University of San Carlos | Department of
COMPUTER ENGINEERING

CPE 3101L – Introduction to HDL

Unit 4: Dataflow Modeling of Combinational Circuits

Lecture 1

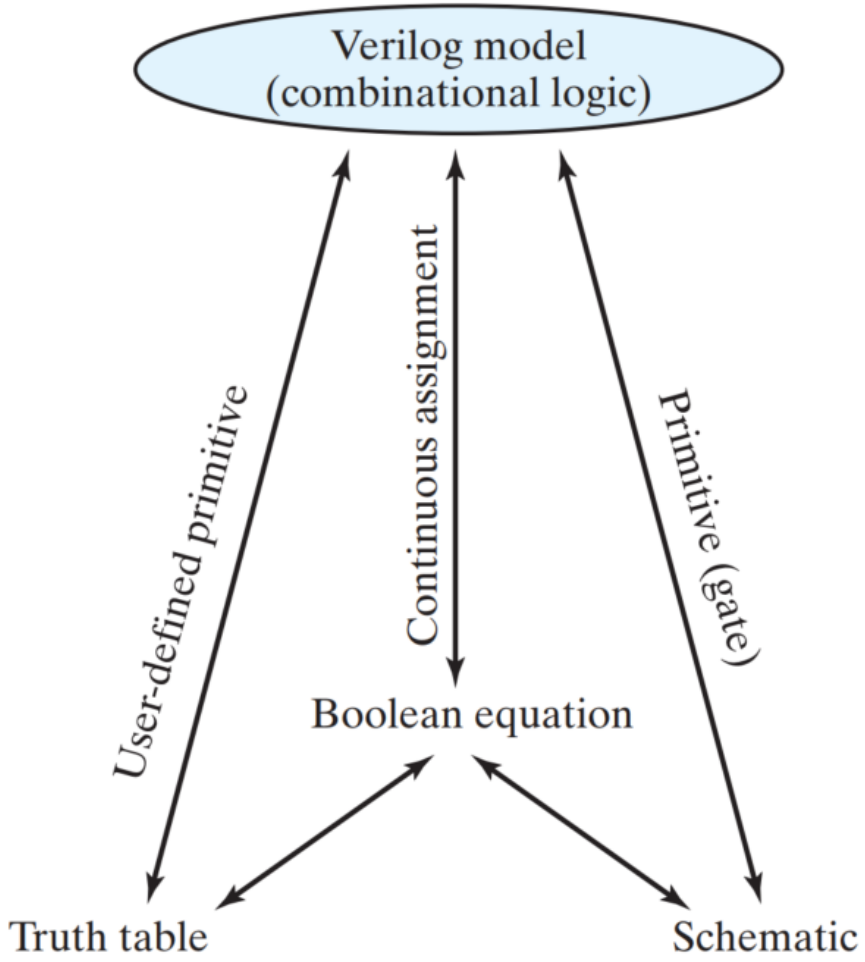
Outline

- Dataflow Modeling of Combinational Circuits
- Verilog HDL Operators
- Continuous Assignment: *Assign* Statement
- Conditional Assignment
- Synthesis of z and x values
- Concatenation and Replication
- Constants and Parameters

Combinational Logic in Verilog

- To create the HDL model of combinational circuits:

- Structural modeling *(Units 2-3)*
  - Gate primitives, ~~UDPs~~, other HDL modules
  - Needs schematic ~~or truth tables~~ or other HDL modules
- Dataflow modeling *(Unit 4)*
  - Needs Boolean equations
- Behavioral modeling *(Unit 5)*
  - Needs behavior or functionality of the circuit
  - *More frequently used in sequential logic (Unit 6)*



Dataflow Modeling of Combinational Circuits

- **Dataflow modeling** uses a number of operators that act on binary operands to produce a binary result.

- **Dataflow HDL models** describe combinational circuits by their ***function*** rather than their gate structure.
  - Boolean function
  - Logical function

Verilog HDL Operators

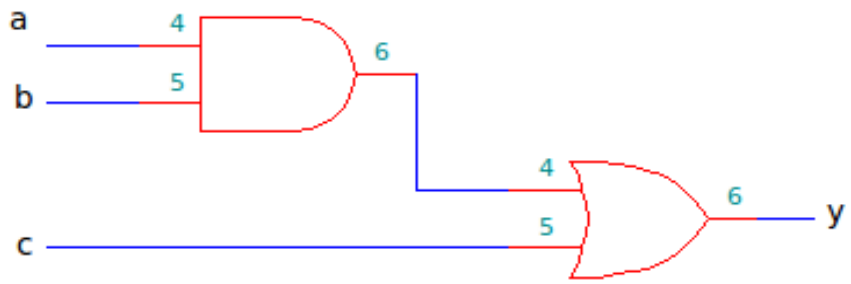
	Symbol	Operation	Symbol	Operation	
Bitwise	&	Bitwise AND	&&	Logical AND	**Logical**
	|	Bitwise OR	||	Logical OR	
	^	Bitwise XOR			
	~	Bitwise NOT	!	Logical NOT	
Equality	==	Equality	!=	Inequality	
	>	Greater than	<	Less than	
	>=	Greater than or equal	<=	Less than or equal	
Arithmetic	+	Binary addition	-	Binary subtraction	
	%	Modulus	**	Exponentiation	
Shift	>>	Logical right shift	>>>	Arithmetic right shift	
	<<	Logical left shift	<<<	Arithmetic left shift	
Concatenation	{ }	Concatenation	{ { } }	Replication	
Conditional	?:	Conditional			

**Use the bitwise operations to describe arithmetic operations and logical operators for logical operations.*

Continuous Assignment

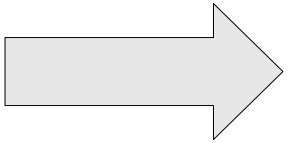
- *Assign* keyword is a straightforward way of assigning a combinational logic function to a net (*wire*)
- Use of operators is allowed in an *assign* statement

- Example:



Structural model** with **gate primitives

```
wire   x, y;

and    u1 (x, a, b);
or     u2 (y, x, c);
```



Dataflow model** using **assign

```
wire   y;

assign y = (a & b) | c;
```

Example 1: Full Adder

- Create a Verilog dataflow description of a full adder.

- Specifications:
  - Inputs: x, y, z
  - Outputs: C, S

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = z \oplus (x \oplus y)$$

$$C = xy + xz + yz$$

Example 1: Design Entry (Full Adder)

- Entity Diagram:

- Logic Function/s:

$$S = z \oplus (x \oplus y)$$

$$C = xy + xz + yz$$

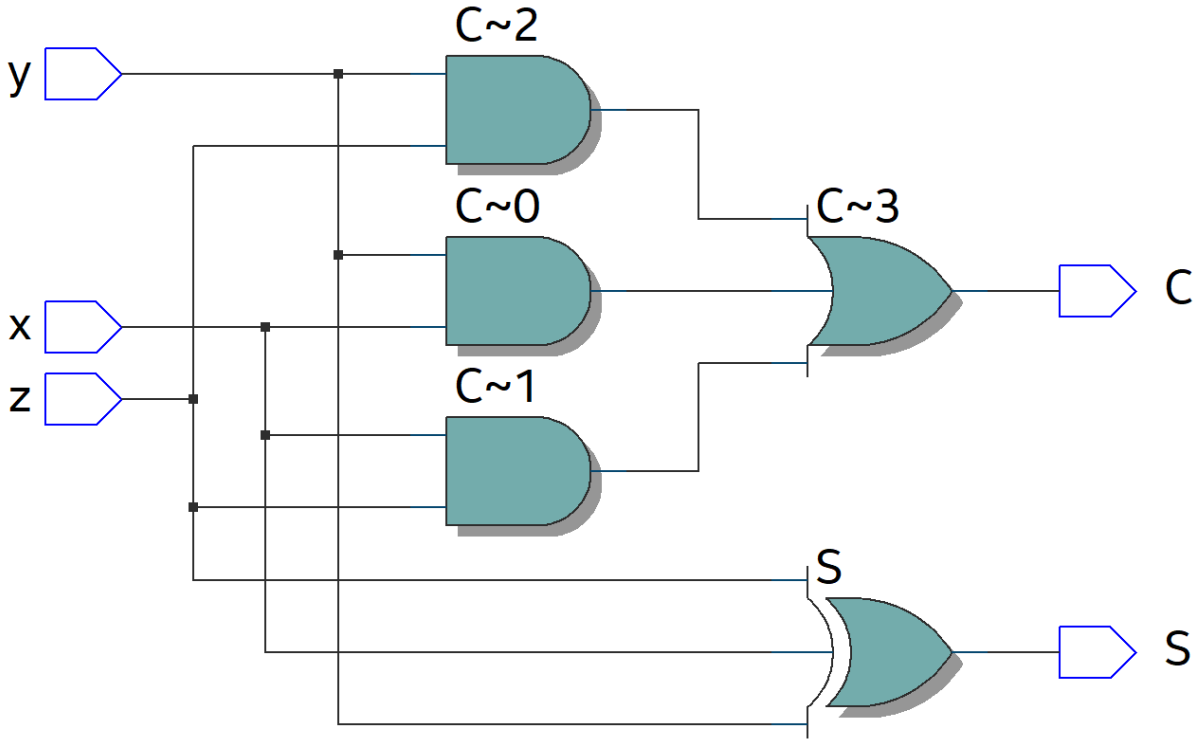
- Verilog HDL Description:

```
// Dataflow model of a Full Adder
module FullAdder (
   input    x, y, z,
   output   C, S
);

   assign S =  x ^ y ^ z;  // S = z ^ (x ^ y);
   assign C =  (x & y) | (x & z) | (y & z);

endmodule
```

Example 1: Synthesized Design (Full Adder)

- RTL Schematic View:

Total logic elements	2
Total registers	0
Total pins	5



Example 1: Testbench File (Full Adder)

```
// Testbench file of Dataflow Full Adder

`timescale 1 ns / 1 ps
module tb_FullAdder ();

   // declare inputs as reg types
   reg   x, y, z;

   // outputs as wire types
   wire  C, S;

   // instantiate UUT
   FullAdder UUT ( .x(x), .y(y), .z(z), .C(C), .S(S) );

   // generate stimuli
   initial begin
      $display("Starting simulation at %0d ns...", $time);

      x = 1'b0;   y = 1'b0;   z = 1'b0;   #10
      x = 1'b0;   y = 1'b0;   z = 1'b1;   #10
      x = 1'b0;   y = 1'b1;   z = 1'b0;   #10
      x = 1'b0;   y = 1'b1;   z = 1'b1;   #10

      x = 1'b1;   y = 1'b0;   z = 1'b0;   #10
      x = 1'b1;   y = 1'b0;   z = 1'b1;   #10
      x = 1'b1;   y = 1'b1;   z = 1'b0;   #10
      x = 1'b1;   y = 1'b1;   z = 1'b1;   #10

      $display("Finished simulation at %0d ns.", $time);
      $stop;
   end

   // response monitor
   initial begin
      $monitor("Time = %2d ns\t xyz = %b%b%b\t C = %b\t S = %b", $time, x, y, z, C, S);
   end

endmodule
```

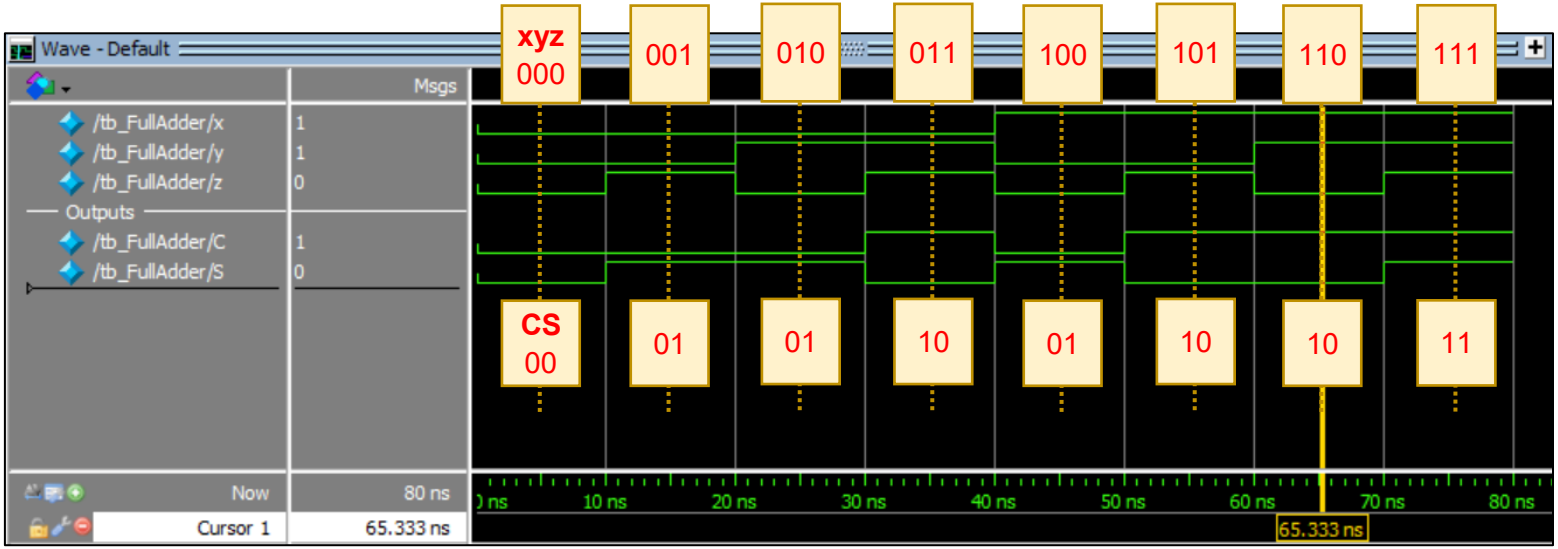
Example 1: Simulation (Full Adder)

- Standard Output:

```
# Starting simulation at 0 ns...
# Time =  0 ns  xyz = 000  C = 0  S = 0
# Time = 10 ns  xyz = 001  C = 0  S = 1
# Time = 20 ns  xyz = 010  C = 0  S = 1
# Time = 30 ns  xyz = 011  C = 1  S = 0
# Time = 40 ns  xyz = 100  C = 0  S = 1
# Time = 50 ns  xyz = 101  C = 1  S = 0
# Time = 60 ns  xyz = 110  C = 1  S = 0
# Time = 70 ns  xyz = 111  C = 1  S = 1
# Finished simulation at 80 ns.
```

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- Waveform:



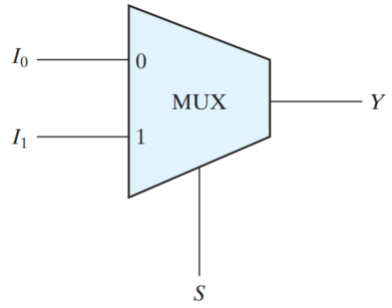
Conditional Assignment

- Assigned value is dependent on the evaluation of a condition
- Uses the ***conditional operator (?:)*** which takes 3 operands:

```
[condition] ? [true_expression] : [false_expression];
```

- Condition *(Boolean expression)* is evaluated:
  - If result is logic 1, *true_expression* is evaluated;
  - Else, *false_expression* is evaluated
- Can be used with ***assign*** statement, but not limited to it

- Example: Modeling a 2x1 mux *(3* assign *statement variations)*

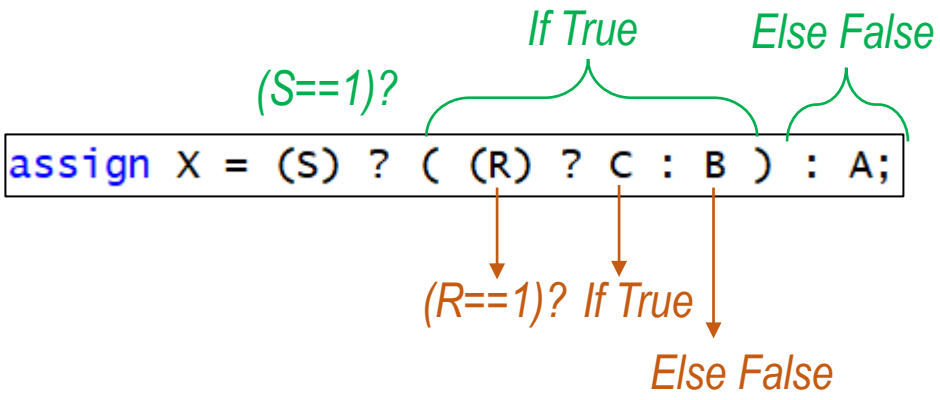


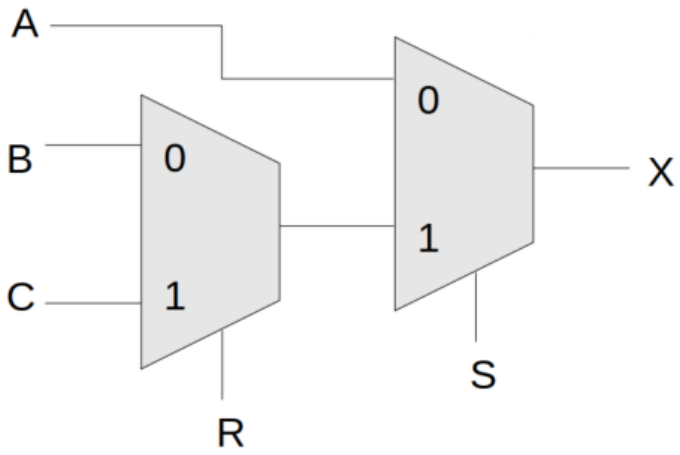

```
assign Y = (S==0) ? I[0] : I[1];
assign Y = (S==1) ? I[1] : I[0];
assign Y = (S)    ? I[1] : I[0];
```

Conditional Assignment

- Conditional assignments may also be **cascaded** or **nested**
- Less lines of code but can be harder to read
- **Synthesized hardware** of conditional assignments infers 2x1 multiplexers

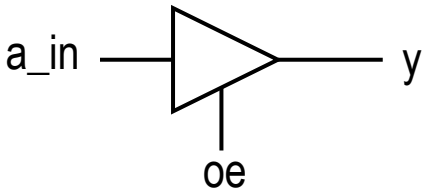
- Example:





Synthesis of z

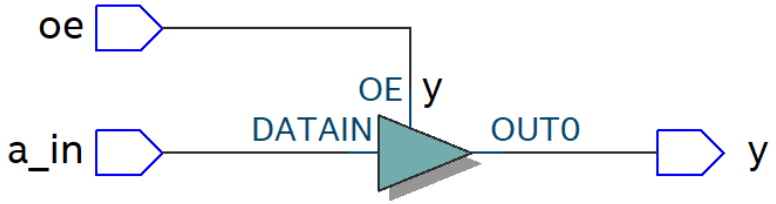
- The **z** value implies high impedance or an open circuit.
- Not a normal logic value (like 1 or 0)
- Can only be synthesized by a *tri-state buffer*



Output Enable (oe)	y
0	z
1	a_in

- Dataflow model of a tri-state buffer:

```
assign y = (oe) ? a_in : 1'bz;
```



Synthesis of x

- **Don't care values (x)** are assigned when input patterns/ combinations never occur, thus the **output value is irrelevant**.

- Example:

Input i	Output y
00	0
01	1
10	1
11	x

```
assign y =  (i == 2'b00) ? 1'b0 :
            (i == 2'b01) ? 1'b1 :
            (i == 2'b10) ? 1'b1 :
                           1'bx;    // i == 2'b11
```

- This approach helps minimize the circuit but it introduces a discrepancy between simulation and synthesis:
  - Simulation: x is a unique value (rather than 0 or 1)
  - Synthesis: x may either be 0 or 1 (not x)

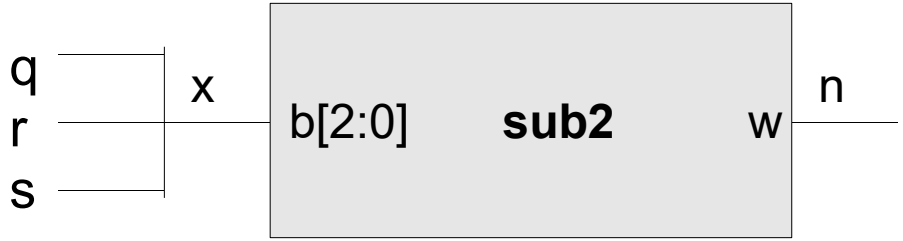
This input pattern (11) never occurs so don't care value (x) is assigned to output y for optimization (i.e. K-maps, etc.)

Concatenation

- Multiple signals may be concatenated to form a larger signal *(larger array)*
- Uses the **concatenation operator ({ })**

```
assign [large_signal] = { [sig1], [sig2], … };
```

- Implementation involves reconnection *(rewiring)* of input/output signals
- Useful for transforming individual signals into **buses**
- Example:




```
wire [2:0]  x;

assign x = {q, r, s};

sub2 U1 ( .b(x), .w(n) );
```

Concatenation

- Syntax: **assign** [*large_signal*] = { [*sig1*], [*sig2*], … };
- Examples:

```
wire a1;
wire [3:0] a4;
wire [7:0] b8, c8, d8;

assign b8 = {a4, a4};
assign c8 = {a1, a1, a4, 2'b00};
assign d8 = {b8[3:0], c8[3:0]};
```

```
wire [7:0] a, rot, shl, shr;

assign rot = {a[6:0], a[7]};  //rotate a once to the left
assign shl = {a[6:0], 1'b0};  //logic shift left once (multiply a by 2)
assign shr = {1'b0, a[7:1]};  //logic shift right once (divide a by 2)
```

Replication

- Used to replicate a signal a finite number of times
- The concatenation operator *({ N{} })* replicates the enclosed signal
- The replication constant *(N)* specifies the number of replications:

```
assign [large_signal] = { N {[signal]} };
```

- Useful for padding bits or for sign-extension

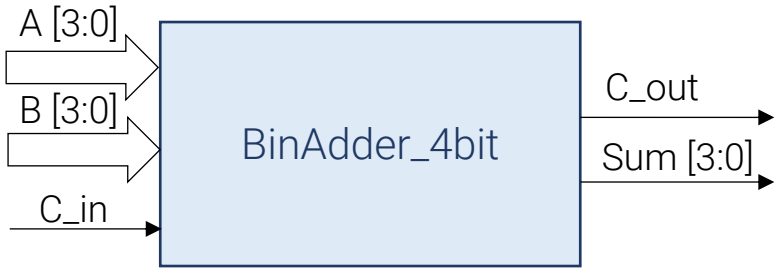
- Example: 2's complement sign-extension from 8-bit to 32-bit

```
wire [7:0]  x;
wire [31:0] y;

assign y = { { 24 {x[7]} }, x }; // sign-extend MSB
```

Example 2: 4-bit Binary Adder

- Create a Verilog dataflow description of a 4-bit binary adder.

- Specifications:
  - Inputs: A [3:0], B [3:0], C_in
  - Outputs: C_out, Sum [3:0]



- Design Entry:

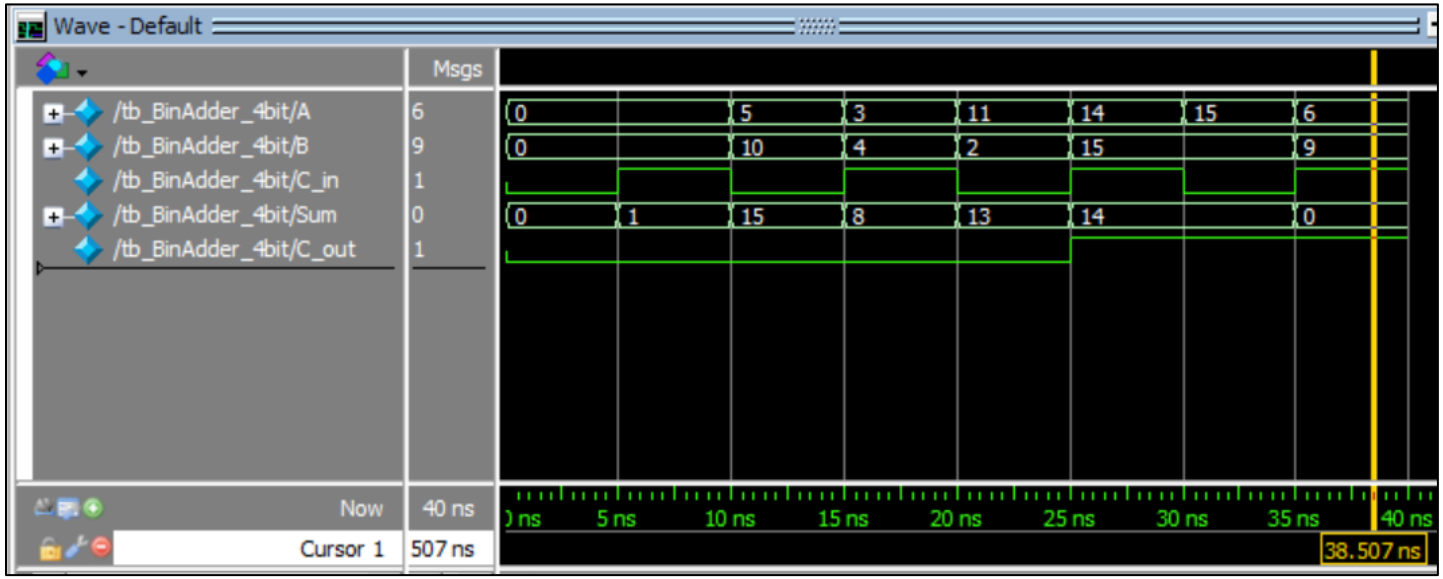
```
// Dataflow model of a 4-bit Binary Adder
module BinAdder_4bit (
    output   [3:0] Sum,
    output         C_out,
    input    [3:0] A, B,
    input          C_in
);

    assign {C_out, Sum} = A + B + C_in;

endmodule
```

Example 2: Testbench (Binary Adder)

```
// Testbench file of Dataflow Binary Adder

`timescale 1 ns / 1 ps
module tb_BinAdder_4bit ();

   // declare inputs as reg types
   reg   [3:0] A, B;
   reg         C_in;

   // outputs as wire types
   wire  [3:0] Sum;
   wire        C_out;

   // instantiate UUT
   BinAdder_4bit UUT ( .A(A), .B(B), .C_in(C_in), .Sum(Sum), .C_out(C_out) );

   // generate stimuli
   initial begin
      $display("Starting simulation at %0d ns...", $time);
      A = 4'd0;   B = 4'd0;   C_in = 1'b0;   #5
      A = 4'd0;   B = 4'd0;   C_in = 1'b1;   #5
      A = 4'd5;   B = 4'd10;  C_in = 1'b0;   #5
      A = 4'd3;   B = 4'd4;   C_in = 1'b1;   #5
      A = 4'd11;  B = 4'd2;   C_in = 1'b0;   #5
      A = 4'd14;  B = 4'd15;  C_in = 1'b1;   #5
      A = 4'd15;  B = 4'd15;  C_in = 1'b0;   #5
      A = 4'd6;   B = 4'd9;   C_in = 1'b1;   #5

      $display("Finished simulation at %0d ns.", $time);
      $stop;
   end

   // response monitor
   initial begin
      $monitor("Time = %2d ns\t A = %b\t= %d\t B = %b\t= %d\t C_in = %b\t C_out = %b\t Sum = %b\t = %d\t {C_out, Sum} = %2d",
               $time, A, A, B, B, C_in, C_out, Sum, Sum, (Sum+(C_out*(16))) );
   end

endmodule
```

Example 2: Simulation (Binary Adder)

```
# Starting simulation at 0 ns...
# Time =  0 ns  A = 0000 =  0  B = 0000 =  0  C_in = 0  C_out = 0  Sum = 0000 =  0  {C_out, Sum} =  0
# Time =  5 ns  A = 0000 =  0  B = 0000 =  0  C_in = 1  C_out = 0  Sum = 0001 =  1  {C_out, Sum} =  1
# Time = 10 ns  A = 0101 =  5  B = 1010 = 10  C_in = 0  C_out = 0  Sum = 1111 = 15  {C_out, Sum} = 15
# Time = 15 ns  A = 0011 =  3  B = 0100 =  4  C_in = 1  C_out = 0  Sum = 1000 =  8  {C_out, Sum} =  8
# Time = 20 ns  A = 1011 = 11  B = 0010 =  2  C_in = 0  C_out = 0  Sum = 1101 = 13  {C_out, Sum} = 13
# Time = 25 ns  A = 1110 = 14  B = 1111 = 15  C_in = 1  C_out = 1  Sum = 1110 = 14  {C_out, Sum} = 30
# Time = 30 ns  A = 1111 = 15  B = 1111 = 15  C_in = 0  C_out = 1  Sum = 1110 = 14  {C_out, Sum} = 30
# Time = 35 ns  A = 0110 =  6  B = 1001 =  9  C_in = 1  C_out = 1  Sum = 0000 =  0  {C_out, Sum} = 16
# Finished simulation at 40 ns.
```



Change RADIX to UNSIGNED *(not Decimal: Signed)*

Example 3: 4-bit Binary Adder (v2)

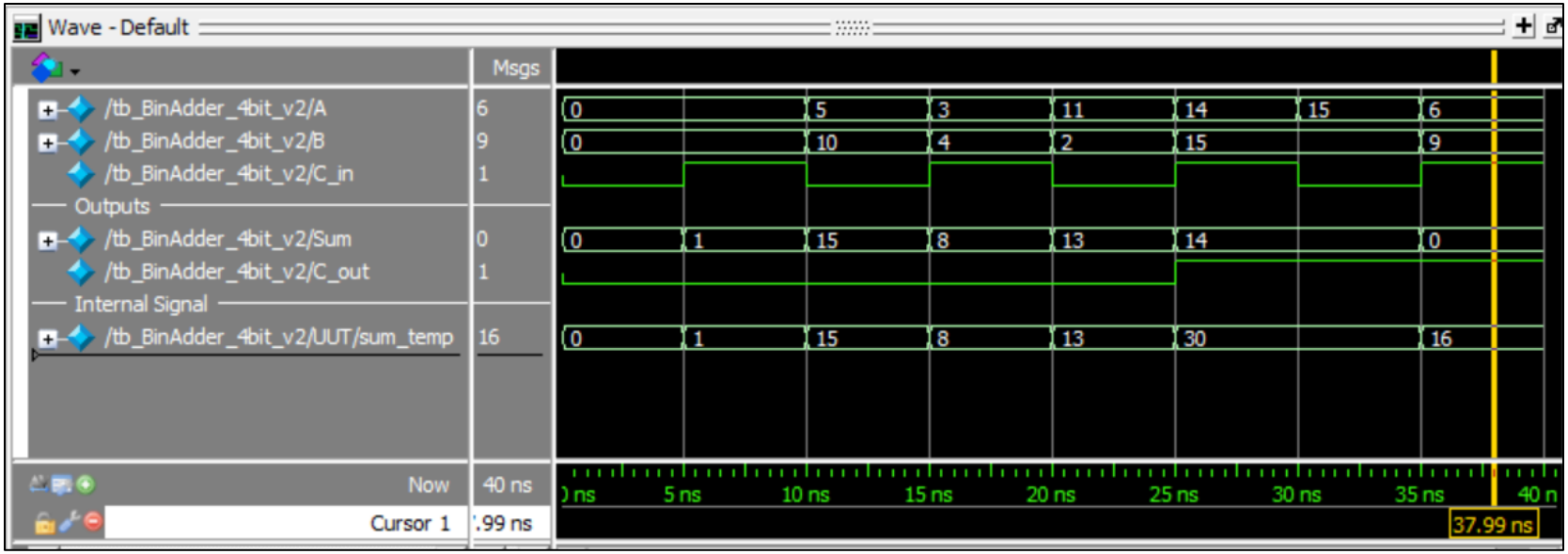
- Verilog dataflow description of a 4-bit binary adder using ***hard literals:***

- Design Entry:

```
// Dataflow model of a 4-bit Binary Adder (v2)
module BinAdder_4bit_v2 (
    input    [3:0] A, B,
    input          C_in,
    output   [3:0] Sum,
    output         C_out
);

    // internal signal
    wire [4:0]  sum_temp;

    assign sum_temp   = {1'b0, A} + {1'b0, B} + {4'b0000, C_in};
    assign Sum        = sum_temp[3:0];
    assign C_out      = sum_temp[4];

endmodule
```

Example 3: Simulation (Binary Adder v2)

```
# Starting simulation at 0 ns...
# Time =  0 ns  A = 0000 =  0  B = 0000 =  0  C_in = 0  C_out = 0  Sum = 0000  =  0  {C_out, Sum} =  0
# Time =  5 ns  A = 0000 =  0  B = 0000 =  0  C_in = 1  C_out = 0  Sum = 0001  =  1  {C_out, Sum} =  1
# Time = 10 ns  A = 0101 =  5  B = 1010 = 10  C_in = 0  C_out = 0  Sum = 1111  = 15  {C_out, Sum} = 15
# Time = 15 ns  A = 0011 =  3  B = 0100 =  4  C_in = 1  C_out = 0  Sum = 1000  =  8  {C_out, Sum} =  8
# Time = 20 ns  A = 1011 = 11  B = 0010 =  2  C_in = 0  C_out = 0  Sum = 1101  = 13  {C_out, Sum} = 13
# Time = 25 ns  A = 1110 = 14  B = 1111 = 15  C_in = 1  C_out = 1  Sum = 1110  = 14  {C_out, Sum} = 30
# Time = 30 ns  A = 1111 = 15  B = 1111 = 15  C_in = 0  C_out = 1  Sum = 1110  = 14  {C_out, Sum} = 30
# Time = 35 ns  A = 0110 =  6  B = 1001 =  9  C_in = 1  C_out = 1  Sum = 0000  =  0  {C_out, Sum} = 16
# Finished simulation at 40 ns.
```



Change RADIX to UNSIGNED

Constants

- Constant values are frequently found in expressions and array boundaries
- Fixed within the module and can't be modified
- Good design practice: replace "hard literals" with symbolic constants
- Declared using the ***localparam*** keyword *(local parameter)*
- Examples:

```
localparam      DATA_WIDTH = 8,
                DATA_RANGE = 2**DATA_WIDTH - 1;

localparam      UART_PORT      = 4'b0001,
                LCD_PORT       = 4'b0010,
                MOUSE_PORT     = 4'b0100;
```

Example 4: 4-bit Binary Adder (v3)

- Verilog dataflow description of a 4-bit binary adder using *constants (localparam)*:


```
// Dataflow model of a 4-bit Binary Adder (v3)
//   using constants (localparam)
//
module BinAdder_4bit_v3 (
    input    [3:0] A, B,
    input          C_in,
    output   [3:0] Sum,
    output         C_out
);

    // constant declarations
    localparam  N = 4;

    // internal signal
    wire [N:0]  sum_temp;

    assign sum_temp   = {1'b0, A} + {1'b0, B} + {4'b0000, C_in};
    assign Sum        = sum_temp[N-1:0];
    assign C_out      = sum_temp[N];

endmodule
```

Useful for allowing the Verilog code to be easily modified *(i.e., changing the design to an 8-bit adder, etc.)*

Parameters

- A Verilog module can be instantiated as a component and becomes a part of a larger design *(through structural modeling)*
- ***Parameter*** is another way of declaring constants, which passes that constant value to a module
- Cannot be modified inside the module *(acts like a constant)*
  - Functions like a **"global" version of a *localparam*** *(constant)*
- Default value must be set inside the module but can be overridden once the design is synthesized
- This makes the module versatile, reusable, and scalable

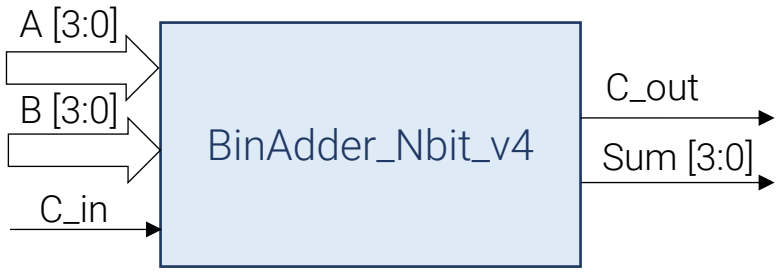
Parameters

- Syntax:

```
module [module_name]
    #(  parameter [parameter_name] = [default_value];
           …
        parameter [parameter_name] = [default_value];
    )
    (       // Port declarations
    );
```

Example 5: 4-bit Binary Adder (v4)

- Verilog dataflow description of a 4-bit binary adder using *parameters*:




```
// Dataflow model of a N-bit Binary Adder (v4)
//   using parameters
//
module BinAdder_Nbit_v4
   #(parameter N = 4)    // default value = 4
(
   input     [N-1:0]   A, B,
   input               C_in,
   output    [N-1:0]   Sum,
   output              C_out
);

   // constant declarations
   // localparam here ...

   // internal signals
   wire [N:0]  sum_temp;

   assign sum_temp   = {1'b0, A} + {1'b0, B} + { {(N) {1'b0}}, C_in};
   assign Sum        = sum_temp[N-1:0];
   assign C_out      = sum_temp[N];

endmodule
```

Useful for allowing the Verilog code to be easily modified **from outside the module** *(i.e., changing the design to an 8-bit adder, etc.)*

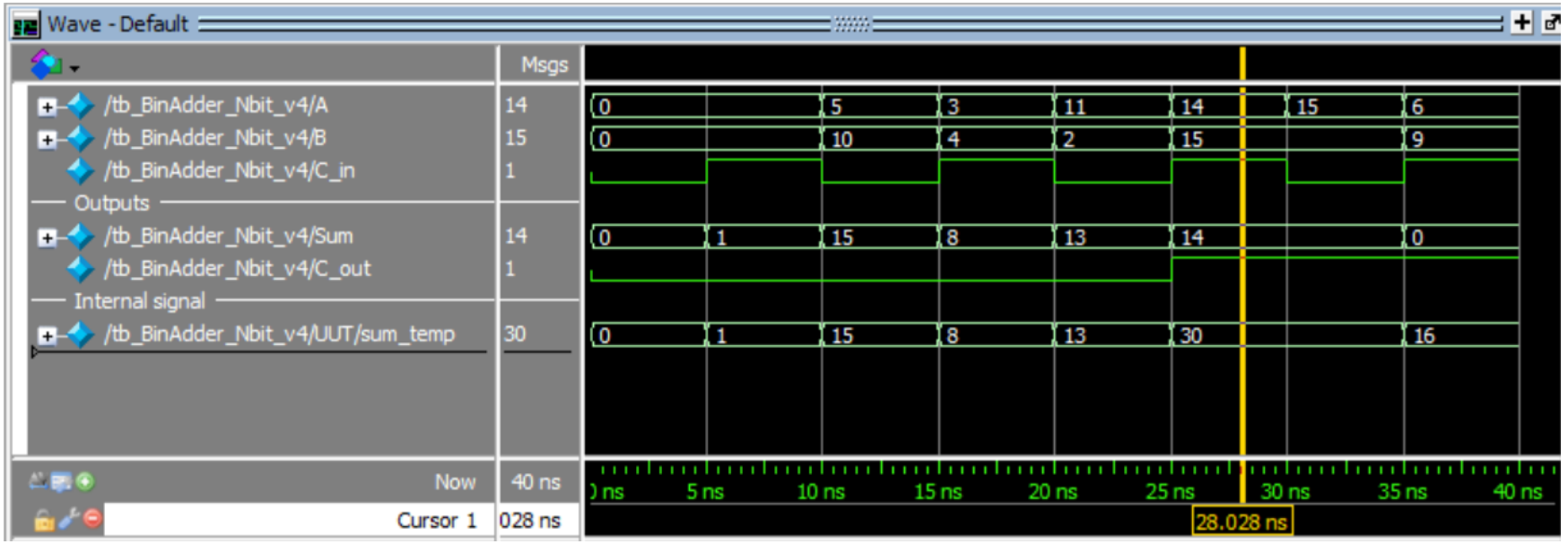
Example 5: Testbench (Binary Adder v4)

```
// Testbench file of Dataflow Binary Adder (v4)
`timescale 1 ns / 1 ps
module tb_BinAdder_Nbit_v4 ();

   // declare inputs as reg types
   reg   [3:0] A, B;
   reg         C_in;

   // outputs as wire types
   wire  [3:0] Sum;
   wire        C_out;

   // instantiate UUT
   BinAdder_Nbit_v4 #( .N(4) )
      UUT ( .A(A), .B(B), .C_in(C_in), .Sum(Sum), .C_out(C_out) );

   // generate stimuli
   initial begin
      $display("Starting simulation at %0d ns...", $time);
      A = 4'd0;   B = 4'd0;   C_in = 1'b0;   #5
      A = 4'd0;   B = 4'd0;   C_in = 1'b1;   #5
      A = 4'd5;   B = 4'd10;  C_in = 1'b0;   #5
      A = 4'd3;   B = 4'd4;   C_in = 1'b1;   #5
      A = 4'd11;  B = 4'd2;   C_in = 1'b0;   #5
      A = 4'd14;  B = 4'd15;  C_in = 1'b1;   #5
      A = 4'd15;  B = 4'd15;  C_in = 1'b0;   #5
      A = 4'd6;   B = 4'd9;   C_in = 1'b1;   #5

      $display("Finished simulation at %0d ns.", $time);
      $stop;
   end

   // response monitor
   initial begin
      $monitor("Time = %2d ns\t A = %b\t= %d\t B = %b\t= %d\t C_in = %b\t C_out = %b\t Sum = %b\t = %d\t {C_out, Sum} = %2d",
               $time, A, A, B, B, C_in, C_out, Sum, Sum, (Sum+(C_out*(16))) );
   end

endmodule
```

Useful for allowing the Verilog code to be easily modified **from outside the module** ***(in this case: from the testbench)***

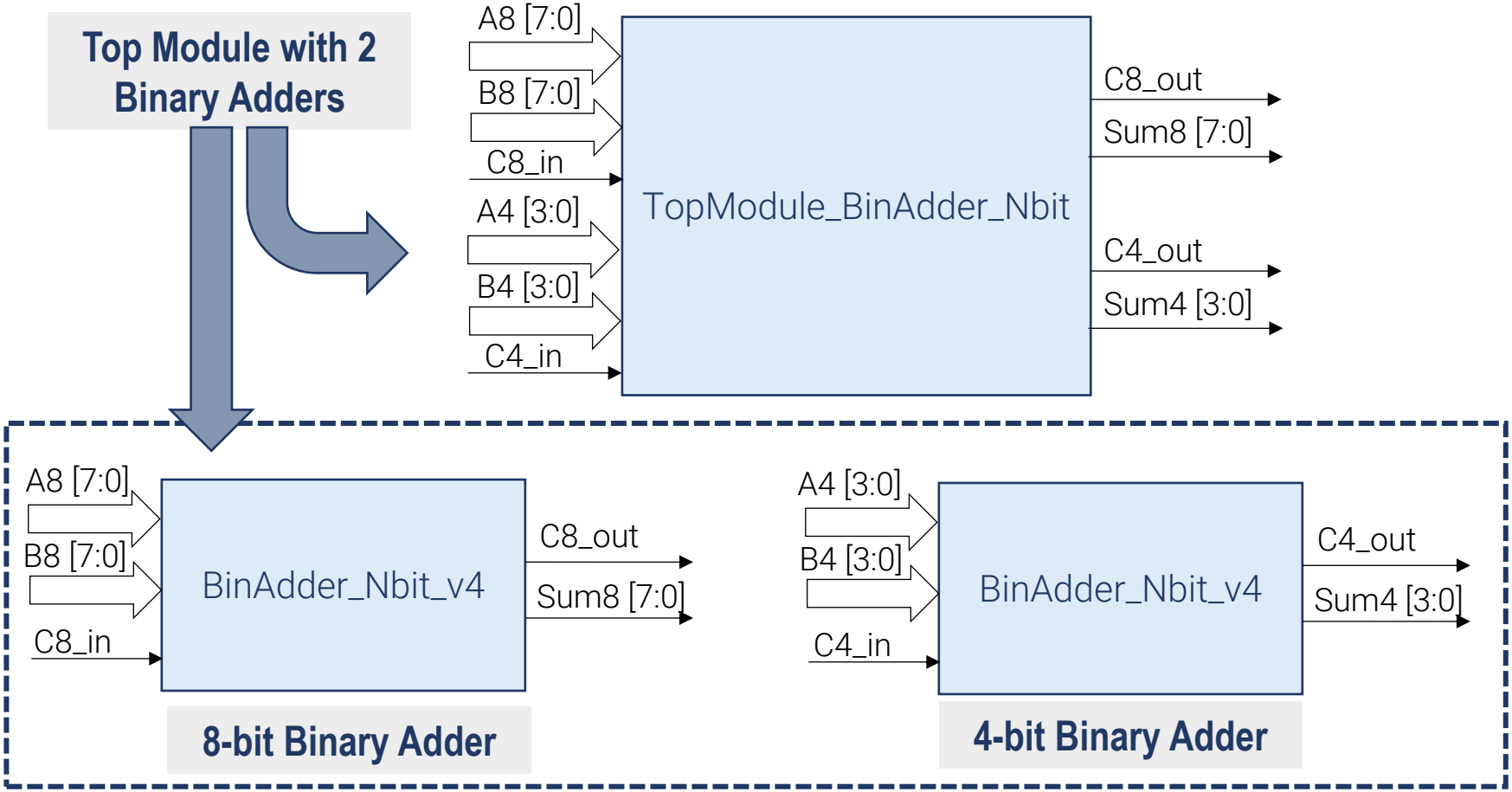
Example 5: Simulation (Binary Adder v4)

- Using N = 4: *4-bit Binary Adder*



Example 6: Reusing Parameterized Binary Adder

- Verilog structural description of *instantiating* the binary adders (v4):



Example 6: Reusing Parameterized Binary Adder

- Verilog structural description of *instantiating* the binary adders (v4):

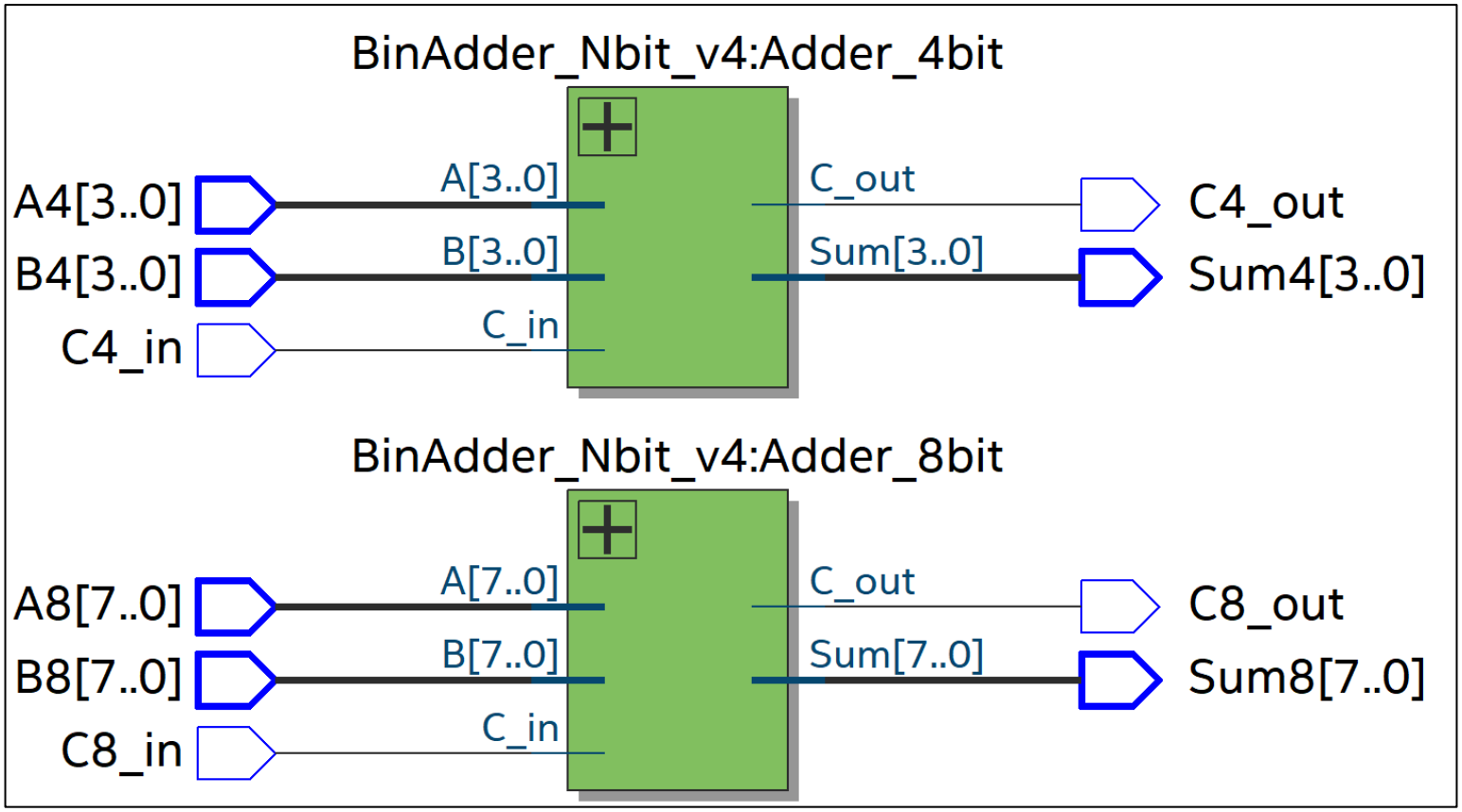
TopModule_BinAdder_Nbit.v | Compilation Report - TopModule_BinAdder_Nbit

```
1  // Top Module instantiating 2 N-bit Binary Adders (v4)
2  //   using parameters
3  //
4  module TopModule_BinAdder_Nbit (
5     // ports for 4-bit Bin Adder
6     input    [3:0] A4, B4,
7     input          C4_in,
8     output   [3:0] Sum4,
9     output         C4_out,
10    // ports for 8-bit Bin Adder
11    input    [7:0] A8, B8,
12    input          C8_in,
13    output   [7:0] Sum8,
14    output         C8_out
15 );
16
17    // instantiate 8-bit adder
18    BinAdder_Nbit_v4 #( .N(8) )
19       Adder_8bit ( .A(A8), .B(B8), .C_in(C8_in), .Sum(Sum8), .C_out(C8_out) );
20
21    // instantiate 4-bit adder
22    BinAdder_Nbit_v4 #( .N(4) )
23       Adder_4bit ( .A(A4), .B(B4), .C_in(C4_in), .Sum(Sum4), .C_out(C4_out) );
24
25 endmodule
26
```

Useful for allowing the Verilog code to be easily modified **from outside the module** ***(in this case: from an external module)***

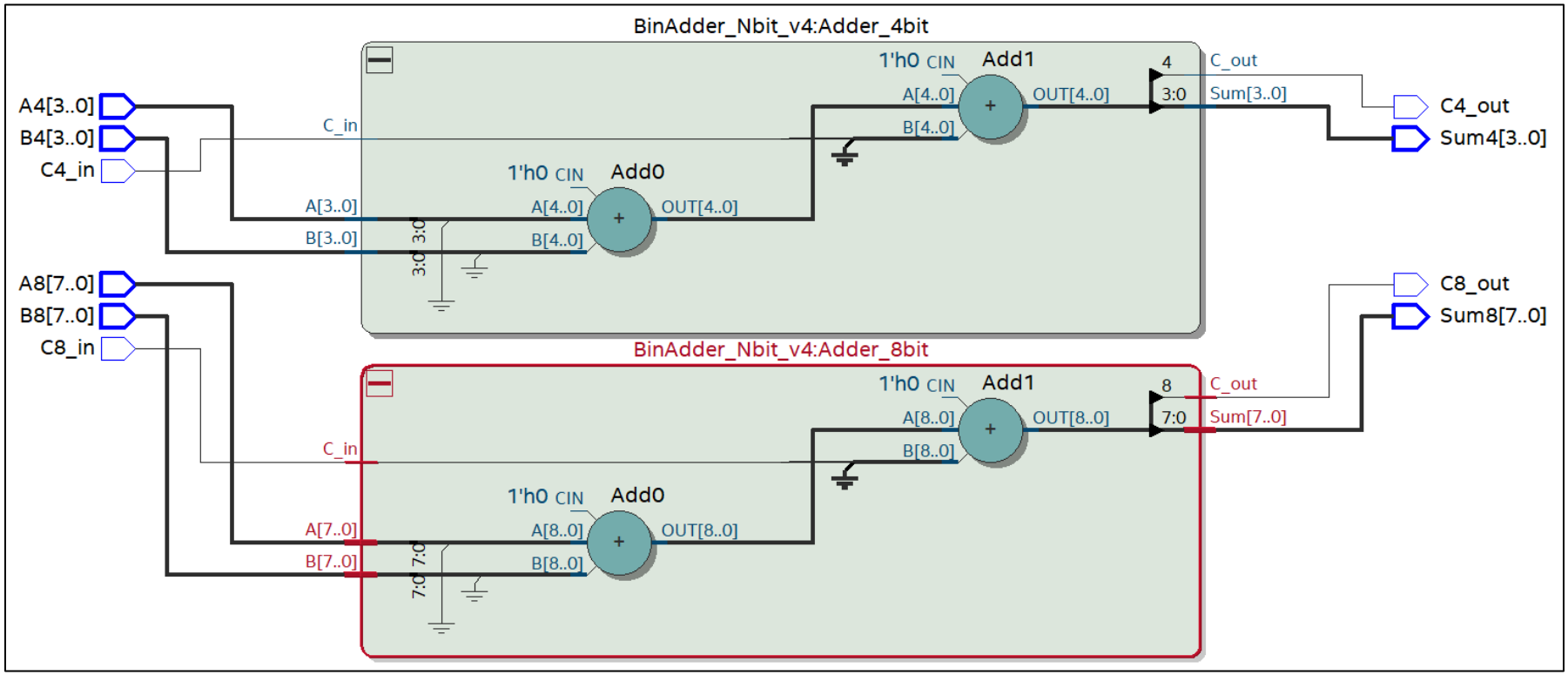
Example 6: Synthesis (Parameterized Bin Adder)

- RTL Schematic View:



Example 6: Synthesis (Parameterized Bin Adder)

- RTL Schematic View:



End of Unit 4