Lecture 5: Anatomy of A SystemVerilog Module

UCSD ECE 111

Prof. Farinaz Koushanfar

Fall 2024

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

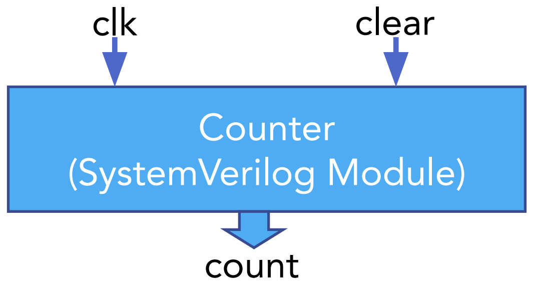
Important announcements (Details in Canvas)

- **Prof office hours:** Tues 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 1:** Homework 1 was posted on Canvas
  - Due on Friday, **10/11/24**
- **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
  - **Late homework policy:** Max of 2 late days, with 20% grade reduction per day

Homework 2 overview

- You will learn how to:
  - Create parameterized modules, how to instantiate a module in another module, how to connect primary ports of 2 modules using explicit name based binding approach
  - Design functional behavior of SystemVerilog arithmetic and logical operators and understand hardware generated for each of the operator post synthesis.
- You will observe change in value of signals in sensitivity list of always block and how it impacts the behavior of the circuit
- There will be two parts for this homework:
  - **Homework-2a**: designing a synthesizable SystemVerilog Model of 4-bit ALU (Arithmetic Logic Unit)
  - **Homework-2b**: developing a synthesizable SystemVerilog code for 4-bit up down binary counter

SystemVerilog Module Anatomy



- A module is a container that holds the information about a design behavior
  - can contain instances of another module(s)
- 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 name declaration
  #(parameter WIDTH=4)      // Parameter declaration
   (
      input logic clk,
      input logic clear,
      output logic [WIDTH-1:0] count
   );
 logic[WIDTH-1:0] cnt; // Local variable declaration

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

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

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

Functionality of design using concurrent statements

System Verilog Module: Name

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

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

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

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

System Verilog Module: Parameter

Parameter

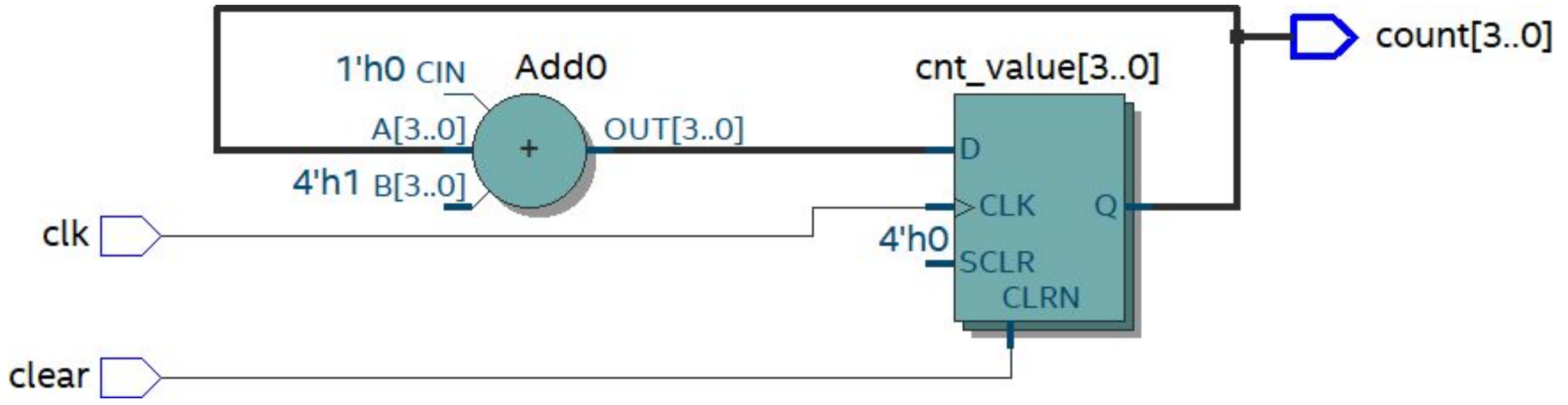
- Optional parameter list specified after module name
- Can have any number of parameters declared with default value
- Enclosed between tokens #(and)
- Useful to make module configurable
  - Module counter has parameter WIDTH with default value set to 4, which will work as a 4-bit counter
  - Parameter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

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

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

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

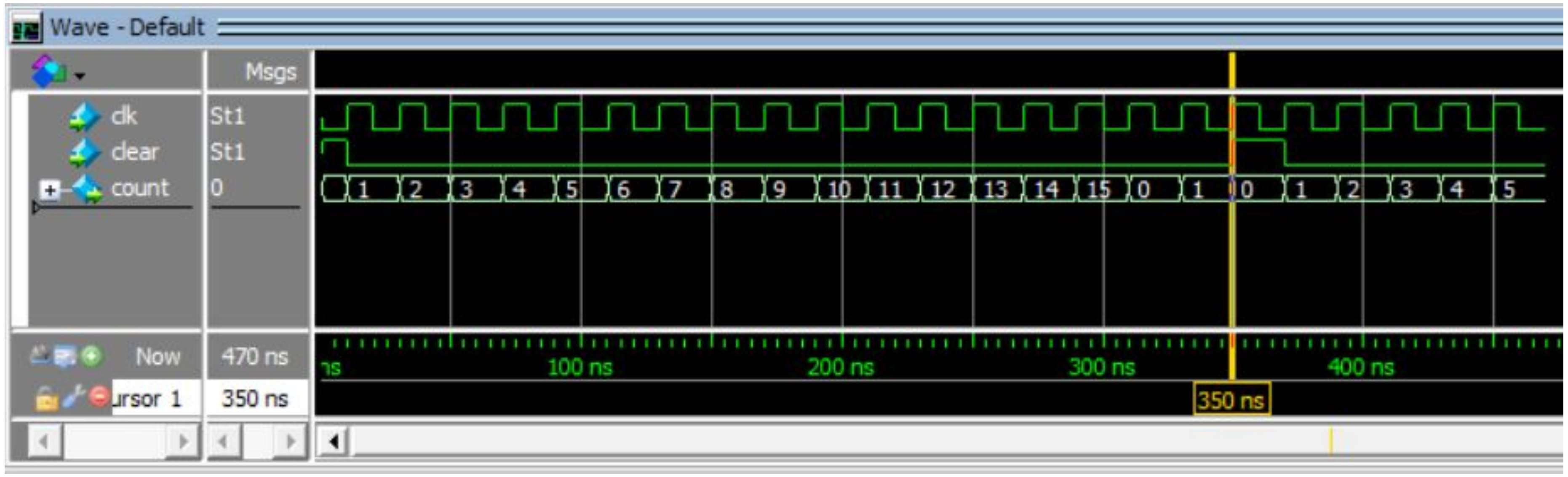
6

4-bit Counter Post Synthesis And Simulation Results

- **Post-Synthesis :**

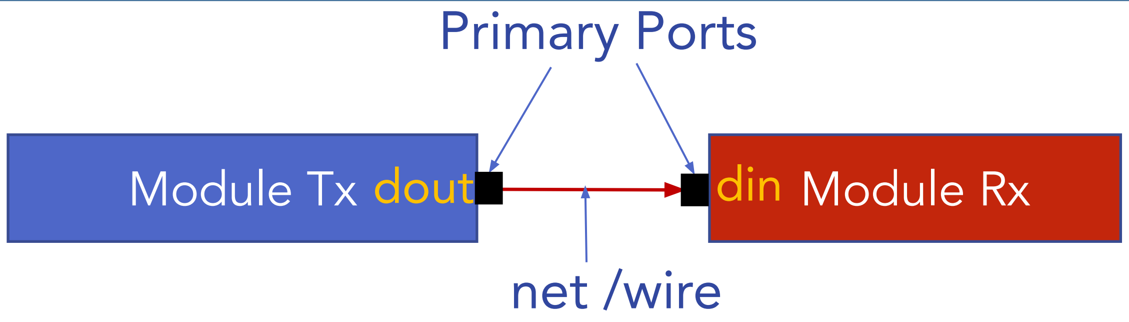


- **Simulation Waveform :**



System Verilog Ports

- Ports are a set of signals that act as inputs and outputs to a particular module
- Ports of modules are connected using a wire (or net)
- Each port has name, direction, width and (built-in or user-defined) data type
- Port direction can be of types:
  - input : data flows into the module
  - output : data flows out of the module
  - inout : data flows in and out of the module (i.e. bi-directional port)



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

System Verilog Ports Data Types

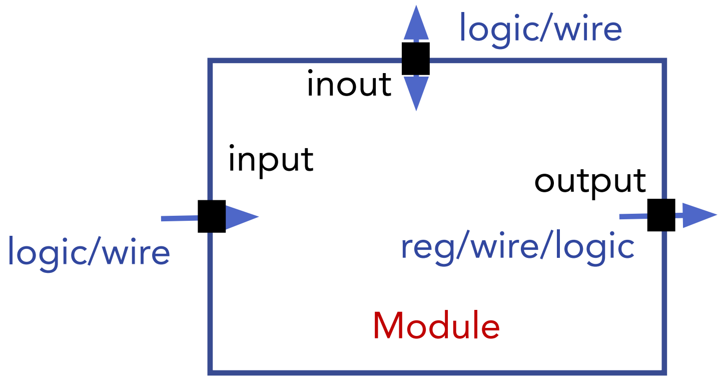
- Port data type can be either : (i) wire, (ii) reg or (iii) logic
  - It is not mandatory to specify port data type
  - Good practice is to specify port data type
    - Example : output logic [3:0] dout;

- 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

- 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;
```



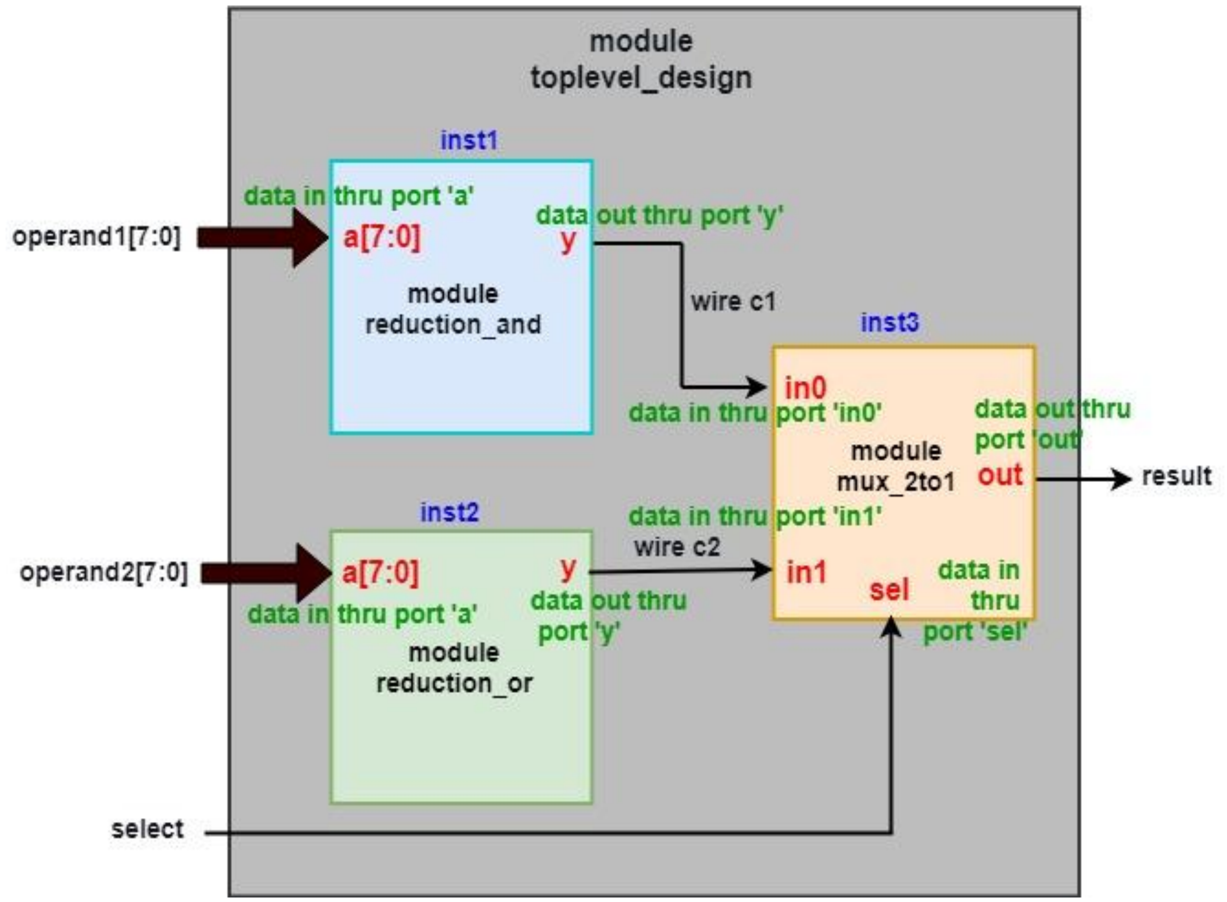
- ❖ wire is a 4-state net
- ❖ reg is a 4-state variable
- ❖ logic is a 4-state net or a variable

System Verilog Port Default

- 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 is width ‘1’ if not specified during port declaration

Hierarchical Modules, Module Instantiation and Ports

- A module can contain instances of another module(s), to create a **hierarchy** of modules



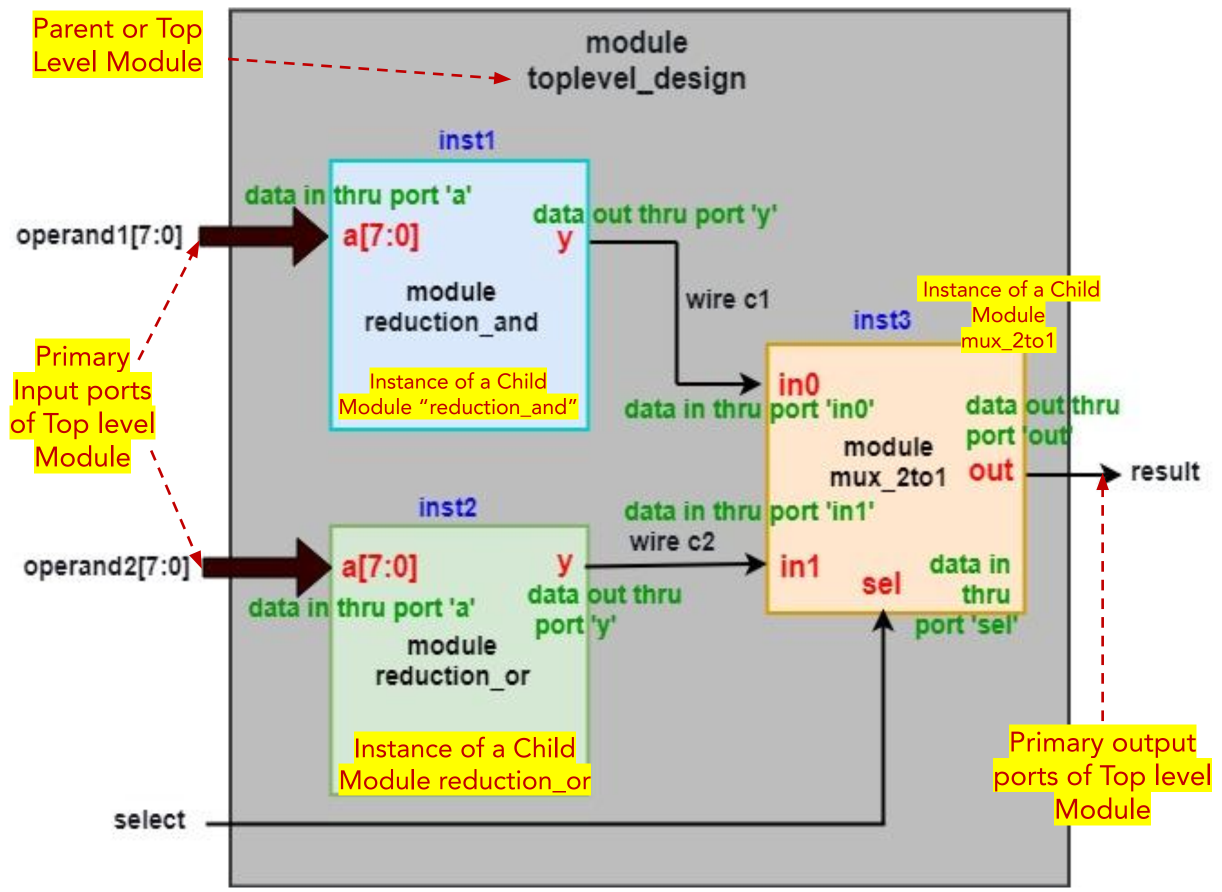
```
module reduction_and (  // Child module definition
  input wire[7:0]      a,
  output        wire    y
 );
  assign y = a[7] & a[6] & a[5] & a[4] &
            a[3] & a[2] & a[1] & a[0];
endmodule: reduction_and
```

```
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 Instantiation

- Module Instantiation is a process of adding child modules within a parent module



```
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
```

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

```
// port order
wire a, b, c;
circuit inst1 (
   a,
   b,
   c
);
```

```
// explicit
wire a, b, c;
circuit inst1 (
   .b (b),
   .a (a),
   .c (c)
);
```

```
// dot-name
wire a, b, c;
circuit inst1 (
   .a,
   .b,
   .c
);
```

```
// dot-star
wire a, b, c;
circuit inst1 (
   .*
);
```

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

- Advantages :
  - Simple to code and less code

Disadvantages :

- Error-prone : Listing a connection in the wrong order
- 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'
endmodule: toplevel_design   // net c1 is specified at position 2, hence automatically connected to port 'y'
```

```
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
```

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 parentheses

Disadvantage :

- Verbosity: for each declaration, both port name and connected signal name is specified

Advantages :

- Named port connections can help prevent accidental connection errors
- Port connections can be listed in any order
- Unused ports can be left out or explicitly listed, with no local signal name in the parentheses
- Code is self-documenting and easy to debug

```
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

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(       // dot-name port connection and explicit port connection can be mixed
   .y(c1),                  // Ports can be listed in any order.
   .operand1                // Both reduction_and and toplevel_design module is required to have same signal/port name "operand1" with same
 );                         width
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
```

N-bit ALU Example With Explicit Name Based Port

```
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

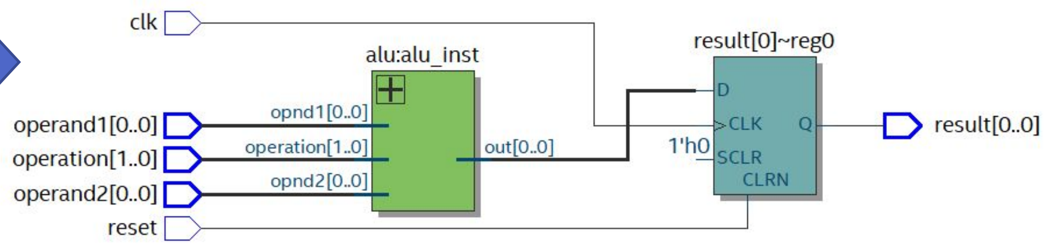
alu module instantiation using explicit name based port connections

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
```

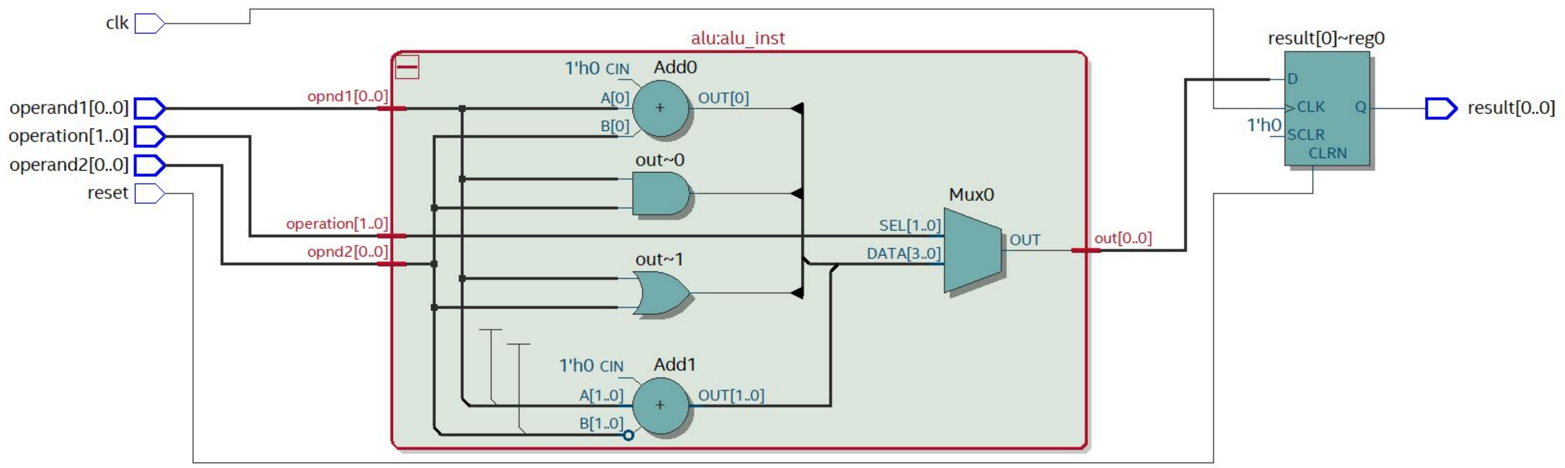
Port declaration with direction and data type

ALU functionality

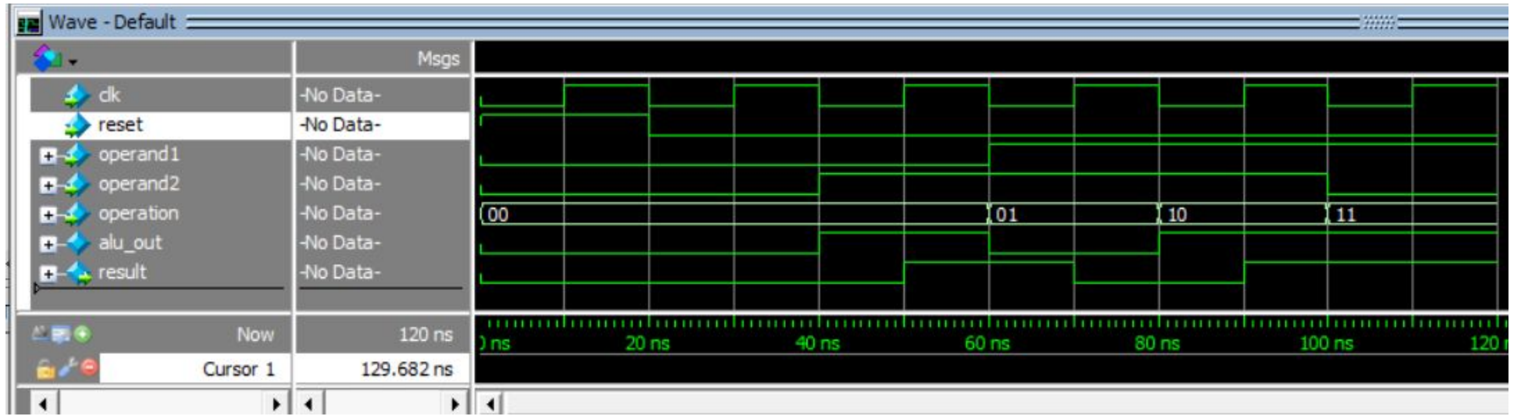


1-bit ALU Post Synthesis And Simulation Results

Post-Synthesis :



Simulation Waveform :



Parameterized Modules

- Modules can be modeled using **parameter** construct and are referred to 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 a module

```
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

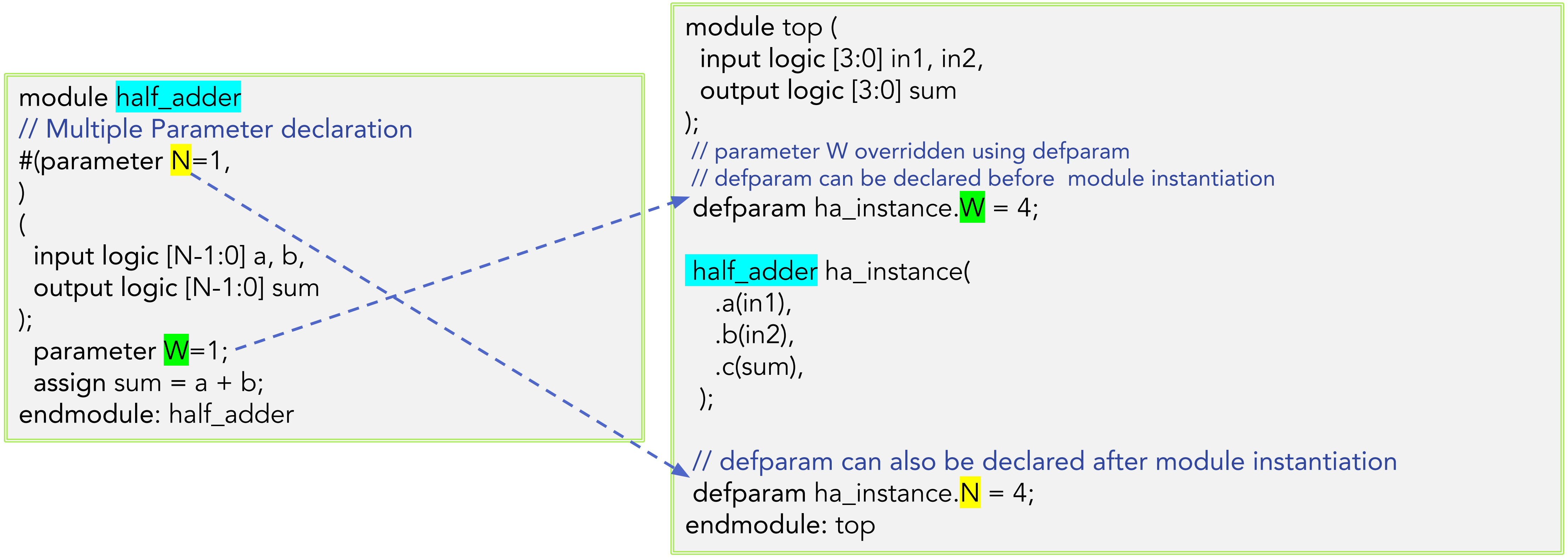
- There are **two types** of parameter constants
  1. Parameter : run-time constant that can be **externally modified**
  2. Localparam : run-time constant that can be **set internally**
- Parameters can be defined within a module and in a **local scope**
- Syntax :
  - parameter *data_type* *signedness* *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
```