

# **ECE 351**

## Verilog and FPGA Design

---

### **Lecture 5:   Module port declarations Packed and unpacked arrays Parameters**

Roy Kravitz

Electrical and Computer Engineering Department

Maseeh College of Engineering and Computer Science



---

## Module and port declarations

Source material drawn from:

- Mark F. And Roy K. ECE 571 lecture slides
- *RTL Modeling with SystemVerilog* by Stuart Sutherland

## Port declarations

3

- Module definitions (except for testbenches) include a port list enclosed in parenthesis
- Modules can have 4 types of ports:
  - **input** – single value or user-defined type that is coming into the module
  - **output** – single value or user-defined type that is coming out of the module
  - **inout** – bidirectional single value or user-defined type that can be both and input and and output of the module
  - **Interfaces** – compound port that can communicate a collection of several values
- A port declaration defines a port's direction, type, data type, size, and name
- Ports can be listed in any order
- The port names are what you connect signals to when you instantiate a module

## Coding styles for port declarations

4

- Combined style (also called ANSI C-style port list) – contains full declaration of each port within the port list parenthesis
  - Preferred style for most engineers, including myself
  - Port declarations are separated by commas (,) except the last port does not have a comma before the closing parenthesis

```
module alu (  
    input wire logic signed [31:0] a,           // 32-bit input  
    input wire logic signed [31:0] b,           // 32-bit input  
    input wire logic [3:0] opcode,              // 4-bit input  
    output var logic signed [31:0] result,      // 32-bit output  
    output var logic overflow,                  // 1-bit output  
    output var logic error                      // 1-bit output  
);
```

## Coding styles for port declarations (cont'd)

5

- Legacy style (also called non-ANSI C-style port list) – separates the port list from the declarations of type, data bypte, sign and size for each port
  - Was the only form allowed in the earlier days of Verilog (Verilog 1995)

```
module alu (a, b, opcode, result, overflow, error);

input [31:0] a, b;           // 32-bit inputs
input [3:0] opcode;         // 4-bit input
output [31:0] result;       // 32-bit output
output overflow, error;     // 1-bit outputs

wire signed [31:0] a, b;    // 32-bit nets
wire [3:0] opcode;         // 4-bit net
reg signed [31:0] result;   // 32-bit variable
reg overflow, error;       // 1-bit variables
```

## Coding styles for port declarations (cont'd)

6

- Legacy style w/ combined direction and size – combines the direction declaration and the type/data type declaration into a single statement
  - Added to Verilog 2001, still in use on some legacy code I've looked at

```
module alu_4 (a, b, opcode, result, overflow, error);

input wire signed [31:0] a, b;    // 32-bit inputs
input wire [3:0] opcode;         // 4-bit input
output reg signed [31:0] result;  // 32-bit output
output reg overflow, error;      // 1-bit output
```

## Module port defaults

7

- Implicit defaults for each port's direction, type, data type, signedness, and size
  - Port type can be a net (ex: wire) or variable (ex: var)
  - Port data type can be logic (4-state) or bit (2-state)
- Defaults:
  - No direction – inout but only until a direction has been defined. After that the direction applies to all subsequent ports until a new direction is specified
  - No type – default type is wire when no data type (e: logic) is specified. Once a data type is specified, default type is wire for input and inout and var for output ports
  - No data type – default type is logic (4-state)
  - No signedness – reg, logic, bit and time default to unsigned. byte, shortint, int, integer and longint default to signed
  - No size – default size is the default size of the data type
- An explicit declaration of a port's direction, type, data type, signedness or size is "sticky" (e.g remains until changed)

## Best practices for module port lists

8

- Use ANSI-style declarations
- Declare both input and output ports as a logic type
- Sutherland's recommendations (and I agree):
  - Use combined ANSI-C style port
  - Declare the direction of each port
  - Declare all port data types as logic
  - Avoid 2-state data types in RTL models
  - Allow the language to infer a wire or var type except for tri-state ports to help make your design intent more readable
  - Declare each port on a separate line except, perhaps, if port names have the same direction, data type, size and similar usage (ex: ain and bin as inputs to an adder)

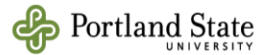
## Best practices for module port lists (cont'd)

9

```
module alu (  
    input logic signed [31:0] a, b,           // ALU operand inputs  
    input logic [3:0] opcode,                 // ALU operation code  
    output logic signed [31:0] result,         // Operation result  
    output logic overflow,                    // Set if result overflow  
    output logic error                        // Set if operation error  
);  
  
////////////////////////////////////  
// model functionality not shown          //  
////////////////////////////////////  
endmodule: alu
```

ECE 351 Verilog and FPGA Design

Source: Sutherland: Ex: 3-3 (Kindle Edition)



## Packed and unpacked arrays

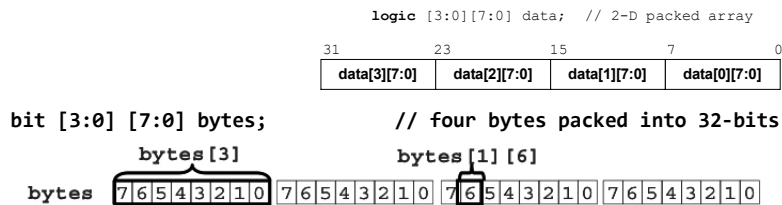
Source material drawn from:

- Mark F. And Roy K. ECE 571 lecture slides
- *SystemVerilog for Design, 2<sup>nd</sup> Edition* by Stuart Sutherland
- *RTL Modeling with System Verilog* by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas

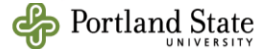
## Packed arrays

11

- SystemVerilog supports packed bit dimensions:
  - Packed bits are also called **vectors**
    - ex: logic [7:0] avar declares an 8-bit vector with the bits numbered left to right as 7..0
  - Packed arrays build on packed bits and allow for multiple dimensions
    - Ex: logic [3:0][7:0] a2dvar declares a 32-bit vector with 4 elements, each 8-bits wide
    - All of the bits are adjacent to each other in a single 32-bit vector
    - Can be addressed as a whole, in subparts, in part select



ECE 351 Verilog and FPGA Design



## Initializing packed arrays

12

- Can be of different sizes when copied
  - Truncated/extended on left
 

```

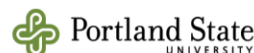
logic [3:0][15:0] a, b, result; // packed arrays
...
result = (a << 1) + b;

logic [3:0][7:0] a = 32'h0;           // vector assignment
logic [3:0][7:0] b = {16'hz,16'h0}; // concatenate operator
logic [3:0][7:0] c = {16{2'b01}};    // replicate operator


bit   [1:0][15:0] a; // 32 bit 2-state vector
logic [3:0][ 7:0] b; // 32 bit 4-state vector
logic [15:0]      c; // 16 bit 4-state vector
logic [39:0]      d; // 40 bit 4-state vector

b = a; // assign 32-bit array to 32-bit array
c = a; // upper 16 bits will be truncated
d = a; // upper 8 bits will be zero filled
                    
```

ECE 351 Verilog and FPGA Design



- |             |                     |  |  |   |   |   |   |   |   |   |   |   |   |   |   |
|-------------|---------------------|--|--|---|---|---|---|---|---|---|---|---|---|---|---|
| b_unpack[0] |                     |  |  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |   |   |   |   |
| b_unpack[1] | <b>Unused space</b> |  |  |   |   |   |   | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| b_unpack[2] |                     |  |  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |   |   |   |   |



Portland State  
UNIVERSITY

## Initializing unpacked arrays (cont'd)

15

### Initialization

```
int d [0:1][0:3] = '{ {7,3,0,5}, {2,0,1,6} }';
// d[0][0] = 7
// d[0][1] = 3
// d[0][2] = 0
// d[0][3] = 5

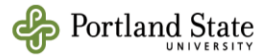
// d[1][0] = 2
// d[1][1] = 0
// d[1][2] = 1
// d[1][3] = 6

int e [0:1][0:3] = '{ 2{7,3,0,5} }';
// e[0][0] = 7
// e[0][1] = 3
// e[0][2] = 0
// e[0][3] = 5

// e[1][0] = 7
// e[1][1] = 3
// e[1][2] = 0
// e[1][3] = 5

int a1 [0:7][0:1023] = '{default:8'h55};
```

ECE 351 Verilog and FPGA Design



## Assigning to unpacked arrays

16

### Assignment: individual or slice

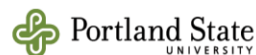
```
byte a [0:3][0:3];

a[1][0] = 8'h5; // assign to one element
a = '{ {0,1,2,3},
      {4,5,6,7},
      {7,6,5,4},
      {3,2,1,0} };
// assign a list of values to the full array

a[3] = '{hF, hA, hC, hE};
// assign list of values to slice of the array

always @(posedge clock, negedge resetN)
  if (!resetN) begin
    a = '{default:0}; // init entire array
    a[0] = '{default:4}; // init slice of array
  end
  else begin
    //...
  end
```

ECE 351 Verilog and FPGA Design





## Assigning to unpacked arrays (cont'd)

17

- Arrays can be copied but must be same size/dimension

```
logic [31:0] a [2:0][9:0];
logic [0:31] b [1:3][1:10];

a = b; // assign unpacked array to unpacked
      // array
```

## Iterating array accesses

18

- **for** and **foreach**

```
initial begin
    bit [31:0] src[5], dst[5];
    for (int i=0; i<$size(src); i++)
        src[i] = i;
    foreach (dst[j])
        dst[j] = src[j] * 2; // dst doubles src values
end

int md[2][3] = '{0,1,2}, {3,4,5}';
initial begin
    $display("Initial value:");
    foreach (md[i,j]) // Yes, this is the right syntax
        $display("md[%0d][%0d] = %0d", i, j, md[i][j]);

    $display("New value:");
    // Replicate last 3 values of 5
    md = '{9, 8, 7}, {3{5}}';
    foreach (md[i,j]) // Yes, this is the right syntax
        $display("md[%0d][%0d] = %0d", i, j, md[i][j]);
end
```

Initial value:	md[0][0] = 0
	md[0][1] = 1
	md[0][2] = 2
	md[1][0] = 3
	md[1][1] = 4
	md[1][2] = 5
New value:	md[0][0] = 9
	md[0][1] = 8
	md[0][2] = 7
	md[1][0] = 5
	md[1][1] = 5
	md[1][2] = 5

## More on two dimensional arrays

19

```
initial begin
  byte twoD[4][6];
  foreach(twoD[i,j])
    twoD[i][j] = i*10+j;

  foreach (twoD[i]) begin // Step through first dim.
    $write("%2d:", i);
    foreach(twoD[,j]) // Step through second
      $write("%3d", twoD[i][j]);
    $display;
  end
end
```

```
0:  0  1  2  3  4  5
1: 10 11 12 13 14 15
2: 20 21 22 23 24 25
3: 30 31 32 33 34 35
```

ECE 351 Verilog and FPGA Design



## Mixing packed and unpacked arrays

20

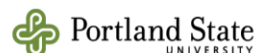
```
packed          unpacked
bit [3:0] [7:0] barray [3]; // Packed: 3x32-bit
bit [31:0] lw = 32'h0123_4567; // Word
bit [7:0] [3:0] nibbles; // Packed array of nibbles
barray[0] = lw;
barray[0][3] = 8'h01;
barray[0][1][6] = 1'b1;
nibbles = barray[0]; // Copy packed values
```

Diagram illustrating the memory layout of the `barray` after the assignments:

	barray[0][3]				barray[0][1][6]																			
barray[0]	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
barray[1]	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
barray[2]	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

- ❑ Packed arrays can contain only bit-wise types (logic, bit, reg) and other packed arrays, structs, or unions or net types (wire, etc.)
- ❑ Unpacked arrays can contain any type (e.g. string, unpacked structs, even instances of interfaces)

ECE 351 Verilog and FPGA Design



## Aggregate Compare and Copy

21

Compare (equality/inequality only) and copy arrays without loops

```
initial begin
    bit [31:0] src[5] = '{0,1,2,3,4}',
              dst[5] = '{5,4,3,2,1}';

    // Aggregate compare the two arrays
    if (src==dst)
        $display("src == dst");
    else
        $display("src != dst");

    // Aggregate copy all src values to dst
    dst = src;

    // Change just one element
    src[0] = 5;

    // Are all values equal (no!)
    $display("src %s dst", (src == dst) ? "==" : "!=");

    // Use array slice to compare elements 1-4
    // $display("src[1:4] %s dst [1:4]",
    //         src[1:4] == dst[1:4] ? "==" : "!=");
end
```

ECE 351 Verilog and FPGA Design



## Using \$bits(expression)

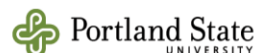
22

- System function **\$bits()** returns number of bits represented by expression
  - Expression can contain any type of value, including packed or unpacked arrays, structures, unions
  - Similar to C **sizeof()**
  - Synthesizable if argument is not dynamically sized (i.e. value of **\$bits()** can be determined at elaboration time)

```
bit [63:0] a;
logic [63:0] b;
wire [3:0][7:0] c [0:15];
struct packed {byte tag; logic [31:0] addr;} d;

$bits(a) // returns 64
$bits(b) // returns 64
$bits(c) // returns 512
$bits(d) // returns 40
```

ECE 351 Verilog and FPGA Design



---

## Parameters

Source material drawn from:

- Mark F. And Roy K. ECE 571 lecture slides
- Roy's ECE 351 and Verilog workshop lecture slides
- *SystemVerilog for Design, 2<sup>nd</sup> Edition* by Stuart Sutherland
- RTL Modeling with SystemVerilog by Stuart Sutherland
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas

## Parameters

24

- ❑ Assigns a value to a symbolic name
- ❑ Used to parameterize modules
- ❑ Used as default values when module is instantiated
- ❑ Can be redefined at elaboration time
- ❑ Can be changed when module is instantiated
- ❑ Redefined also with `defparam`

```
module parity (y, in);  
    parameter size = 8;  
    input [size-1:0] in;  
    output y;  
  
    assign y = ^in;  
endmodule  
  
module top();  
    logic [15:0] data;  
  
    // instantiate parity  
    parity #(16) i0 (y, data);
```

## Parameters (cont'd)

25

- Parameters included in module port list:

```
module adder #(parameter MSB = 32, LSB = 0)
    (output logic [MSB:LSB] sum;
     output logic co;
     input wire [MSB:LSB] a, b;
     input wire          ci;
     ...
endmodule
```

- Parameters can (and should be) be sized and typed

```
parameter [31:0] A;          // unsigned 32-bit parameter
parameter signed [63:0] B;   // signed 64-bit parameter
```

## Local Parameters

26

- True constant

- Cannot be redefined at elaboration time
- Cannot be redefined outside the module

```
module paramram(d, a, wr, q);
    parameter D_WIDTH = 8, A_WIDTH = 8;
    localparam DEPTH = (2**A_WIDTH)-1;
    input [D_WIDTH-1:0] d;
    input [A_WIDTH-1:0] a;
    input wr;
    output wire [D_WIDTH-1:0] q;

    logic [D_WIDTH-1:0] mem [DEPTH:0]; // declare memory array

    always @(wr) mem[a] = d; // write
    assign q = mem[a]; //read
endmodule
```

## `define

27

### ☐ `define

- defines a global text substitution
- Fixed at elaboration time – cannot be redefined

```
`define size 8

Logic [`size - 1:0] a, b;

always @(posedge clk)
    a <= b;
```

This is NOT the same as a parameter...if you're familiar w/ C it'd be better to think this way:

- parameter <-> const
- `define <-> #define

## Next Time

28

### ☐ Topics:

- Packages
- User-defined types including typedef, enum, struct, union
- RTL operators and expressions

### ☐ You should:

- Read Sutherland Ch 4

### ☐ Homework, projects and quizzes

- Homework #1 has been posted. Should be completed by 10:00 PM on Wed, 22-Apr
  - ☐ (60 pts) True/False, multiple choice and short answers. Should be completed online on D2L
  - ☐ (40 pts) SystemVerilog programming and simulation. Should be submitted to your Homework #1 dropbox on D2L
    - The .sv source code that you wrote
    - QuestaSim (or ModelSim) transcript showing that your design work correctly