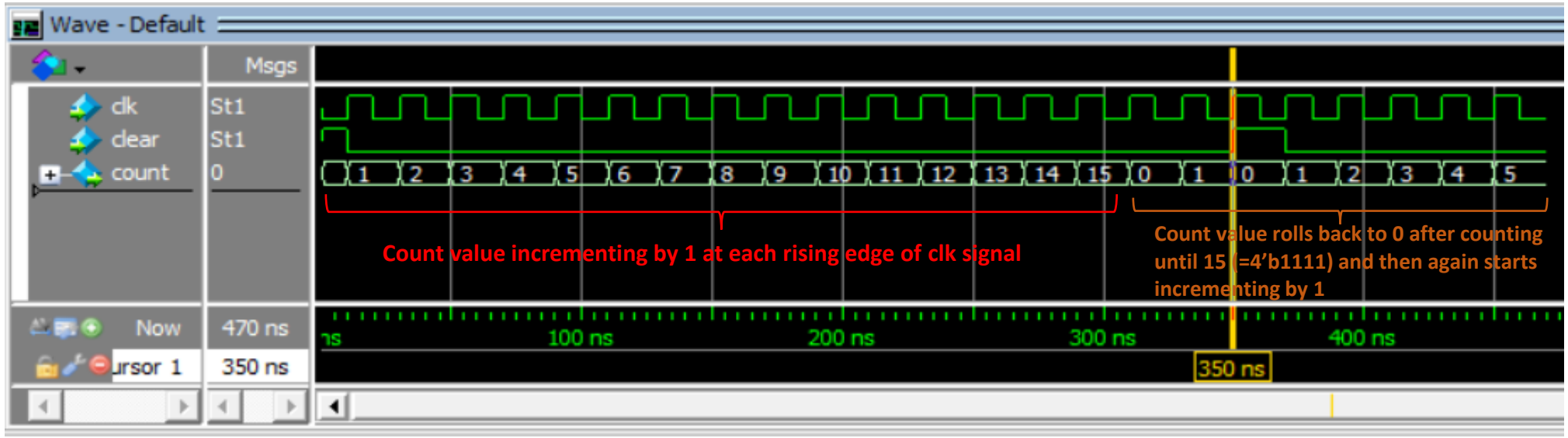
- Optional parameter list specified after module name
- Can have any number of parameters declared with default value
- Enclosed between tokens **#(** and **)**
- Useful to make module configurable
  - Module counter has parameter WIDTH with default value set to 4, which will work as a 4-bit counter
  - Parameter WIDTH can be overridden during instantiation of this module to some other value say, 8 to make it 8-bit counter

4-bit Counter Post Synthesis And Simulation Results

❑ Post-Synthesis :



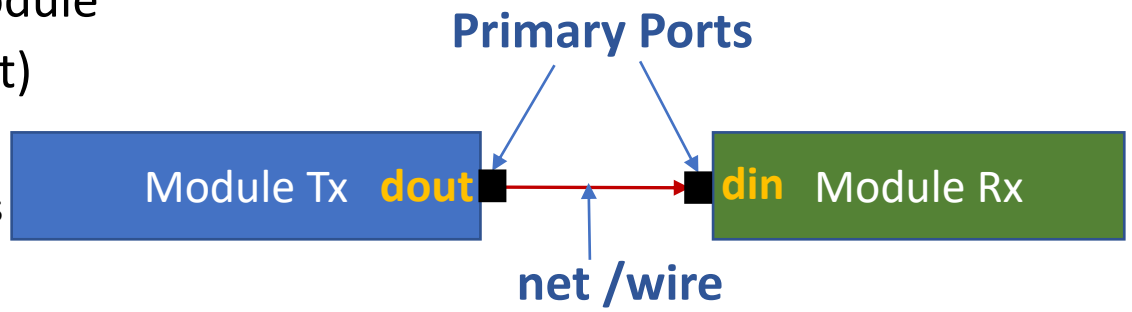
❑ Simulation Waveform :



Port

What is a Port ?

- A port enables passing of data into and out of a module
- Ports of modules are connected using a wire (or net)
- Module can have zero to any number of ports
  - Testbench parent module does not have any primary ports
  - Design modules usually have multiple primary ports



Module Tx is sending data out through dout port into Module Rx through its din port

Each Port has :

- name, direction, width
- data type (built-in and user-defined)

What is a port direction ?

- Specifies direction of a signal for dataflow to and from module
  - **input** : data flows into the module
  - **output** : data flows out of the module
  - **inout :** data flows in and out of the module

inout is also known as **bi-directional** port !

Port Declaration Syntax :

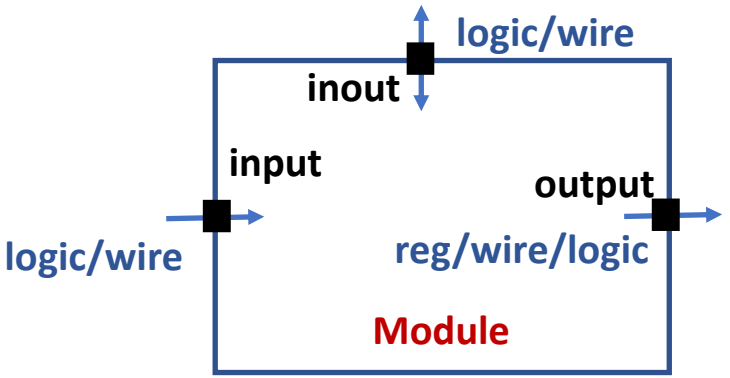
<direction> <datatype> <width> *port_name*

Examples :

```
input wire [3:0] din,   // 4-bit input port
output reg [3:0] dout, // 4-bit output port
inout wire [3:0] a,  // 4-bit inout bi-directional port
input carry_in,  // default datatype is wire if not specified
output wire sum  // 1-bit output port
input wire carry_in  // 1-bit input port
```

Port Data Type

- **Port data type can be either : wire, reg or logic**
  - It is not mandatory to specify port data type.
  - Good practice is to specify port data type
    - **Example : output logic [3:0]** dout;



- **Rules :**
  - **Input** can be of type **wire** and **logic** type
  - **Output** can be of type **reg, wire** and **logic** type
  - **Inout** can be of type **wire** and **logic** type
  - **Input** and **inout** ports cannot be declared as **reg** type in Verilog

- **wire** is a 4-state net
- **reg** is a 4-state variable
- **logic** is a 4-state net or a variable

- In SystemVerilog modules, all module ports and local variables or nets can be declared as a **logic** type. Simulator will correctly infer nets or variables !
  - **Best practice is to declare all ports as logic type when modeling design !**

- **Port data types can be of user-define data types**

```
typedef logic[3:0] uByte;
uByte my_byte;
```

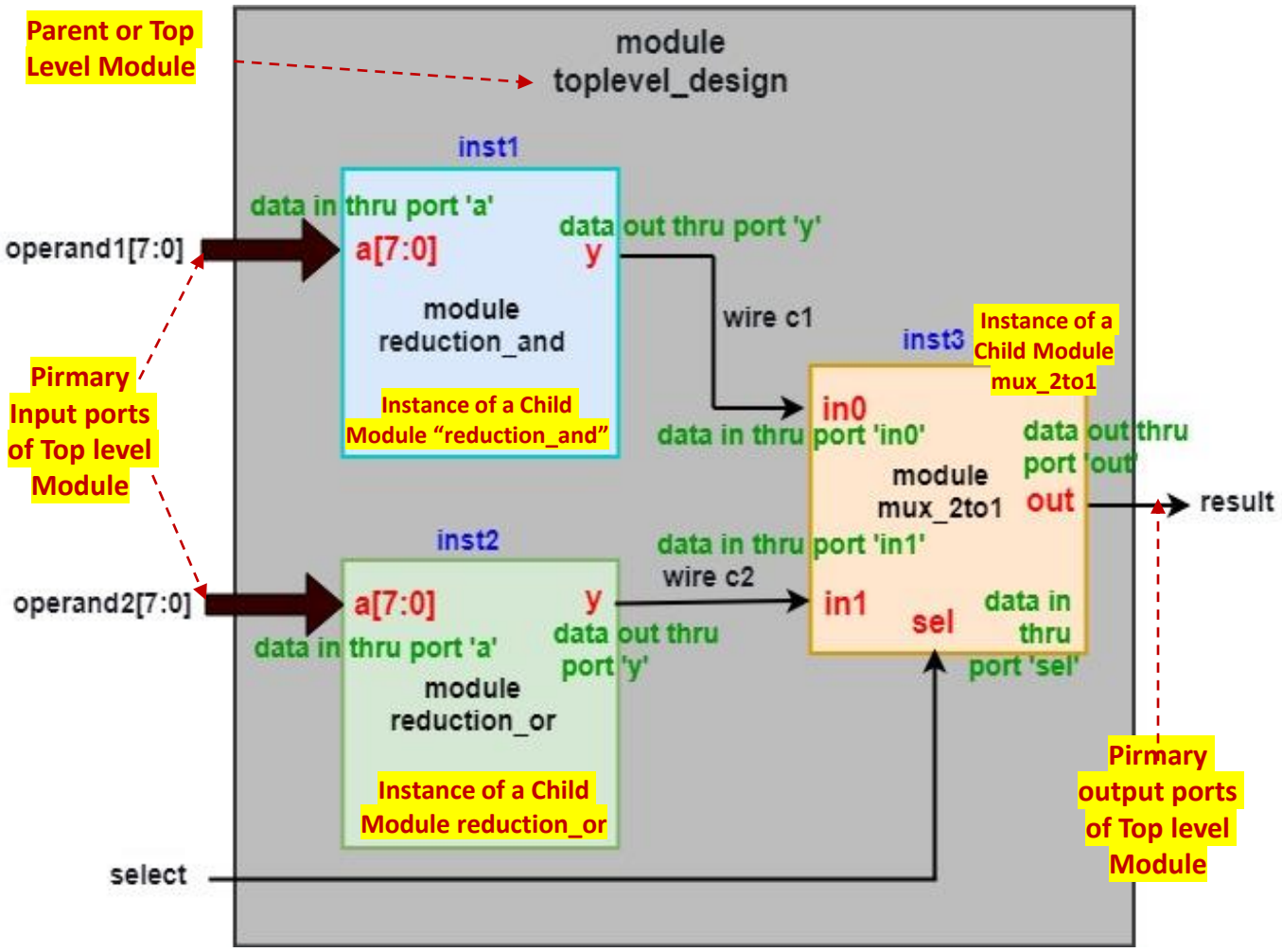
Port Defaults

❑ Port Defaults

- Default port direction is “**input**” if not specified during port declaration
- Default port data type is “**wire**” if not specified during port declaration and within module definition.
- Default port width if not specified is width ‘1’

Hierarchical Module, Module Instantiation and Ports

- A module can contain instances of another module(s), to create hierarchy of modules
- Data flows into and out of modules through ports



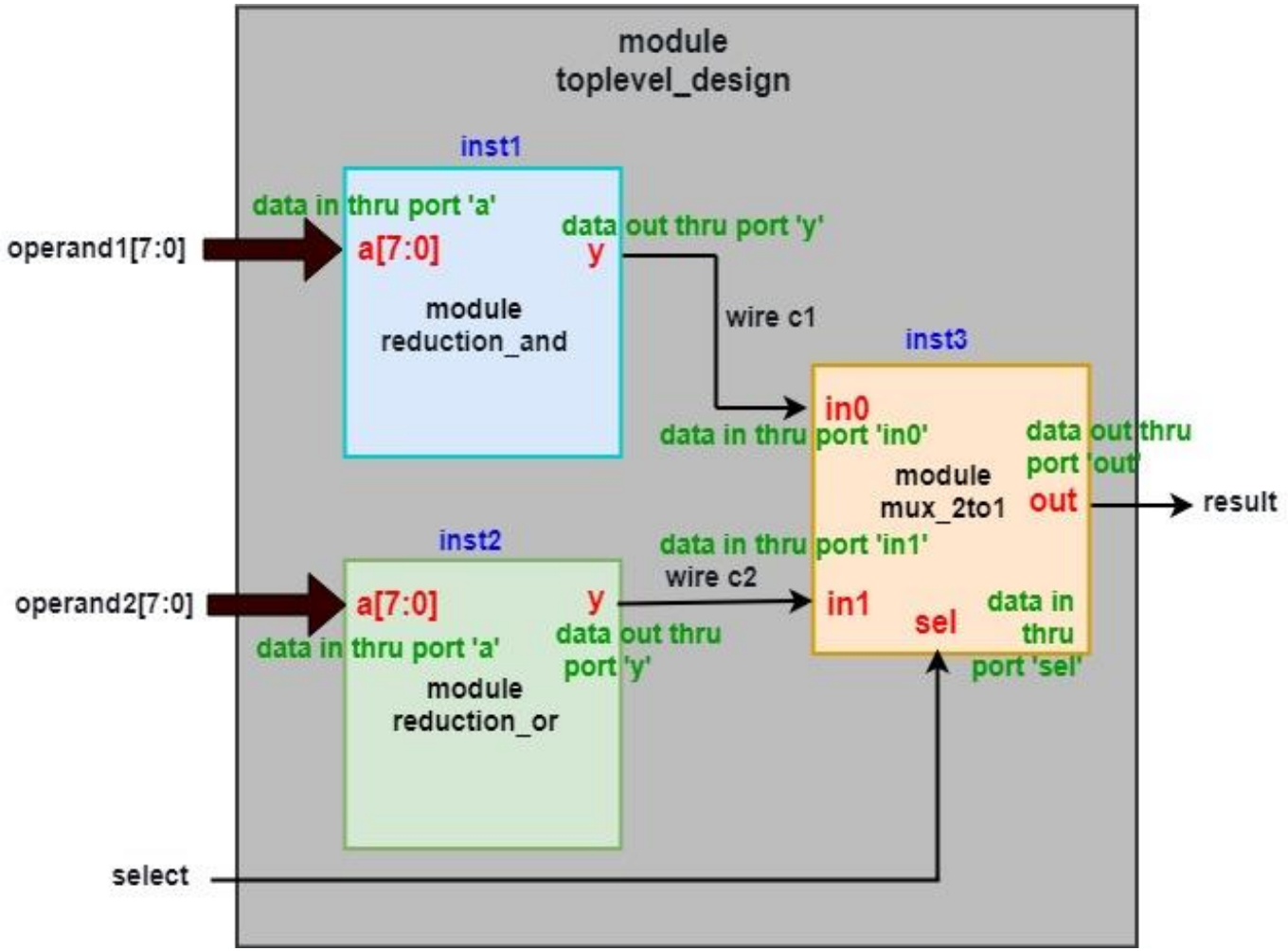

```
module reduction_and (  // Child module definition
  input      wire[7:0]  a,
  output     wire       y
 );
  assign y = a[7] & a[6] & a[5] & a[4] &
             a[3] & a[2] & a[1] & a[0];
endmodule: reduction_and
```

```
module reduction_or  ( // Child module definition
  input      wire[7:0]  a,
  output     wire       y
);
  assign y = a[7] | a[6] | a[5] | a[4] |
             a[3] | a[2] | a[1] | a[0];
endmodule: reduction_or
```

```
module mux_2to1 ( // Child module definition
  input      wire  in0,
  input      wire  in1,
  input      wire  sel,
  output     wire  out
 );
  assign out = (sel==1) ? (out = in0) : (out = in1);
endmodule: mux_2to1
```

Hierarchical Module And Module Instantiation

- Think **Module Instantiation** as a process of adding child modules within a parent module
  - **Example :** Module **reduction_and**, **reduction_or** and **mux_2to1** are **instantiated** in a **toplevel_design** module
  - toplevel_design is a hierarchical module since it contains child modules (**child modules can also be called** as **sub-modules**)




```
module toplevel_design (
 input     wire[7:0]  operand1,
 input     wire[7:0]  operand2,
 input     wire       select,
 output    wire       result
);

 // Declare local nets
 wire c1, c2;

 // Instantiation of module reduction_and
 reduction_and inst1(operand1, c1);

 // Instantiation of module reduction_or
 reduction_or inst2(operand2, c2);

 // Instantiation of module mux_2to1
 mux_2to1 inst3(c1, c2, select, result);

endmodule: toplevel_design
```

Note : Position based port connection method used in module instances inst1, inst2, inst3

Port Connections

- **SystemVerilog provides two ways to define the connections to the module instance ports :**
  - Connect by port order (also known as positional port connections)
  - Connect by Port name (also known as named port connections)
    - explicit named connections **(suggested approach for port connections)**
    - dot-name connections
    - dot-star connections

Explicit Named Port Connections

- Module port name is explicitly associated with connected signal
- In explicit name connections, the name of the port is preceded by a period and the local signal name is enclosed in parenthesis
- **Dis-Advantages :**
  - Verbosity, for each declaration, both port name and connected signal name needs to be specified. This can require considerable amount of duplication
- **Advantages :**
  - Named port connections can help prevent accidental connection errors
  - Port connections can be listed in any order
  - Unused ports can be left out of the connection list
  - Unused ports can be explicitly listed, but with no local signal name in the parenthesis
  - Code is self-documenting. Easy to debug since each port and associated signal connection is visually apparent.

```
module toplevel_design (
 input      wire[7:0]  operand1,
 input      wire[7:0]  operand2,
 input      wire       select,
 output     wire  result
);
 // declare local nets
 wire c1, c2;
 // module reduction_and instantiated using positional based port connection approach
 reduction_and inst1(
   .y(c1),
   .a(operand1)        // Ports 'a' and 'y' can be listed in any order
 );
endmodule: toplevel_design
```

```
module reduction_and (
 input      wire[7:0]  a,
 output     wire       y
);
 assign y = a[7] & a[6] & a[5] & a[4] &
            a[3] & a[2] & a[1] & a[0];
endmodule: reduction_and
```

Recommended usage

Positional Port Connections

- Connects the local net names to the ports of the module instances using the order in which the ports in the module are defined
- Order of the ports in the design module should be known before correct connection
- **Advantages :**
  - Simple to code and less code
- **Disadvantages :**
  - **Error-prone :** Listing a connection in the wrong order can result in design bugs
  - **Difficult to debug :** Not easy to review which port of the module is driven by specific local net
  - **Difficult to manage :** If the module port list changes, all connections in the instantiation of the module needs to be reviewed

```
module toplevel_design (
  input      wire[7:0]  operand1,
  input      wire[7:0]  operand2,
  input      wire       select,
  output     wire  result
 );
  // declare local nets
  wire c1, c2;
  // module reduction_and instantiated using positional based port connection approach
  reduction_and inst1(operand1, c1);// port operand1 is specified at position 1, hence automatically connected to port 'a'
                                    // net c1 is specified at position 2, hence automatically connected to port 'y'
endmodule: toplevel_design
```

```
module reduction_and (
  input      wire[7:0]  a, // a is at position 1
  output     wire       y  // 'y' is at position 2
 );
  assign y = a[7] & a[6] & a[5] & a[4] &
             a[3] & a[2] & a[1] & a[0];
endmodule: reduction_and
```

Dot-Name Port Connections

- Only module port name needs to be specified after dot operator
- SystemVerilog infers that a net or variable of same name is connected to the port
- Explicit named port connection can be mixed with dot-name port connections
- **Advantages :**
  - Concise representation, easier to read and easier to maintain.
- **Rules :**
  - A net or variable with a name that exactly matches the port name must be declared prior to the module instance
  - The net or variable vector size must exactly match with the port vector size
  - Data types one each side of the port must be compatible

```
module toplevel_design (
  input     wire[7:0]  operand1,
  input     wire[7:0]  operand2,
  input     wire       select,
  output    wire       result
 );
  // declare local nets
  wire c1, c2;
  // module reduction_and instantiated using mix of dot-name and explicit name based port connection approach
  reduction_and inst1(
    .y(c1),
    .operand1
  );
endmodule: toplevel_design
```

Both "toplevel_design" and "reduction_and" module have same port name "operand1"

```
module reduction_and (
  input     wire[7:0]  operand1,
  output    wire       y
 );
 assign y = a[7] & a[6] & a[5] & a[4] &
            a[3] & a[2] & a[1] & a[0];
endmodule: reduction_and
```

// dot name port connection and explicit port connection can be mixed
// Ports can be listed in any order.
// Both reduction_and and toplevel_design module is required to have same signal/port name "operand1" with same width

Dot-Star Port Connections

- Dot-star connection is represented with a special token **.***
- Dot-star is a wildcard that indicated that all ports and signals of the same name should automatically be connected for that module instance.
- Explicit named port connection can be mixed with dot-star port connections
- **Dis-Advantages :**
  - Code maintenance and debugging more difficult since port and its associated signal mapping is not mentioned
- **Advantages :**
  - Least amount of code and most concise form of port connection
  - Does not allow a port to be inadvertently left unconnected
  - Does not infer unconnected port
- **Rules :**
  - For dot-star connection to work it is required that signals have same name as port names
  - The net or variable vector size must exactly match with the port vector size

```
module toplevel_design (
  input     wire[7:0]  operand1,
  input     wire[7:0]  operand2,
  input     wire       select,
  output    wire  result
 );
  // declare local net 'y' with same width as port 'y' in reduction_and
  wire y;
  // module reduction_and instantiated using mix of dot-star based port connection approach
  reduction_and inst1(
    .*
  );
endmodule: toplevel_design
```

Both "toplevel_design" and "reduction_and" module have same port name "operand1" and also have same local variable name and port name 'y'

```
module reduction_and (
  input     wire[7:0]  operand1,
  output    wire       y
 );
  assign y = a[7] & a[6] & a[5] & a[4] &
             a[3] & a[2] & a[1] & a[0];
endmodule: reduction_and
```

// operand1 port of toplevel_design module get connected to operand1 port of reduction_and

// local wire 'y' name in toplevel_design gets connected to port 'y' of reduction_and module

N-bit ALU Example With Explicit Name Based Port Connections

```
module alu_top // Module alu_top instantiates "alu"module
#(parameter N=1) // Parameter declaration
( input logic clk, reset,
  input logic[N-1:0]operand1, operand2,
  input logic[1:0] operation,
  output logic[N-1:0] result
);
 logic [N-1:0] alu_out; // local net declaration

 // Instantiation of module alu
 alu #(.N(N)) alu_instance(
  .opnd1(operand1),
  .opnd2(operand2),
  .operation(operation),
  .out(alu_out)
 );
 // Register alu output
 always@(posedge clk or posedge reset) begin
  if(reset == 1)
   result = 0;
  else
   result = alu_out;
 end
endmodule: alu_top // Module alu_top end declaration
```

Port declaration with direction and data type

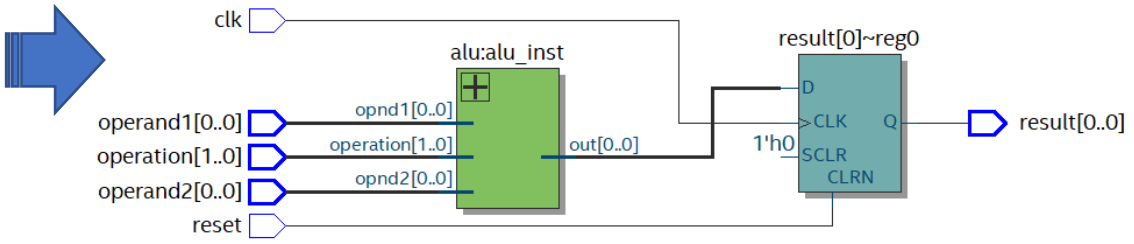
alu module instantiation using explicit name based port connections

Concurrent assignment statements

```
module alu // Module start declaration
#(parameter N=1) // Parameter declaration
(
 input logic[N-1:0] opnd1, opnd2,
 input logic[1:0] operation,
 output logic[N-1:0] out
);
 always@(opnd1 or opnd2 or operation)
 begin
  case(operation)
   2'b00: out = opnd1 + opnd2;
   2'b01: out = opnd1 - opnd2;
   2'b10: out = opnd1 & opnd2;
   2'b11: out = opnd1 | opnd2;
  endcase
 end
endmodule: alu
```

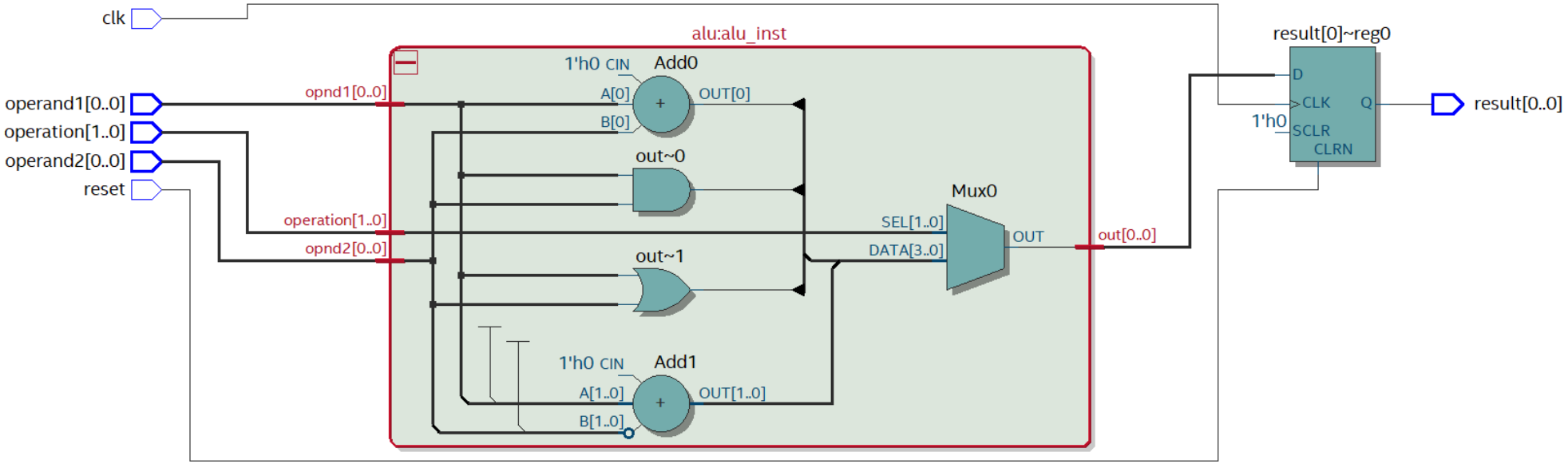
Port declaration with direction and data type

ALU functionality

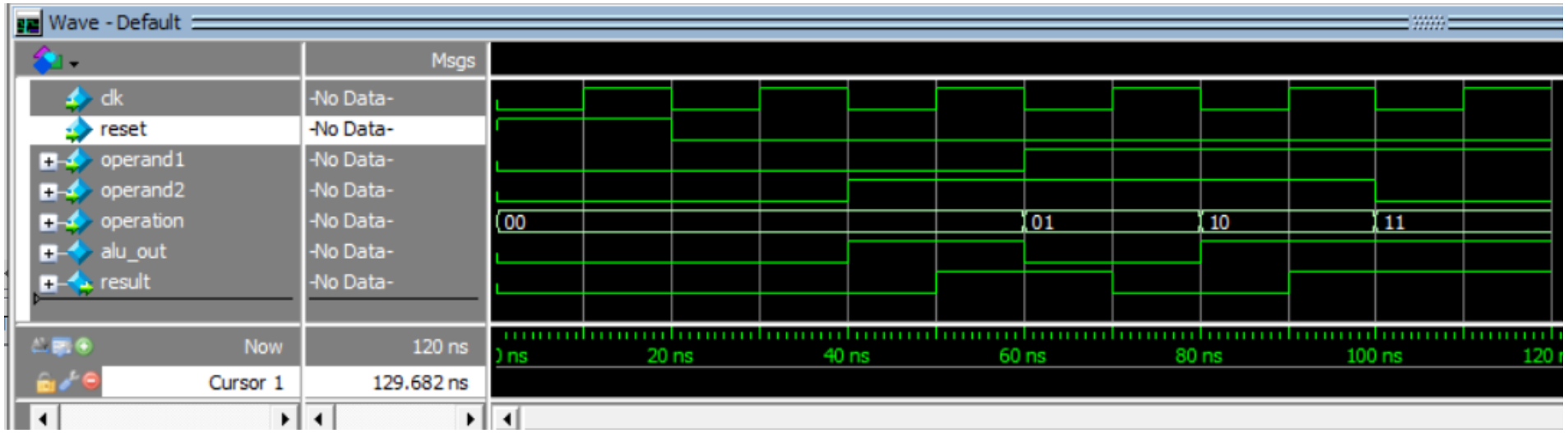


1-bit ALU Post Synthesis And Simulation Results

❑ Post-Synthesis :



❑ Simulation Waveform :



Parameters

Parameters

- Modules can be modeled using parameter construct
- Modules which contain parameter constants are referred as "**parameterized modules**"
- Parameter constants can be used to define port signal width, local variables width and net width
- Parameters are runtime constants (i.e. value can be configured during compilation or elaboration time and once simulation starts running or synthesis start, it has a fixed value)
- There can be multiple parameters defined in module
- See example below where module half adder has a parameter N with default value set to '1', and this half_adder module is instantiated in module top with N overridden to value '4' to make it 4-bit wide adder

```
module half_adder
#(parameter N=1) // Parameter declaration
(
  input logic [N-1:0] a, b,
  output logic [N-1:0] sum
);
  assign sum = a + b;
endmodule
```

```
module top
(
  input logic [3:0] in1, in2,
  output logic [3:0] sum
);
half_adder #(.N(4)) ha_instance(
    .a(in1),
    .b(in2),
    .sum(sum),
 );
endmodule
```

Parameters

- **There are two types of parameter constants**
  - Parameter : run-time constant that can be externally modified
  - Localparam : run-time constant that can be set internally
  - Parameters can be defined with module and in a local scope
  - **Syntax :**
    - **parameter** *data_type signedness size* name = value_expression;
    - **localparam** *data_type signedness size* name = value_expression;

```
module example1
#(parameter N=1) // Parameter declaration
(
  input logic[N-1:0] opnd1, opnd2,
  input logic[1:0] operation,
  output logic[N-1:0] out
);
// Parameter defined in a local scope of a module
parameter SIZE = 32;
……
……
endmodule: example1
```

```
module example2
  parameter N = 8; // N defaults to logic signed [31:0]
  parameter integer Y = 4 // Integer data type parameter
  parameter PI = 3.14; // defaults to real data type
  parameter string NAME = “usb”; // explicit string type
  localparam P = $clog2(N);  // explicit type
  localparam [1:0] STATE_IDLE = 2’b00, // 4 constants of logic type
                   STATE_REQ = 2’b01,
                   STATE_ACK = 2’b10,
                   STATE_GNT = 2’b11;

……
……
endmodule: example2
```

Parameters

- **Parameters can be overridden using defparam during module instantiation time**
  - Using **defparam,** order of parameter when overriding is not required to be maintained

```
module half_adder
// Multiple Parameter declaration
#(parameter N=1,
)
(
  input logic [N-1:0] a, b,
  output logic [N-1:0] sum
);
  parameter W=1;
  assign sum = a + b;
endmodule: half_adder
```

```
module top (
  input logic [3:0] in1, in2,
  output logic [3:0] sum
);
 // parameter W overridden using defparam
 // defparam can be declared before  module instantiation
 defparam ha_instance.W = 4;

 half_adder ha_instance(
    .a(in1),
    .b(in2),
    .c(sum),
  );

 // defparam can also be declared after module instantiation
 defparam ha_instance.N = 4;
endmodule: top
```