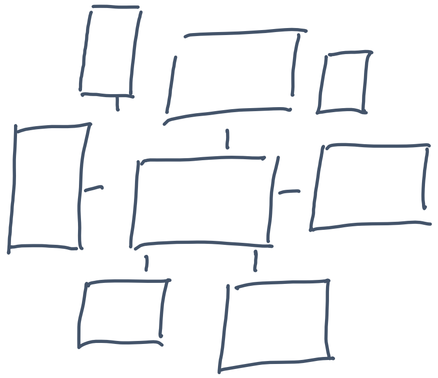
More on Introduction to SystemVerilog

Today's Agenda

Intended topics for today's session

- More on lexical rules, data types, user defined data types, logic values, number representation, packages



FPGA-based SoC Design

International Master of Science in Electrical Engineering

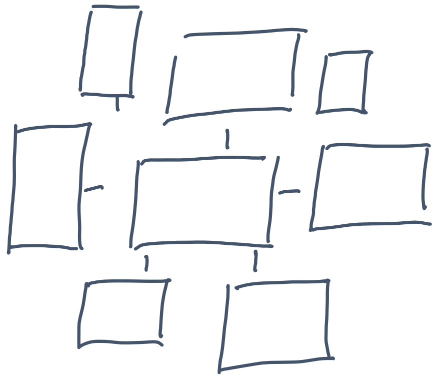
Prof. Dr. C. Jakob
University of Applied Sciences Darmstadt
h_da
Faculty of Electrical Engineering and Information Technology
fbeit

More on Introduction to SystemVerilog

Objectives

By the end of this lecture you will be able to …

- … apply a wider range of SystemVerilog features in order to describe and implement more complex digital circuitries.



FPGA-based SoC Design

International Master of Science in Electrical Engineering

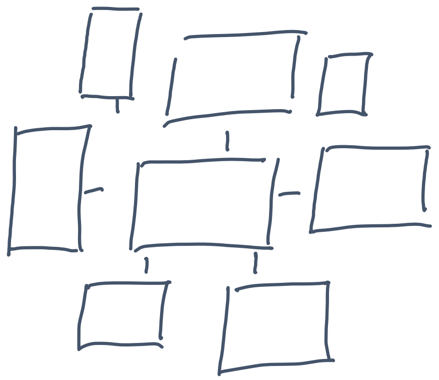
Prof. Dr. C. Jakob
University of Applied Sciences Darmstadt
h_da
Faculty of Electrical Engineering and Information Technology
fbeit

More on Introduction to SystemVerilog

Recommended Readings

Textbooks, Application Notes, White Papers ...

- Sutherland, S., “RTL Modeling with SystemVerilog for Simulation and Synthesis: Using SystemVerilog for ASIC and FPGA Design”, CreateSpace Independent Publishing Platform, 2017
- Spear, C., “SystemVerilog for Verification: A Guide to Learning the Testbench Language Features”, Springer, 3rd edition, 2012.



FPGA-based SoC Design

International Master of Science in Electrical Engineering

Prof. Dr. C. Jakob
University of Applied Sciences Darmstadt
h_da
Faculty of Electrical Engineering and Information Technology
fbeit

If I designed a computer with 200 chips, I tried to design it with 150. And then I would try to design it with 100. I just tried to find every trick I could in life to design things real tiny.

Stephen Gary Wozniak, 'the Woz', co-founder of Apple Inc. and *computer pioneer*

Introduction to SystemVerilog HDL – Part #6

International Master of Science in Electrical Engineering

Prof. Dr. C. Jakob
University of Applied Sciences Darmstadt
h_da
Faculty of Electrical Engineering and Information Technology
fbeit

SystemVerilog - Lexical Rules

FPGA-based SoC Design

Lexical Rules

1/1

```
/*
     Multiline comment ...
*/
module xor_8_reduction(
  input  logic [3:0] d, output logic q
  );
  // Single line comment
  assign q = ^d;

endmodule
```

[SystemVerilog] Source Code: xor_8_reduction.sv

Note

- SystemVerilog is **case-sensitive**, therefore upper and lower case letters have different meanings: D is not d!
- Moreover, SystemVerilog is a free formatting language, what means that spaces can be added freely to format the source code.
- Comments in SystemVerilog are just like those in C.

SystemVerilog - Data Types

IEEE Standard 1800-2012

Basic Data Types

bit	User defined vector size; defaults to unsigned
byte	8-bit signed intefer or ASCII character
shortint	16-bit signed integer
int	32-bit signed integer
longint	64-bit signed integer

Table: **2-state** Data Types (only 0 or 1)

logic	User defined vector size; defaults to unsigned
reg	User defined vector size; defaults to unsigned
integer	32-bit signed integer
time	64-bit unsigned integer

Table: **4-state** Data Types (0,1,x,z)

Note

- 2-state data types simulate faster and consume less memory than 4-state data types
- However, 2-state data types in simulation may not accurately model the actual hardware implementation

- **This new data type is intended to remove confusion associated with the reg data type in Verilog.** It was sometimes mistakenly understood that a register of the given variable name should be created in hardware.
- However, there is no correlation between using a reg variable and the hardware inferred in Verilog.
- **The context of the reg variable** determines whether combinational or sequential logic is inferred.

Basic Data Types

`bit`	User defined vector size; defaults to unsigned
`byte`	8-bit signed intefer or ASCII character
`shortint`	16-bit signed integer
`int`	32-bit signed integer
`longint`	64-bit signed integer

Table: 2-state Data Types (only 0 or 1)

`logic`	User defined vector size; defaults to unsigned
`reg`	User defined vector size; defaults to unsigned
`integer`	32-bit signed integer
`time`	64-bit unsigned integer

Table: 4-state Data Types (0,1,x,z)

Assigning the same value to all bits of a variable …

1/1

```
unsigned int data;
...
assign data = ‘0;
```

[SystemVerilog] Source Code Excerpt

Note

- The respective value is assigned to all bit positions

Strings and String Methods in SystemVerilog

IEEE Standard 1800-2012

Strings

1/1

```
module demo_testbench();
  // string declaration and initialization
  string message = “Simulation starts!”;

  initial
    begin : testbench_scheduler
      $display(%s”,message);
      ...
    end

endmodule
```

[SystemVerilog] Source Code Excerpt

Note

- The data typ string represents an ordered set of characters.
- The length of a string variable is equal to the number of characters in the set.
- The length can dynamically vary throughout the course of a simulation.
- Commonly used in testbenches

Strings Methods available

1/1

```
module demo_design #(parameter string mode = "standard")  (
  input logic clk, reset_n,
  input logic [7:0] data
  ...
);

localparam SETTINGS = (mode.icompare("standard") == 2) ? 12 : 15;

...
```

built-in method notation

[SystemVerilog] Source Code Excerpt

Note

- The Intel Quartus Prime software supports string data types for parameters.
- SystemVerilog also supports a number of special methods to work with strings. .
- These methods use the built-in method notation.

Basic string methods

- `str.len()`
- `str.putc()`
- `str.getc()`
- `str.tolower()`
- `str.compare(s)`
- `str.icompare(s)`
- `str.substr (i, j)`

Increment and Decrement Assignment Operators

IEEE Standard 1800-2012

Increment and Decrement Assignment Operators

1/1

```
// Post-Increment: i is assigned to data and then incremented
data = i++;

// Pre-Increment: i is first incremented and then its new value is assigned to data
data = ++i;

// Post-Decrement: i is assigned to data and then decremented
data = i--;

// Pre-Decrement: i is first decremented and then its new value is assigned to data
data = --i;
```

[SystemVerilog] Source Code Excerpt

Note

- SystemVerilog includes the C increment and decrement assignment operators.
- These increment and decrement assignment operators behave as **blocking assignments**.

Compound Assignment Operators

IEEE Standard 1800-2012

Compound Assignment Operators

1/1

```
// %= operator
data %= range;         // this is equivalent to: data = data % range

// <<= operator
data <<= scale;        // this is equivalent to: data = data << scale

// >>= operator
data >>= scale;        // this is equivalent to: data = data >> scale

// &= operator
data &= mask;          // this is equivalent to data = data & mask

// ^= operator
data ^= mask;          // this is equivalent to data = data ^ mask

// |= operator
data |= mask;          // this is equivalent to data = data | mask
```

[SystemVerilog] Source Code Excerpt

Compound Assignment Operators

1/1

```
// += operator
data += adc_sample;    // this is equivalent to: data = data + adc_sample

// -= operator
data -= adc_sample;    // this is equivalent to: data = data – adc_sample

// *= operator
data *= scale;         // this is equivalent to: data = data * scale

// /= operator
data /= scale;         // this is equivalent to data = data / scale
```

[SystemVerilog] Source Code Excerpt

Note

- SystemVerilog includes the C compound assignment operators.

Compound Assignment Operators

1/1

```
always_comb begin
    q_alu = op_a;
    unique case(op_code)
        ADD:    q_alu  +=  op_a;
        SUB:    q_alu  -=  op_a;
        MUL:    q_alu  *=  op_a;
        DIV:    q_alu  /=  op_a;
        MOD:    q_alu  %=  op_a;
        B_AND:  q_alu  &=  op_a;
        B_OR:   q_alu  |=  op_a;
        B_XOR:  q_alu  ^=  op_a;
        SL:     q_alu <<=  op_a;
        SR:     q_alu >>=  op_a;
        ASL:    q_alu <<<= op_a;
        ASR:    q_alu >>>= op_a;
    endcase
 end
```

[SystemVerilog] Source Code Excerpt

Note

- A dedicated use case – ALU datapath
- All assignment operator is semantically equivalent to a blocking assignment.

- Something in between:

  Arithmetic right shift (>>>) - shift right specified number of bits, fill with value of sign bit if expression is signed, otherwise fill with zero,

  Arithmetic left shift (<<<) - shift left specified number of bits, fill with zero.

  Whereas **logical shift** (<<, >>) always fill the vacated bit positions with zeroes.

Numeric Constants

IEEE Standard 1800-2012

Numeric Constants

Numbers	Bits	Base	Value	Stored
`3’b101`	3	2	5	101
`’b11 **`	?	2	3	00 … 011
`8’b11`	8	2	3	0000011
`8’b1010_1011`	3	2	171	10101011
`6’o42`	6	8	34	100010
`3’d6`	3	10	6	110
`8’hAB`	8	16	171	10101011
`42`	?	10	42	00 … 0101010

Table: Numeric constants in SystemVerilog

** Automatic scaling. Often used in parametrized modules.

Note

General form: `<size> <signed> <radix> value` (”<>” indicates optional part)

User-Defined Data Types

IEEE Standard 1800-2012

User-Defined Data Types

1/1

```
// type definitions
typedef unsigned int uint32;
typedef logic [15:0] data_bus;
...
```

[SystemVerilog] Source Code Excerpt

Note

- Typedef allows users to create their own names for type definitions that they will use frequently in their code.
- It can be applied for a single or multiple modules:
  - **Local declaration**: If typedef is used inside of a module, then it is only visible within that module
  - **External declaration**: If typedef is used outside of any module, then it can be visible within several modules.

Structures

1/1

```
struct {
   logic [7:0] data_word;
   logic even_parity_bit;
} data_struct;

data_struct data_struct_new;

assign data_struct_new.data_word        =  data[7:0];
assign data_struct_new.even_parity_bit  = ^data[7:0];
```

[SystemVerilog] Source Code Excerpt

Note

- A structure is a user-defined collection of various datatypes. Each of the constituent members of a structure is also called its field. These fields can be either standard datatypes or, they can be user-defined types (using typedef) and, possibly, another structure.
- Structures are used to group inter-related data.
- We use the 'dot notation' to access a particular element of a structure.

Structures and User-Defined Data Types

1/1

```
typedef struct {
   logic [7:0] data_word;
   logic even_parity_bit;
} data_struct;

data_struct my_data ={8’b00000000,1’b0};

assign my_data.data_word   =  data[7:0];
assign my_.even_parity_bit = ^data[7:0];
```

[SystemVerilog] Source Code Excerpt

Note

- We can also combine struct and typedef to create new data types

Initializing Structures

1/1

```
typedef struct {
   logic a, b, c;
   int   d, e, f;
} my_struct;

my_struct new = {1’b1, 1’b1, 1’b0, 1, 2, 3};
```

[SystemVerilog] Source Code Excerpt

Note

- Initial values can be specified within curly braces by listing them in the order that they are declared within the structure

Structures Passed Through Module Ports

1/1

```
typedef struct {
   logic [7:0] data_word;
   logic even_parity_bit;
} data_struct;


module data_proc (   input logic clk, reset_n,
                     input data_struct,
                     ...
```

[SystemVerilog] Source Code Excerpt

Note

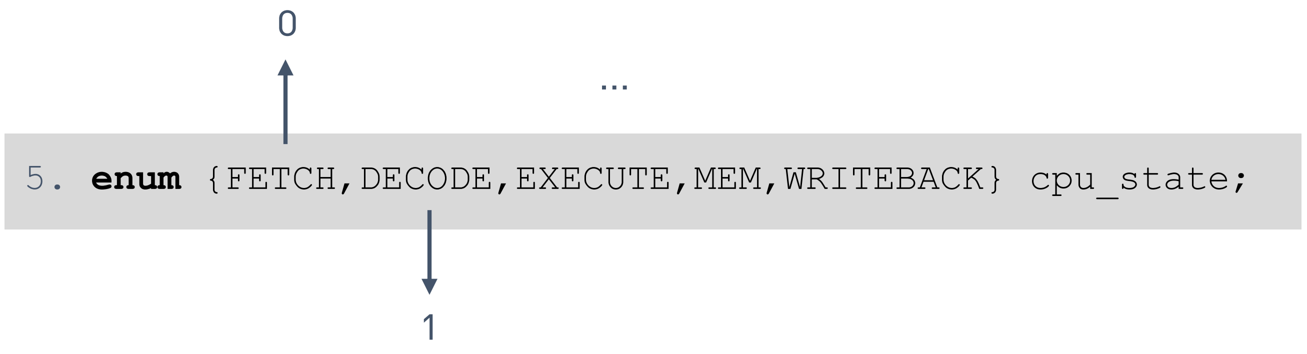
- In order to be able to pass structures through the ports of a module, we must place the typedef struct statement outside of the module.
- Then, you can pass variables of that type into and out of the module

Enumerations

1/1

```
// enumerated data types
enum {FETCH,DECODE,EXECUTE,WRITEBACK} proc_state;
enum bit {TRUE, FALSE} return_state;
...
```

[SystemVerilog] Source Code Excerpt



```
enum {FETCH,DECODE,EXECUTE,MEM,WRITEBACK} cpu_state;
```

```
enum logic [1:0] {FETCH,DECODE, EXECUTE};
```

Note

- SystemVerilog also introduces enumerated types.
- Enumerations allow you to define a data type whose values have names and thus improves the readability of the code.
- Such data types are appropriate and useful for representing state values, opcodes and other such non-numeric or symbolic data.

- Note that if the type is not defined in ‘enum’ then the type will be set as ‘integer’ (32-bit int, 2-state values) by default, which may use extra memory unnecessarily.
- The first name listed gets the value 0.
- The second name listed gets the value 1, ...

Enumerations used with User-Defined Data Types

1/1

```
typedef enum {add, sub, mult, div } instruction;
instruction c;
 ...
```

[SystemVerilog] Source Code Excerpt

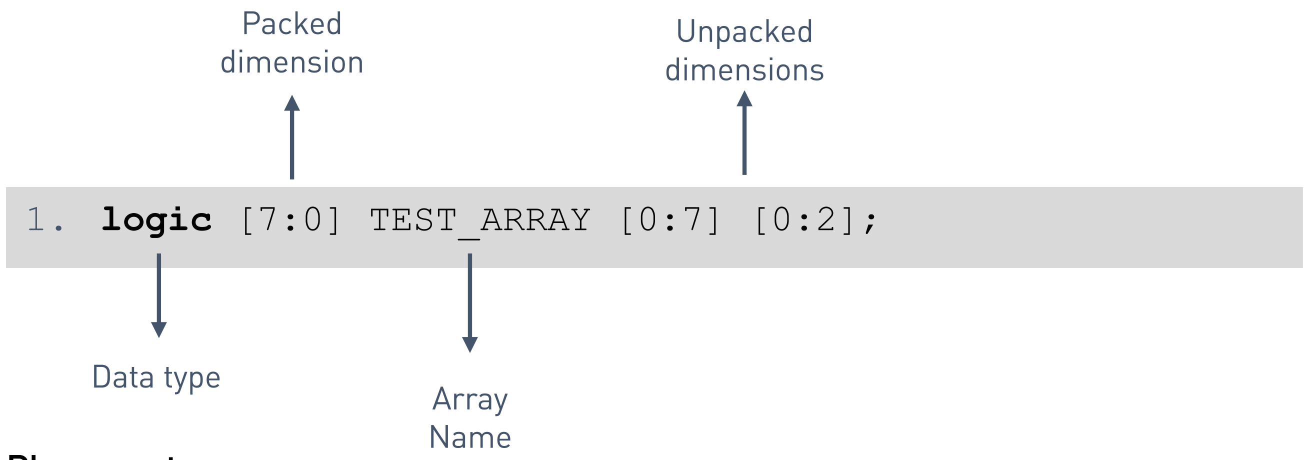
Note

- Enumerations are often used in conjunction with typedef.

Arrays in SystemVerilog

IEEE Standard 1800-2012

Arrays

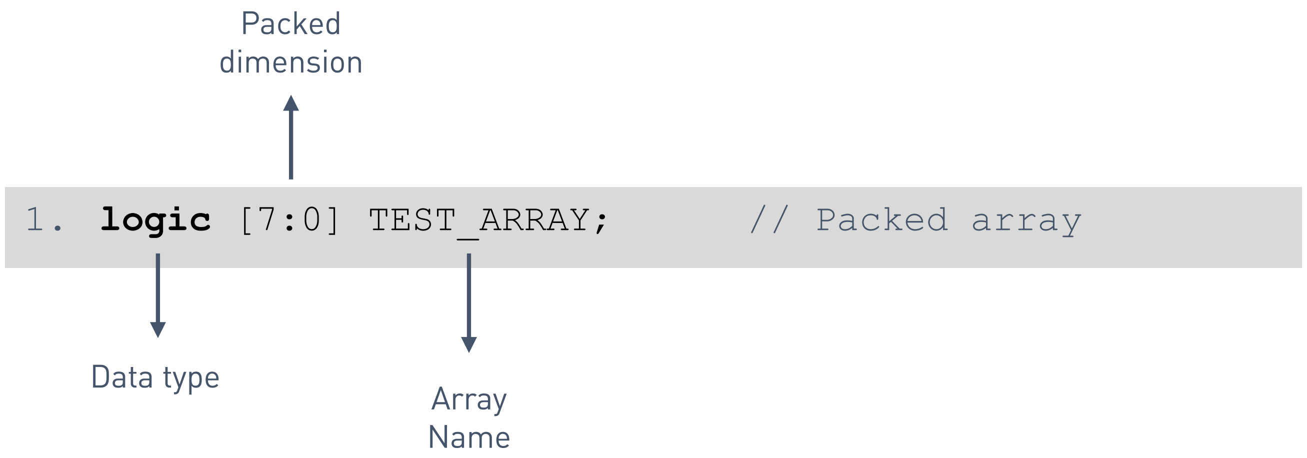


```
logic [7:0] TEST_ARRAY [0:7] [0:2];
```

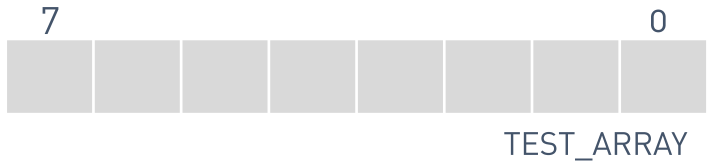
Please note

- The term packed array is used to refer to the dimensions declared before the data identifier name.
- The term unpacked array is used to refer to the dimensions declared after the data identifier name.
- A packed array is guaranteed to be represented as a contiguous set of bits.
- An unpacked array may or may not be so represented as a contiguous set of bits.

Arrays



```
logic [7:0] TEST_ARRAY;      // Packed array
```



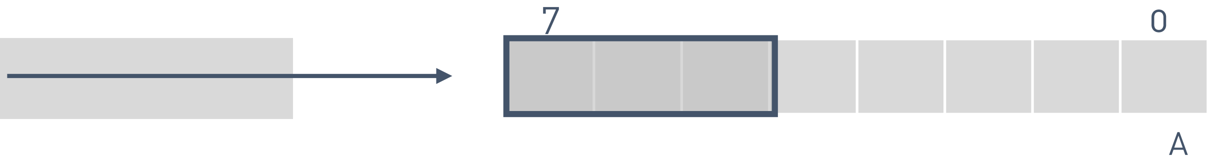
Note

- **One dimensional packed array is also denoted as a vector**
- Packed arrays are guaranteed to be laid out contiguously in memory.
- Packed arrays are restricted to the "bit" types (bit, logic, int, ...)
- Write access:

```
assign TEST_ARRAY = 12;
```

Arrays

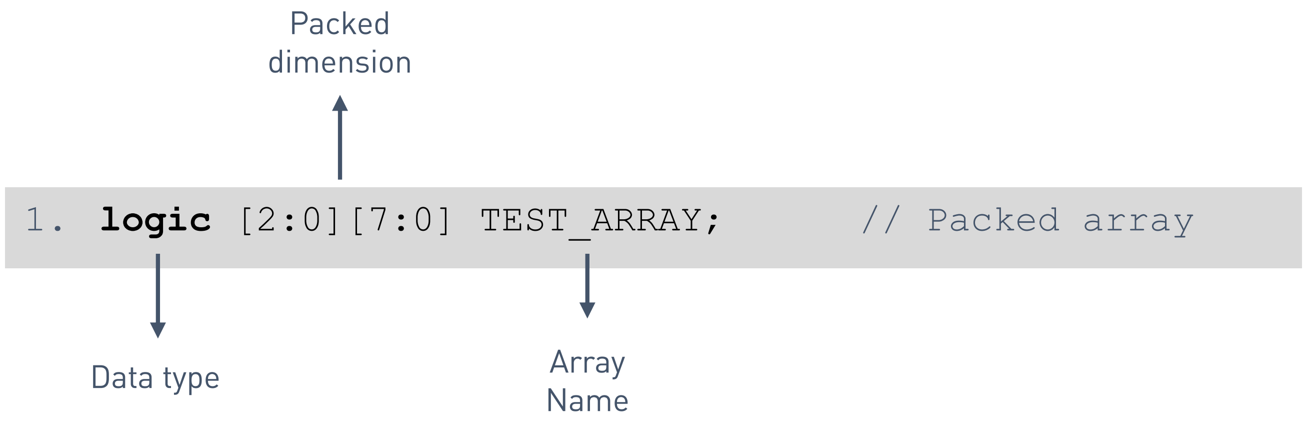
```
logic [7:0] A; assign A[7:5] = 3’b101;
```



Note

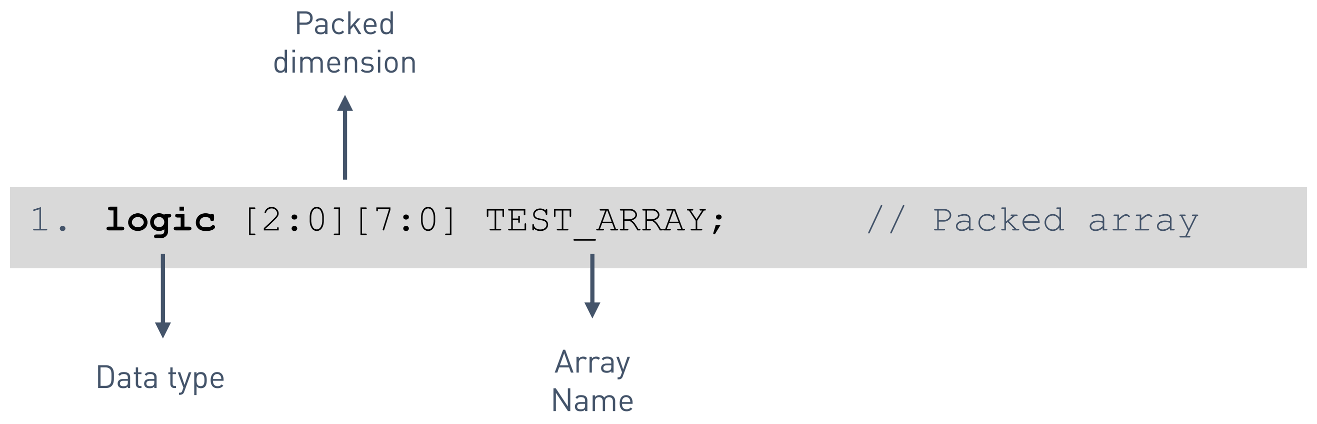
- Access parts of a vector …

Arrays

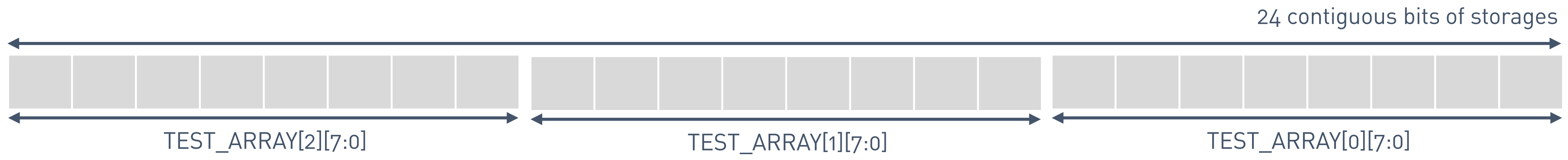


```
logic [2:0][7:0] TEST_ARRAY;      // Packed array
```

Arrays



```
logic [2:0][7:0] TEST_ARRAY;      // Packed array
```



Note

- Two-dimensional packed array.
- **This represents still a set of contiguous bits in memory, however segmented into smaller groups.**
- A packed array is a mechanism for subdividing a vector (24-bit in the upper example) into sub-fields (8-bit in the upper example), which can be conveniently accessed as array elements.

Arrays

```
logic [2:0][7:0] TEST_ARRAY = 24’d0;
```

Note

- When declaring a packed array, you can optionally include an initial value as part of the declaration.
- Alternatively, it is possible to set the initial values as follows:

```
logic [2:0][7:0] TEST_ARRAY = {8’d1, 8’d2, 8’d3};
```

- Using the replication operator:

```
logic [2:0][7:0] TEST_ARRAY = {3{8’hAF}};
```

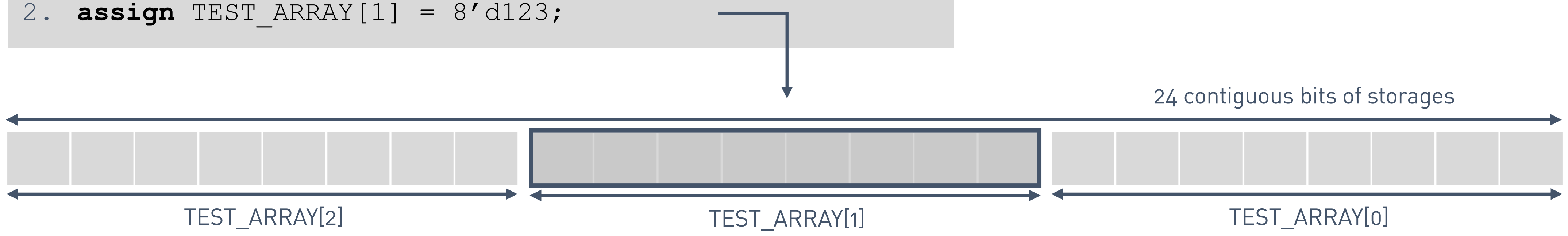
Arrays

```
logic [2:0][7:0] TEST_ARRAY;      // Packed array
```

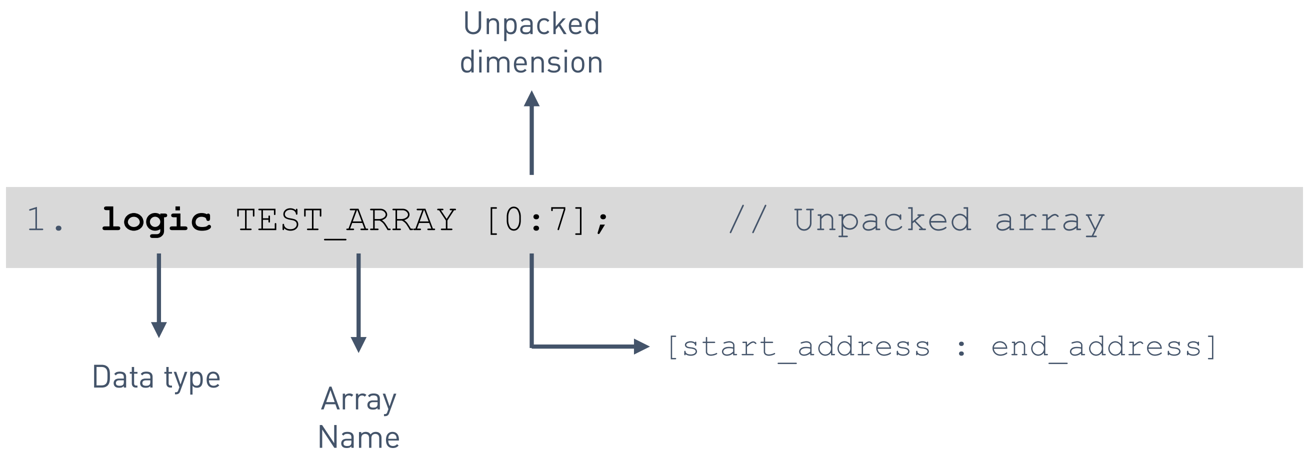
Note

- Multi-dimensional packed array.
- This represents still a set of contiguous bits in memory, however it is also segmented into smaller groups.

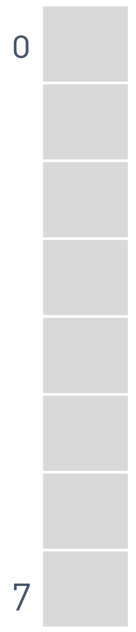
```
assign TEST_ARRAY[1] = 8'd123;
```



Arrays



```
logic TEST_ARRAY [0:7];      // Unpacked array
```



Note

- Single dimension unpacked array.
- Alternative way of expressing the unpacked dimension.

```
logic TEST_ARRAY [8];      // Unpacked array
```

Arrays

```
1.  assign TEST_ARRAY = 8'd123;     // Unpacked array
```

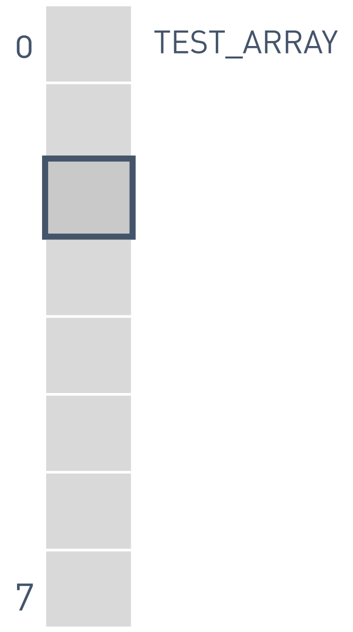
Note

- **The previous expression leads to a SystemVerilog error:**

  Error (10928): SystemVerilog error at test_design.sv(19): packed array type cannot be assigned to unpacked array type - types do not match

- The following code examples accesses the fourth location of the TEST_ARRAY:

```
2.  assign TEST_ARRAY[2] = 1'b1;
```



Arrays

Packed dimension

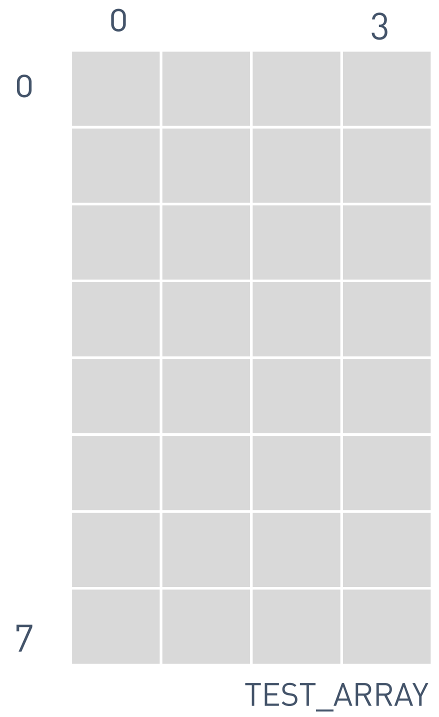
```
logic TEST_ARRAY [0:7][0:3];      // Unpacked array
```

Data type

Array Name

Note

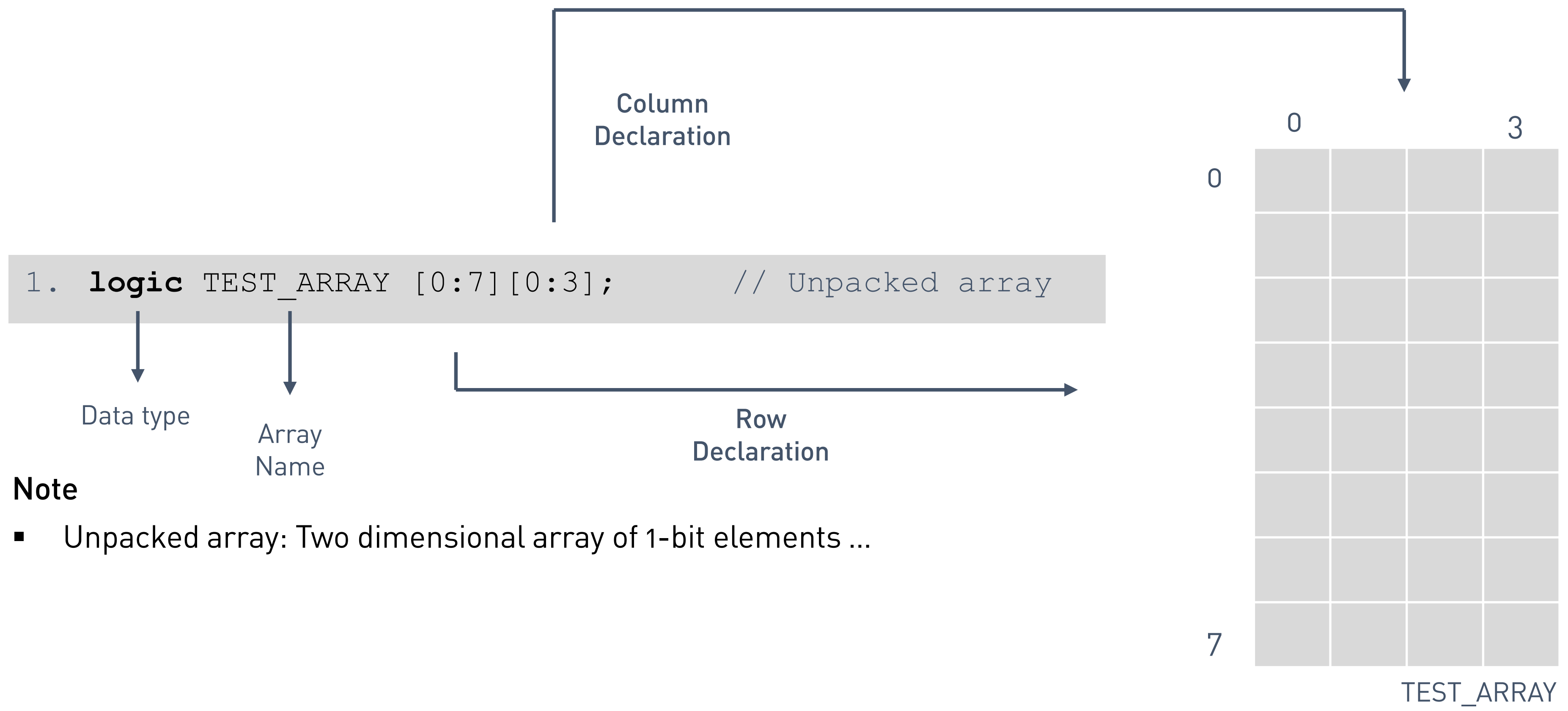
- Unpacked array: Two dimensional array of 1-bit elements …



Introduction to SystemVerilog

h_da – fbeit - FPGA-based SoC Design

Arrays



```
logic TEST_ARRAY [0:7][0:3];      // Unpacked array
```

Note

- Unpacked array: Two dimensional array of 1-bit elements ...

Arrays

Packed dimension

```
1.  logic TEST_ARRAY [0:7][0:3];       // Unpacked array
```

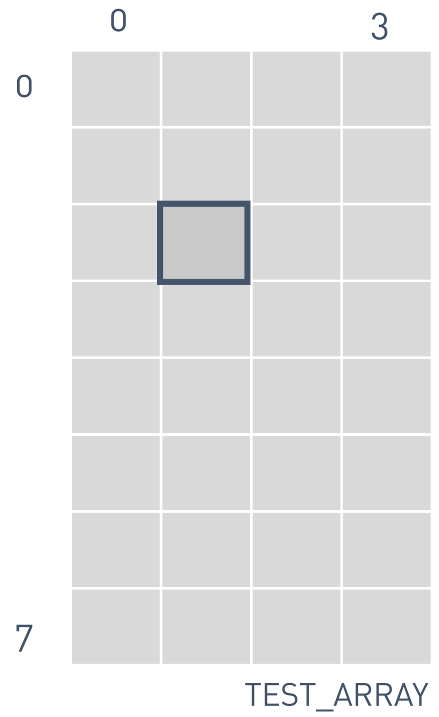
Data type

Array Name

Note

- Unpacked array: Two dimensional array of 1-bit elements ...
- Write access:

```
2.  assign TEST_ARRAY[2][1] = 1'b1;
```



Arrays

- It is also possible to have **mixed arrays.** This means that it has at least one packed dimension and one unpacked dimension. Let's start with the following array:

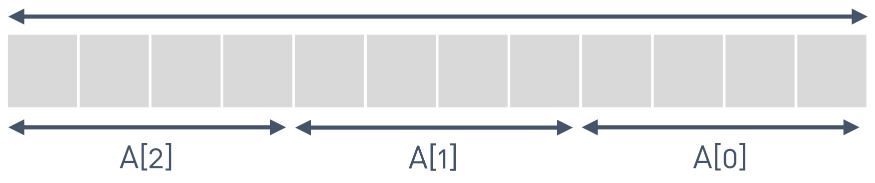
```
logic [2:0][3:0] A;  // packed array
```

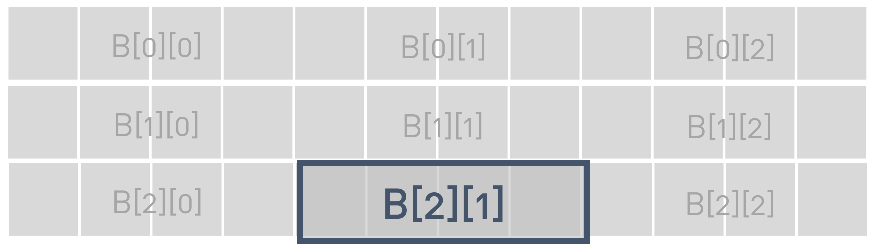
- Let's assume that you would like to have three groups of the previous array.
- Just add an unpacked dimension …

```
logic [2:0][3:0] B [0:2];
```

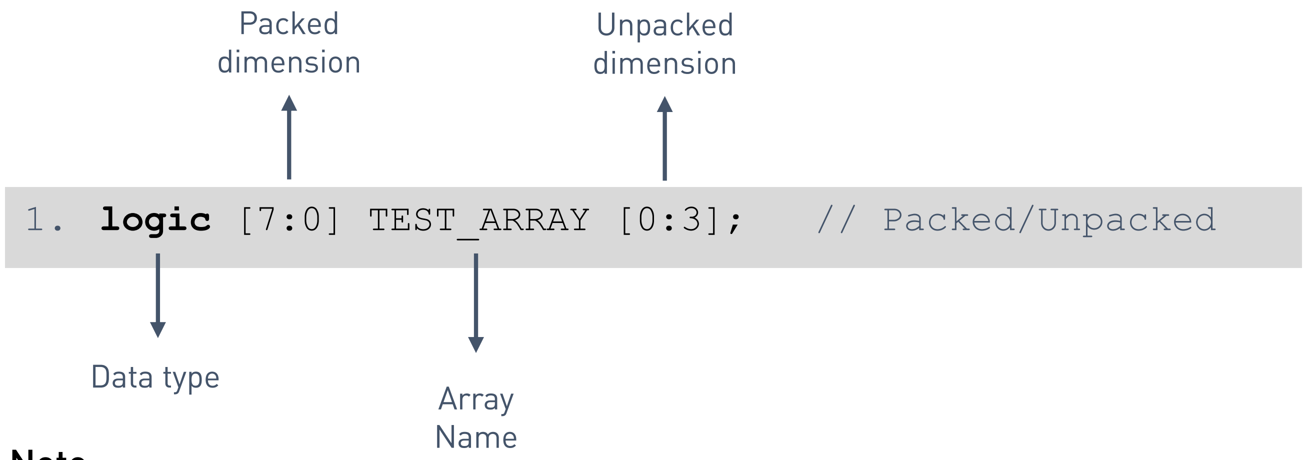
- Accessing a single element:

```
assign B[2][1] = 4'b1010;
```

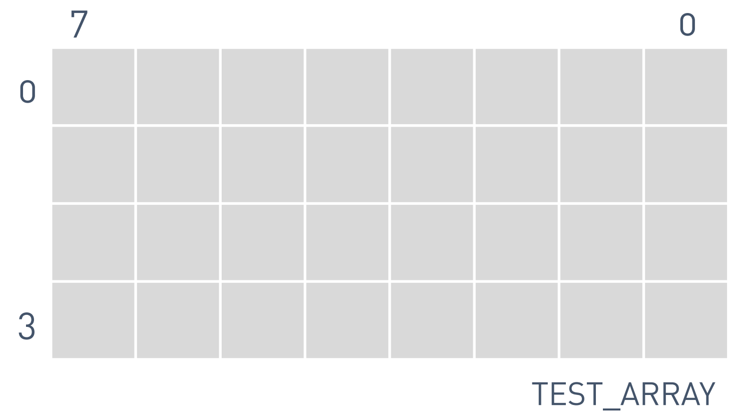




Arrays



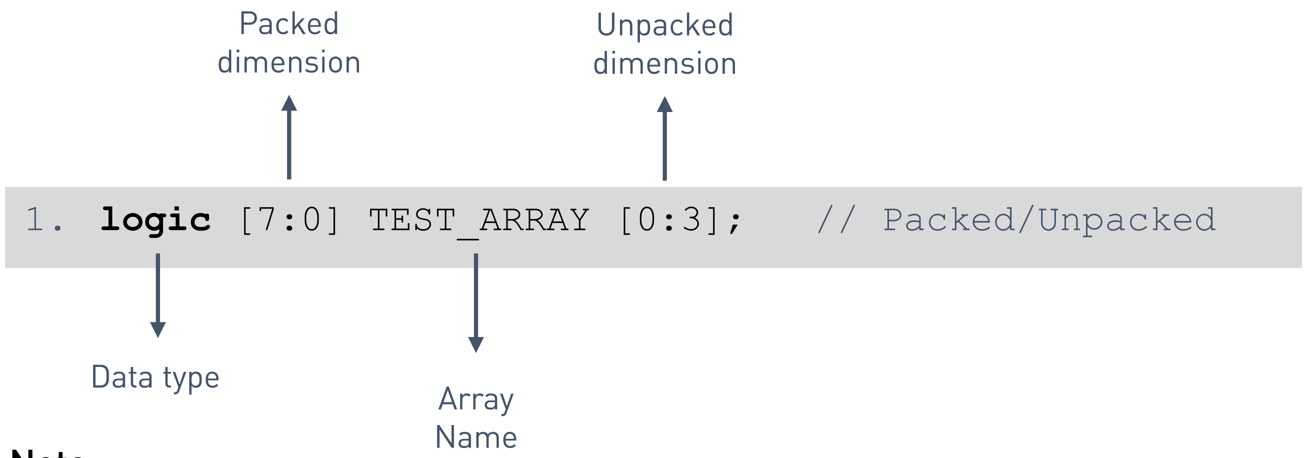
```
logic [7:0] TEST_ARRAY [0:3];   // Packed/Unpacked
```



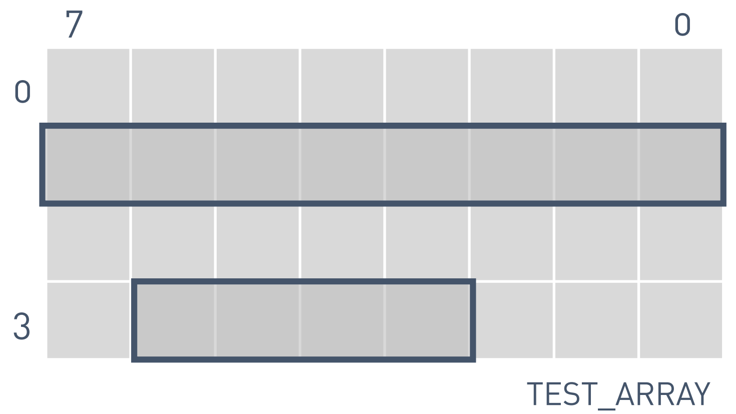
Note

- It is also possible to have mixed arrays. This means that it has at least one packed dimension and one unpacked dimension.

Arrays



```
logic [7:0] TEST_ARRAY [0:3];    // Packed/Unpacked
```



Note

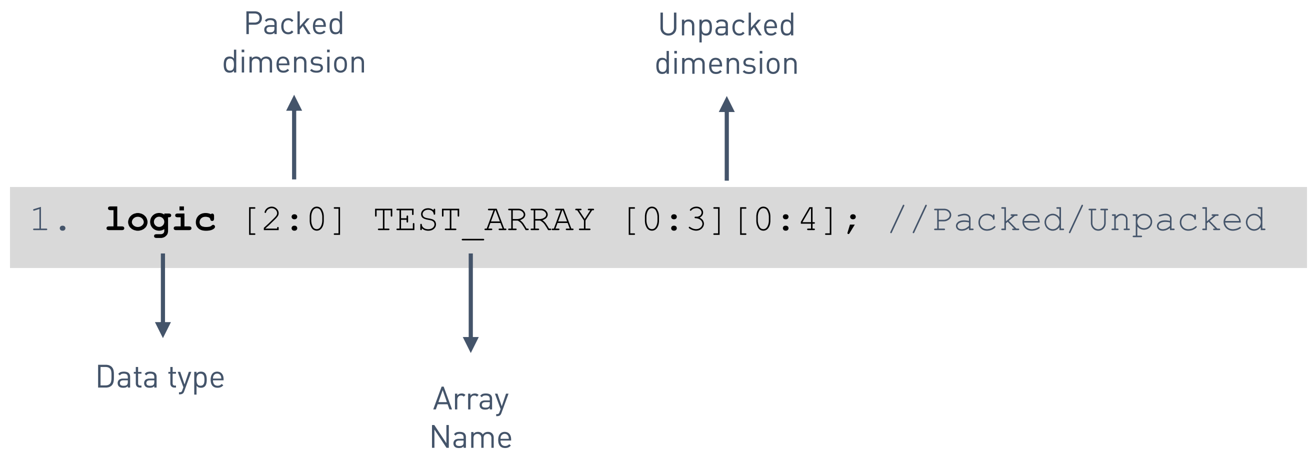
- Write access #1:

```
assign TEST_ARRAY[1] = 8'b10101010;
```

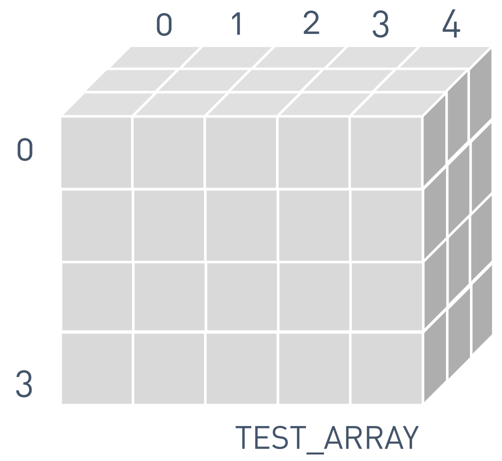
- Write access #2:

```
assign TEST_ARRAY[3][6:3] = 4'b1010;
```

Arrays



```
logic [2:0] TEST_ARRAY [0:3][0:4]; //Packed/Unpacked
```



Arrays

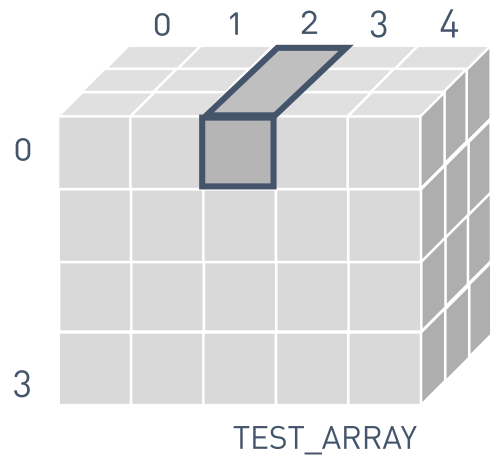
Packed dimension

Unpacked dimension

```
logic [2:0] TEST_ARRAY [0:3][0:4]; //Packed/Unpacked
```

Data type

Array Name



Note

- Write access:

```
assign TEST_ARRAY[0][2] = 3’b101;
```

Array Reduction Methods

1/1

```
// data array ...
byte data_set[2:0] = '{ 1, 3, 5 };
int result;

// applying array reduction methods ...
assign result = data_set.sum;
assign result = data_set.product;
assign result = data_set.and;     // bit-wise AND
assign result = data_set.or;      // bit-wise OR
assign result = data_set.xor;     // bit-wise XOR
```

[SystemVerilog] Source Code Excerpt

Note

- SystemVerilog Array Reduction methods operate on an **unpacked array** to reduce the array to a single value.

Some more examples

1/1

```
// 128-element array of 32-bit wide elements
logic [31:0] A [127:0];
// 2 arrays of 16-bit wide wires indexed from 7 to 0
logic [15:0] B[7:0], C[7:0];
// 256-entry memory mem_test of 8-bit registers
logic [7:0] mem_test [0:255];
// two-dimensional array of one bit registers
logic data_array [7:0][0:255];
```

[SystemVerilog] Source Code Excerpt

```
mem_test              = 0;  // Illegal!
data_array[1]         = 0;  // Illegal!
data_array[1][31:12]  = 0;  // Illegal!
mem_test[1]           = 0;  // OK
data_array[1][0]      = 0;  // OK
```

[SystemVerilog] Source Code Excerpt

Some more examples

a text file containing hex values

start and end address to specify the section to load (optional)

```
$readmemh("hex_memory_file.mem", memory_array, [start_address], [end_address])
$readmemb("bin_memory_file.mem", memory_array, [start_address], [end_address])
```

a text file containing hex values

SystemVerilog array

Note

- SystemVerilog allows you to initialize memory from a text file with either hex or binary values.
- Values might be separated by whitespace, tab or a newline

```
// Declare a ROM variable
logic [15:0] rom [0:3];

initial begin
    $readmemh("single_port_rom_init.txt", rom);
end
```

```
// this is a comment
DEAD
BEEF
DEAD
BEEF
```

[SystemVerilog] Source Code Excerpt

SystemVerilog Packages

IEEE Standard 1800-2012

Packages

Looking back …

- According to the latest Verilog standard (IEEE 1364-2005), declarations of variables, nets, tasks and functions must be declared within a module. Objects declared within a module are local to the module.
- Verilog does not provide a place to make global declarations, such as global functions. A declaration that is used in multiple design blocks must be declared in each block. This requires redundant declarations and is prone to errors

Packages

1/1

```
package global_defs;
    enum {FETCH,DECODE,EXECUTE,WRITEBACK} proc_state;
    typedef logic [15:0] data_bus;
    typedef logic [15:0] ctrl_bus;
    ...
endpackage
```

[SystemVerilog] Source Code: global_defs.sv

Note

- To enable sharing a user-defined type definition across multiple modules, SystemVerilog adds packages to the Verilog language.
- Packages enable the sharing of parameters, localparms, consts, types, structs, tasks or functions across multiple modules.

Packages

1/1

```
package global_defs;
    enum {FETCH,DECODE,EXECUTE,WRITEBACK} proc_state;
    typedef logic [15:0] data_bus;
    typedef logic [15:0] ctrl_bus;
    ...
 endpackage
```

[SystemVerilog] Source Code: global_defs.sv

Note

- To enable sharing a user-defined type definition across multiple modules, SystemVerilog adds packages to the Verilog language. Packages enable the sharing of parameters, localparams, consts, types, structs, tasks or functions across multiple modules.

1/1

```
module rt_cpu_sys (
    input global_defs :: data_bus local_databus,
    input global_defs :: ctrl_bus local_ctrlbus,
    ...
    );
```

[SystemVerilog] Source Code Excerpt: rt_cpu_sys.sv

1. **How to import Packages - #1**
   Using the double colon scope operator allows the type defined in the global_defs package to be referenced.

Packages

1/1

```
package global_defs;
    enum {FETCH,DECODE,EXECUTE,WRITEBACK} proc_state;
    typedef logic [15:0] data_bus;
    typedef logic [15:0] ctrl_bus;
    ...
 endpackage
```

[SystemVerilog] Source Code: global_defs.sv

Note

- To enable sharing a user-defined type definition across multiple modules, SystemVerilog adds packages to the Verilog language. Packages enable the sharing of parameters, localparms, consts, types, structs, tasks or functions across multiple modules.

1/1

```
import global_defs::*;
module rt_cpu_sys (
    input data_bus local_databus,
    input ctrl_bus local_ctrlbus,
    ...
```

[SystemVerilog] Source Code Excerpt: rt_cpu_sys.sv

2. How to import Packages - #2
   Placing the import instruction outside of the module is known as global import. The single types defined in the package can be referenced directly
   Package content is accessible for all modules in the design.

Packages

1/1
```
package global_defs;
    enum {FETCH,DECODE,EXECUTE,WRITEBACK} proc_state;
    typedef logic [15:0] data_bus;
    typedef logic [15:0] ctrl_bus;
    ...
 endpackage
```

[SystemVerilog] Source Code: global_defs.sv

Note

- To enable sharing a user-defined type definition across multiple modules, SystemVerilog adds packages to the Verilog language. Packages enable the sharing of parameters, localparms, consts, types, structs, tasks or functions across multiple modules.

1/1
```
module rt_cpu_sys
import global_defs::*; (
    input data_bus local_databus,
    input ctrl_bus local_ctrlbus,
    ...
```

[SystemVerilog] Source Code Excerpt: rt_cpu_sys.sv

3. **How to import Packages - #3**
   Placing the import instruction inside the module is known as local import. The single types defined in the package can be referenced directly. The package content is however only accessible from inside the module.

Type casting

```
// casting to change size to 10-bits, no truncation
// warning!
10´(x - 2)
// multiplication of two real numbers, casting to
// type int
int´(2.0 * 3.0)
// casting to signed
signed´(x)
// casting to defined enum p_state, converts
// number the correponding value in the enum p_state
fruit´(0)
```

1/1

[SystemVerilog] Source Code Excerpt

Note

- Verilog is loosely typed, so no significant type checking is done at compile time
- SystemVerilog has more complex data types so it needs to be stricter about type conversions. In conclusion, SystemVerilog requires mechanism for converting between data types: SystemVerilog provides a static cast operator, the apostrophe. Casting allows assignments of variables that may not be otherwise valid.

Operators in SystemVerilog

IEEE Standard 1800-2012

Operators in SystemVerilog

```
[]    bit-select or part select
()    parenthesis
```

[SystemVerilog] Operators: selection

```
!     logical negation
&&    logical AND
||    logical OR
```

[SystemVerilog] Operators: logical operators

```
~     bit-wise negation
&     bit-wise AND
|     bit-wise OR
^     bit-wise XOR
```

[SystemVerilog] Operators: bit-wise operators

```
{}    concatenation
```

[SystemVerilog] Operators: concatenation operator

```
{{}}  replication
```

[SystemVerilog] Operators: replication operators

```
&     reduction AND
|     reduction OR
~&    reduction NAND
~|    reduction NOR
^     reduction XOR
~^    reduction XNOR
```

[SystemVerilog] Operators: reduction operators

```
>     greater than
>=    greater than or equal to
<     less than
~|    less than or equal to
==    equal
!=    not equal
```

[SystemVerilog] Operators: relational operators

Operators in SystemVerilog contd.

```
1. ===   case equality
2. !==   case inequality
```

[SystemVerilog] Operators: case equality operators

```
1. >>    shift right (zero filling)
2. <<    shift left  (zero filling)
3. >>>   shift right (sign-extended)
4. <<<   shift left  (sign-extended)
```

[SystemVerilog] Operators: shift operators

```
1. +     unary (sign) plus
2. -     unary (sign) minus
3. *     multiply
4. /     divide
5. %     modulus
6. +     binary plus
7.  -    binary minus
```

[SystemVerilog] Operators: arithemtic operators

```
1. ?:    conditional
```

[SystemVerilog] Operators: conditional operators

Operator Precedence in SystemVerilog

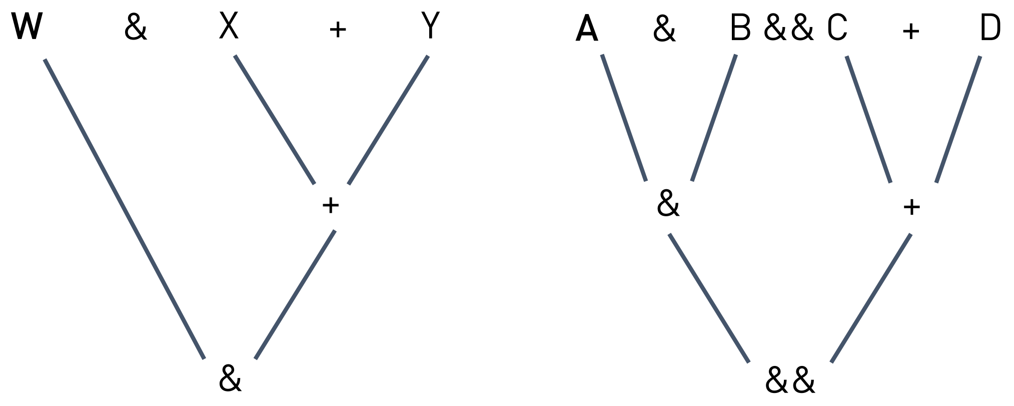
IEEE Standard 1800-2012

Operator Precedence in SystemVerilog

Highest	~	Not
	*,/,%	Mult,di
	+,-	Add, sub
	<<,>>	Logical shift
	<<<,>>>	Arithmetic shift
	<,<=,>, >=	Comparison
	==, !=	Equal, not equal
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	|, ~|	OR, XOR
Lowest	? :	Ternary operator

Note

- Pretty similar to other programming languages ...
- Examples:



- Operator Precedence Table below shows the precedence of operators from highest to lowest. Operators on the same level evaluate from left to right.
- It is strongly recommended to use parentheses to define order of precedence and improve the readability of the code.

Block Names in SystemVerilog

1/1

```
always_comb begin : write_registers
    if(cs && addr[7:6] == 2'b00) begin ctrl_reg
         go       <= data_in[15];
         ctrl_sel <= data_in[10:8];
         ctrl_val <= data_in[7:0];
         ...
         end : ctrl_reg;
    else if(cs && addr[7:6] == 2'b01) begin : ram_reg
         even   <= data_in[15];
         enable <= data_in[14];
         wait   <= data_in[10:8];
         end : ram_reg
    else if(cs && addr[7:6] == 2'b10)
         ram_data <= data_in
 end : write_registers
```

[SystemVerilog] Source Code Excerpt: rt_cpu_sys.sv

Tristate Buffers and Bidirectional Ports in SystemVerilog

IEEE Standard 1800-2012

Tristate Buffers

1/1

```
module tristate (
   input logic d, input logic enable,
   output tri q
   );

   assign q = enable ? d : 1’bz;

endmodule
```

[SystemVerilog] Source Code: tristate.sv

Note

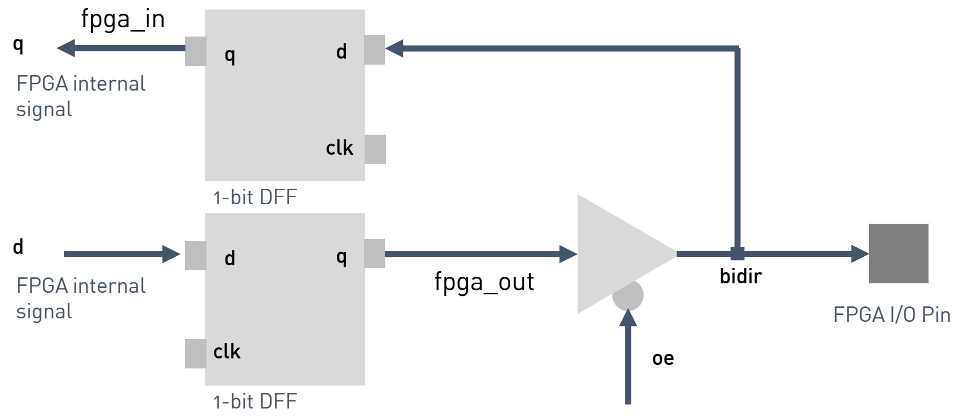
- Tristate buffers are switches having three logic states 1, 0 and z (or commonly known as high impedance state).
- Can be mapped to FPGA I/O cells only. Thus tri-state ports can only be used in the top-level design.
- The ability to tri-state is particularly useful when multiple buffer circuits are coupled to the same load since this permits the buffer circuits that are not active in driving the bus to be decoupled therefrom in order to avoid signal contention on the bus.

Bidirectional ports

1/1

```
module bir_dir_interface(
     input logic d, oe, clk,
     inout bidir, output logic q
 );

 logic fpga_in, fpga_out;

 assign bidir = (oe == 1’b1) ? fpga_out : 1’bz;

 assign q = fpga_out;

 always_ff @(posedge clk) begin
    fpga_in  <= bidir;
    fpga_out <= d;
    end

 endmodule
```

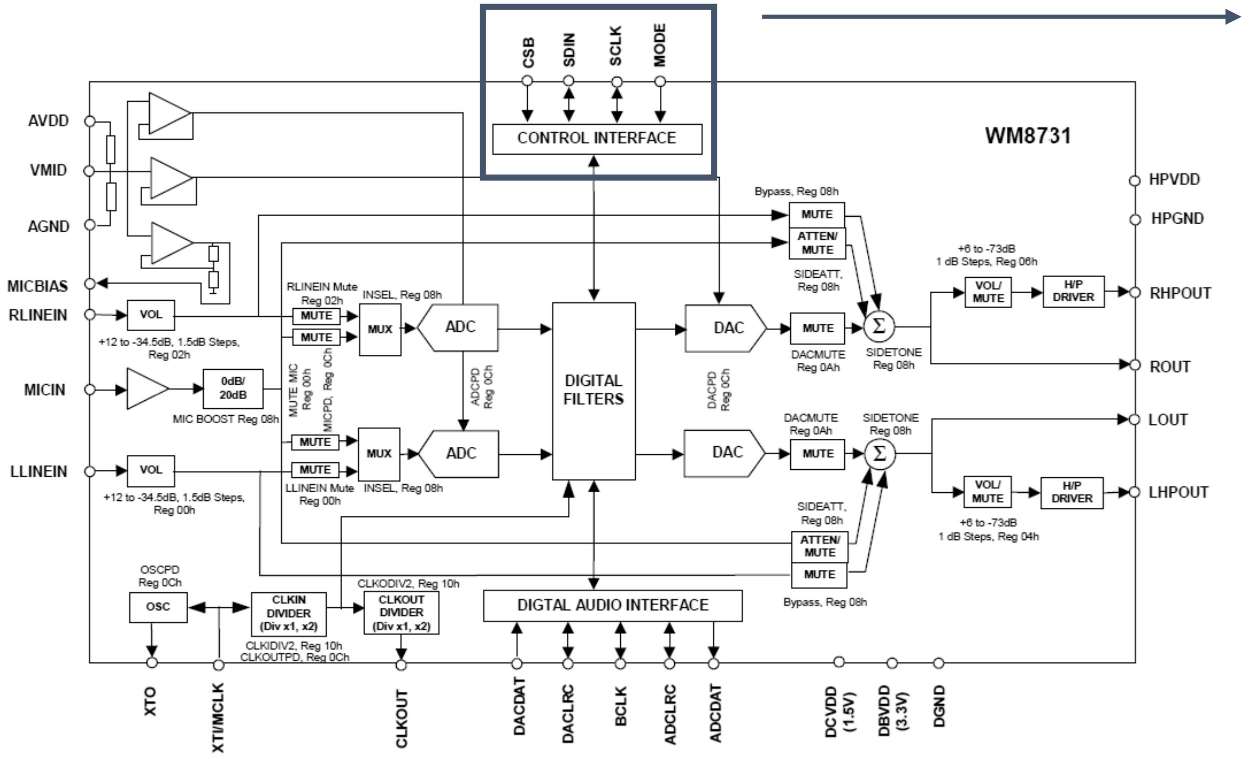
[SystemVerilog] Source Code: bidir_interface.sv



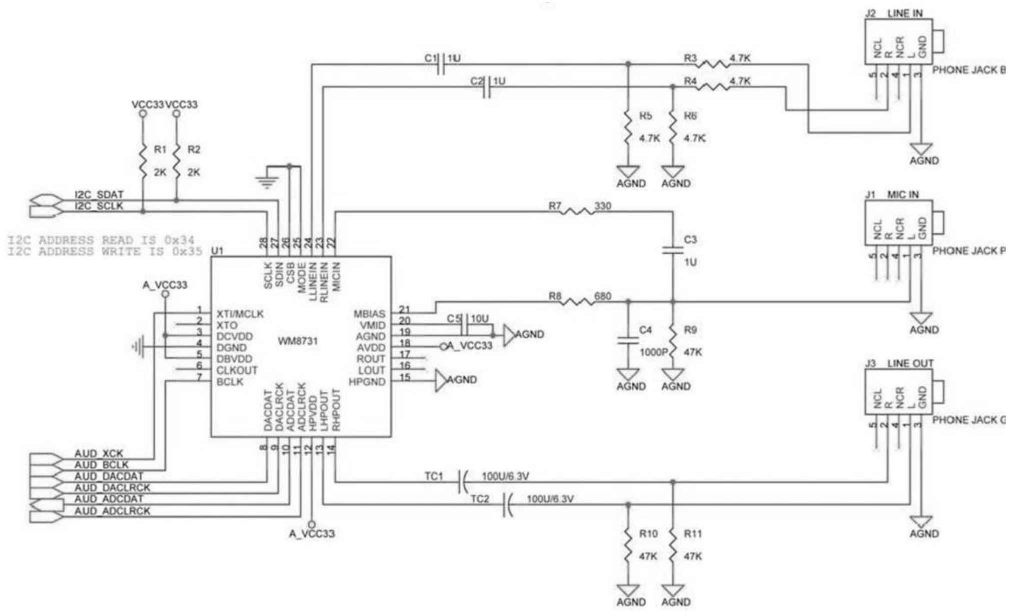
Note

- This example implements a clocked bidirectional pin in SystemVerilog HDL.
- The value of oe determines whether bidir is an input, feeding in fpga_in , or a tri-state, driving out the value fpga_out .

Bidirectional ports



- Sample rate adjustable from 8 kHz to 96 kHz
- Bit Format
- Loudness
- …



Functions

IEEE Standard 1800-2012

Functions

```
function <range_or_type> <function_name>;
   <input declaration(s)>
   <variable declaration(s)>
   begin
      // function body ...
      <statements>
   end
endfunction
```

1/1

[SystemVerilog] Source Code Excerpt

Return value
As stated below, there is an implicit variable is declared inside the function with the same name. The width and type of this implicit variable is declared by `<range_or_type>`.

Function call
Calls a function, which returns a value for use in an expression

```
var = <function_name> (Argument(s));
```

[SystemVerilog] Source Code Excerpt

Note

- Identical code sections are often used in different places in a design. **They are used to improve the readability and to exploit re-usability code**
- Functions are used to group statements to again define a new overall mathematical or logical function.
- **Functions may not contain timing controls such as delays or event control statements.**
- A function returns a single value.
- A function my return a value by assigning the function name, as if it were a variable, or by using the return keyword.

Functions contd.

type and range of the implicitly generated return value

1/1

```
function logic [7:0] reverse_bits;
   input logic [7:0] byte;
   integer i;
   begin
      for(i = 0; i < 8; i = i + 1)
         reverse_bits[7-i] = byte[i];
   end
endfunction
```

input argument

local variable

the begin/end statement is actually just necessary in case of grouping multiple statements

[SystemVerilog] Source Code Excerpt

Note

- As illustrated above, functions may implement (only) combinatorial behaviour.
- They compute a value on the basis of the present value of the input arguments and return a single value.

Functions contd.

Input argument

1/1

```
function logic [7:0] shift_left(logic [7:0] data);
   begin
          shift_left = {data[6:0],1'b0};
   end
endfunction
```

[SystemVerilog] Source Code Excerpt

Note

- Input arguments can be declared in this form as well.

Functions – Gotcha!

```
function signed logic [7:0] signed_op (logic [7:0] data);
   begin
          ...
   end
endfunction
```

1/1

[SystemVerilog] Source Code Excerpt

Note

- Be careful when using signed data types!
- The upper declaration states that the return value is a signed type.
- This is different to a variable declaration, where the signed keyword appears after the type:

```
logic signed [7:0] my_variable;
```

[SystemVerilog] Source Code Excerpt

Functions contd.

1/1

```systemverilog
module function_example_1(
    input logic [7:0] d,
    output logic [7:0] q
 );

 function logic [7:0] reverse_bits;
    input logic [7:0] byte;
    integer i;
    for(i = 0; i < 8; i = i + 1)
        reverse_bits[7-i] = byte[i];

 endfunction

 always_comb
    q = reverse_bits(d);

endmodule
```

[SystemVerilog] Source Code: function_example_1.sv

Functions are declared within a parent module ...

and can be called from an always blocks

Alternatively:
Call a function from a continuous assignment or another function

```systemverilog
assign q = reverse_bits(d);
```

[SystemVerilog] Source Code Excerpt

Functions

1/1

```
function <range_or_type> <function_name>;
   <input declaration(s)>
   <variable declaration(s)>
   begin
      // function body ...
      <statements>
   end
endfunction
```

[SystemVerilog] Source Code Excerpt

Note

- Functions may be used if there are no delay, timing, or event control constructs
- Functions may be used if only a single value needs to be returned.
- Functions may be used if there is at least one input argument
- Functions may be used if there are no output or inout arguments
- Functions may be used if there are no non-blocking assignments

Tasks

IEEE Standard 1800-2012

Tasks

1/1

```
task <task_name>;
   <i/o declarations>
   <variable and event declarations>
   begin
      // task body ...
      <statements>
   end
endfunction
```

[SystemVerilog] Source Code Excerpt

Note

- **A task is similar to a function, but unlike a function it has both input and output ports. Therefore tasks do not return values.**
- Tasks can be used for modelling both combinational and sequential logic.
- A task can contain timing controls. In addition to that it can call other tasks and functions.
- A task can have zero, one, or more arguments. Values are passed to and from a task through arguments. The arguments can be input, output, or inout.

Tasks

1/1

```systemverilog
task reverse_bits;
   input logic [7:0] byte;
   output logic [7:0] q;
   integer i;
   begin
       for(i = 0; i < 8; i = i + 1)
           q[7-i] = byte[i];
   end
endtask
```

[SystemVerilog] Source Code Excerpt

Note

- As it is the case with functions, tasks are declared within a parent module as well ...

Tasks

1/1

```
task SEND_ACKNOWLEDGE;          // semicolon needed
   begin                        // begin-end necessary
      // Send an acknowledge
      tb_ack = 1;
      @(posedge tb_local_clock);
      # 1;
      tb_ack = 0;
      @(posedge tb_local_clock);
      # 1;
   end                          // begin-end necessary
 endtask                        // no semicolon
```

Here, the tasks performs a write access to a global variable tb_ack

[SystemVerilog] Source Code Excerpt

Note

- Tasks are often used for testbench scheduling purposes.
- In addition to local variables, tasks can access (reading/writing) global variables as well (defined in the module that is using this task)..