CND212: Digital Testing and Verification

SystemVerilog Arrays

SystemVerilog Enhancement - Arrays

- SystemVerilog enhances the use of arrays to include the following:
  - Compact Declaration
  - Packed Arrays
  - Dynamic Arrays
  - Associative Arrays
- SystemVerilog compact array declaration is similar to C's style

```
int lo_hi[0:15];        // 16 ints [0]..[15]
int c_style[16];        // 16 ints [0]..[15]
```

Fixed Arrays – Declaration & Initialization

- The Array size is set during compile time
- Declaring and initializing fixed arrays

```
initial begin
  static int ascend[4] = '{0,1,2,3}; // Initialize 4 elements
  int descend[5];

  descend = '{4,3,2,1,0};           // Set 5 elements
  descend[0:2] = '{7,6,5};          // Set just first 3 elements
  ascend = '{4{8}};                 // Four values of 8
  ascend = '{default:42};           // All elements are set to 42
end
```

- Declaring muti-dimensional fixed arrays

```
int array2 [0:7][0:3];  // Verbose declaration
int array3 [8][4];      // Compact declaration
array2[7][3] = 1;       // Set last array element
```

Fixed Arrays Operations

- Using *for* and *foreach*

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

Fixed Arrays Operations

1-D Array

0	1	2	3	4	5	← Indices
						← Elements

2-D Array

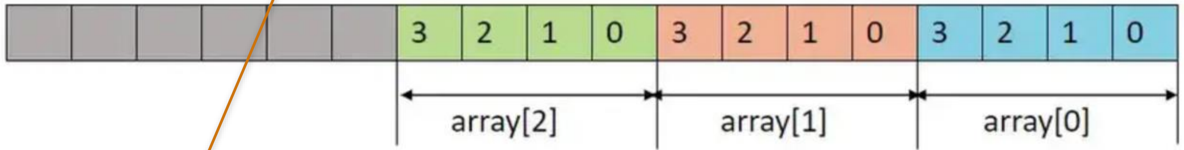
	0	1
0	1	100
1	2	200
2	3	300
3	4	400
4	5	500
5	6	600

```
module foreach_example;
  int array[5] = '{100, 200, 300, 400, 500};
  initial begin
    foreach (array[i]) begin
      $display("array[%0d] = %0d", i, array[i]);
    end
  end
endmodule
```

```
module foreach_example;
  int array[6][2] = '{'{1, 100}, '{2, 200}, '{3, 300}, '{4,400}, '{5, 500}, '{6, 600}};
  initial begin
    foreach (array[i,j]) begin
      $display("array[%0d][%0d] = %0d", i,j, array[i][j]);
    end
  end
endmodule
```

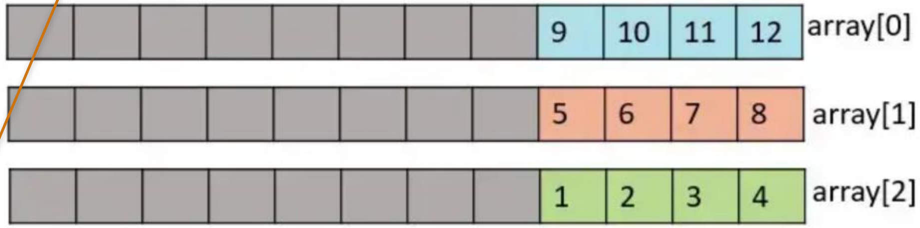
Packed vs. Unpacked Arrays

<u>Packed:</u>
Dimension is written before the variable name




```
module packed_array_example;
  bit [2:0][3:0] array = '{4'h2, 4'h4, 4'h6};
  initial begin
    foreach (array[i]) begin
      $display("array[%0h] = %0h", i, array[i]);
    end
  end
endmodule
```

<u>Unpacked:</u>
Dimension is written after the variable name

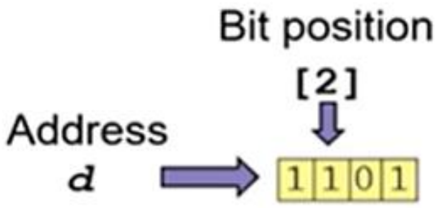



```
module unpacked_array_example;
  int array [2:0][3:0] = '{'{1, 2, 3, 4},
                          '{5, 6, 7, 8},
                          '{9, 10, 11, 12}
                         };
  initial begin
    foreach (array[i,j]) begin
      $display("array[%0d][%0d] = %0d", i, j, array[i][j]);
    end
  end
endmodule
```

Packed vs. Unpacked Arrays

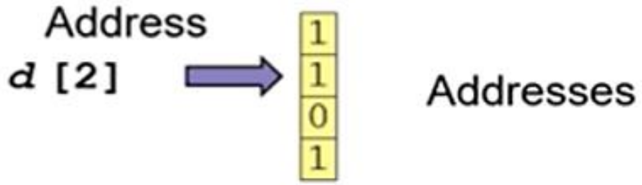
Packed:

```
logic [3:0] d;
```



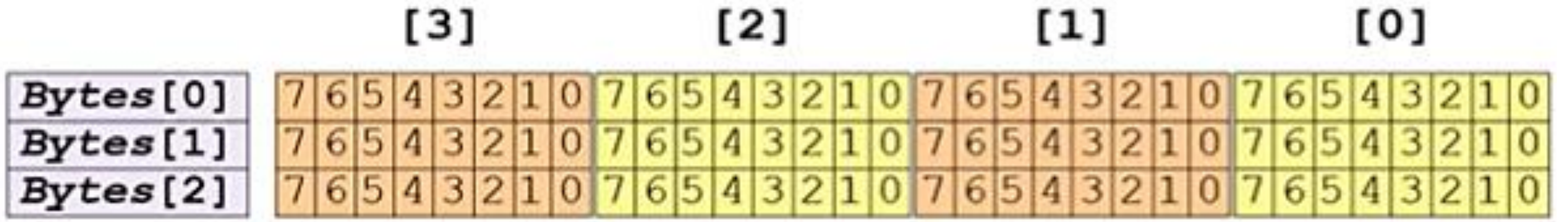
Unpacked:

```
logic d [4];
```



Array Example-1

```
logic [3:0][7:0] Bytes [3];    // 3 entries of packed 4 bytes
```

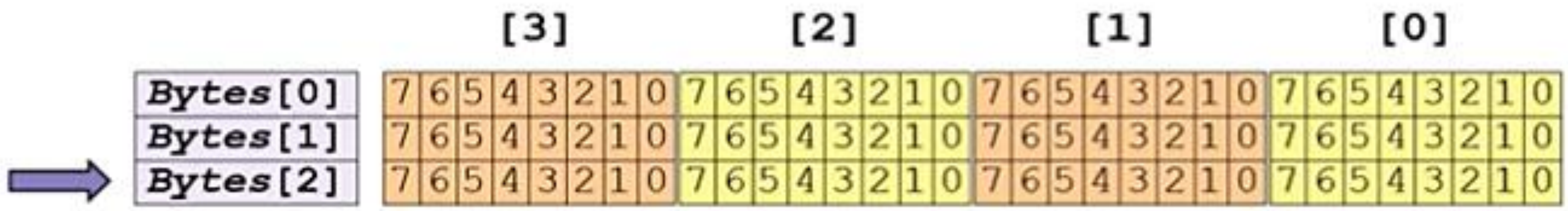



```
Bytes[2] = 32'hbeef_deed;
```

Unpacked assignment

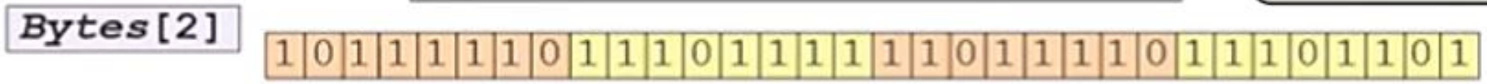
Array Example-1

```
logic [3:0][7:0] Bytes [3];   // 3 entries of packed 4 bytes
```



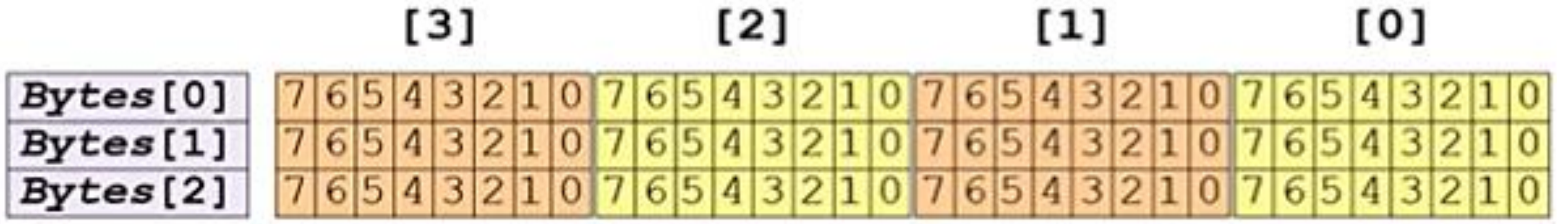

```
Bytes[2] = 32'hbeef_deed;
```

Unpacked assignment



Array Example-2

```
logic [3:0][7:0] Bytes [3];    // 3 entries of packed 4 bytes
```

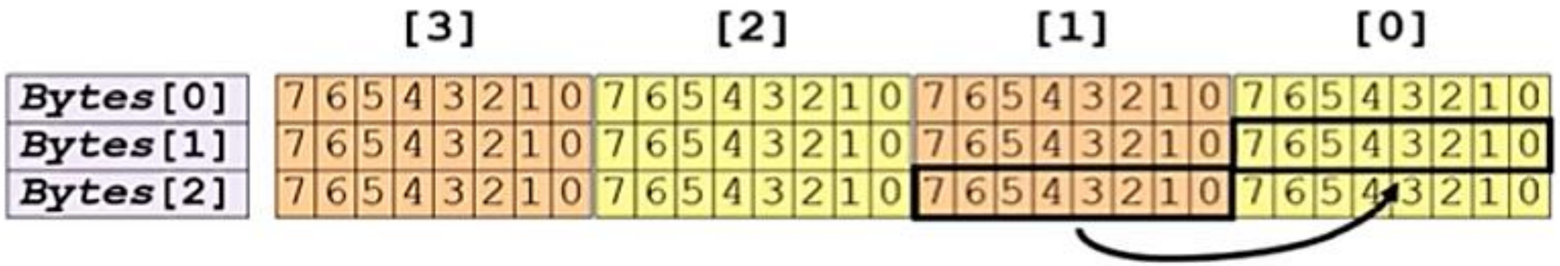


Packed assignment

```
Bytes[1][0] = Bytes[2][1];
```

Array Example-2

```
logic [3:0][7:0] Bytes [3];   // 3 entries of packed 4 bytes
```

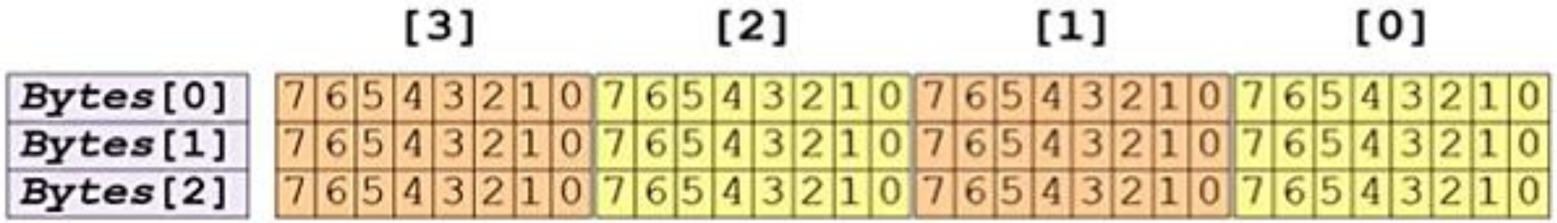


Packed assignment

```
Bytes[1][0] = Bytes[2][1];
```

Array Example-3

```
logic [3:0][7:0] Bytes [3];    // 3 entries of packed 4 bytes
```

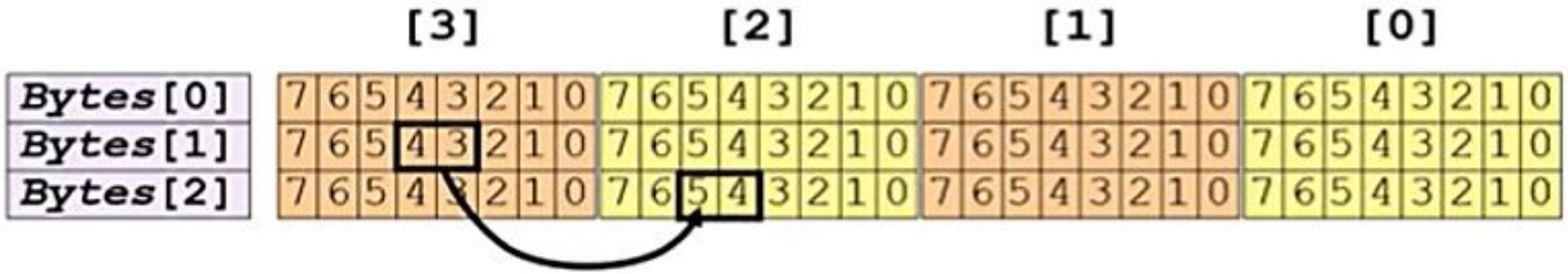



```
Bytes[2][2][5:4] = Bytes[1][3][4:3];
```

Packed bit/bit slice assignment

Array Example-3

```
logic [3:0][7:0] Bytes [3];   // 3 entries of packed 4 bytes
```




```
Bytes[2][2][5:4] = Bytes[1][3][4:3];
```

Packed bit/bit slice assignment

Dynamic Arrays

- Dynamic arrays can be allocated and resized during run-time simulation.
- A dynamic array is initially empty, and new space is allocated during simulation time using *new[ ]* constructor
- Dynamic array size() method
  - Returns the current size of a dynamic array.
- Dynamic array delete() method
  - Deletes array elements.

```
module dynamic_array_example;
  int array [];
  initial begin
    array = new[5];
    array = '{5, 10, 15, 20, 25};

    // Print elements of an array
    foreach (array[i]) $display("array[%0d] = %0d", i, array[i]);

    // size of an array
    $display("size of array = %0d", array.size());

    // Resizing of an array and copy old array content
    array = new[8] (array);
    $display("size of array after resizing = %0d", array.size());

    // Print elements of an array
    foreach (array[i]) $display("array[%0d] = %0d", i, array[i]);

    array.delete();
    $display("size of array after deleting = %0d", array.size());

  end
endmodule
```

Associative Arrays

- Useful for allocating large sparse arrays without compromising memory space
- Memory is allocated only when adding an element
- Array index can be of any type, e.g. string or class.
- An associative array implements a lookup table of the elements of its declared type.
  - The data type to be used as an index serves as the lookup key and imposes an ordering

```
byte assoc[byte], idx = 1;
initial begin
  // Initialize widely scattered values
  do begin
    assoc[idx] = idx;
    idx = idx << 1;
  end while (idx != 0);

  // Step through all index values with foreach
  foreach (assoc[i])
    $display("assoc[%h] = %h", i, assoc[i]);

  // Step through all index values with functions
  if (assoc.first(idx))   // Get first index
    do
      $display("assoc[%h]=%h", idx, assoc[idx]);
    while (assoc.next(idx));  // Get next index

  // Find and delete the first element
  void'(assoc.first(idx));
  void'(assoc.delete(idx));
  $display("The array now has %0d elements", assoc.num());
end
```

Associative Array-Methods

Function	Description
num ();	Returns the number of entries in the associative array
size ();	Also returns the number of entries, if empty 0 is returned
delete ([input index]);	index when specified deletes the entry at that index, else the whole array is deleted
exists (input index);	Checks whether an element exists at specified index; returns 1 if it does, else 0
first (ref index);	Assigns to the given index variable the value of the first index; returns 0 for empty array
last (ref index);	Assigns to given index variable the value of the last index; returns 0 for empty array
next (ref index);	Finds the smallest index whose value is greater than the given index
prev (ref index);	Finds the largest index whose value is smaller than the given index

Associative Array-Example

```
module sv;
  bit[7:0] assoc[bit[7:0]];
  bit[7:0] idx = 1;

  initial begin
    do begin
      assoc[idx] = idx;
      idx = idx << 1;
    end while (idx != 0);

    foreach (assoc[i])
      $display("assoc[%d]= %0d", i , assoc[i]);

    $display("The array now has %0d elements",assoc.num());
    assoc.delete(4);
    foreach (assoc[i])
        $display("assoc[%d]= %d", i , assoc[i]);
    $display("The array now has %0d elements",assoc.num());
end
endmodule
```

```
assoc[  1]= 1
assoc[  2]= 2
assoc[  4]= 4
assoc[  8]= 8
assoc[ 16]= 16
assoc[ 32]= 32
assoc[ 64]= 64
assoc[128]= 128
The array now has 8 elements
```

```
assoc[  1]=   1
assoc[  2]=   2
assoc[  8]=   8
assoc[ 16]=  16
assoc[ 32]=  32
assoc[ 64]=  64
assoc[128]= 128
The array now has 7 elements
```

Structs (Packed vs. Unpacked)

Unpacked struct Example

- Each member has a distinct memory location
- Each member can be assigned a value independently
- Entire set of struct members can be assigned via member name (position independent)
- Operation on the entire unpacked structure is illegal

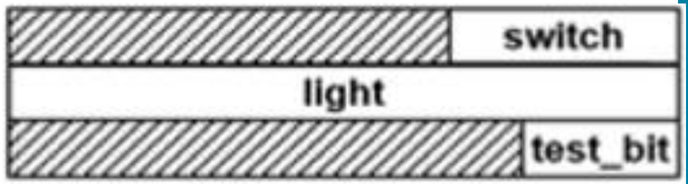
```
typedef enum logic[1:0] {OFF = 2'd0, ON= 2'd3} switch_val_enum;

typedef enum  {RED, GREEN, BLUE} colors_enum;

typedef struct {
    switch_val_enum  switch;   // 2  bits
    colors_enum      light;    // 32 bits
    logic            test_bit; // 1 bit
} sw_lgt_pair_unpacked_struct;

initial begin
sw_lgt_pair_unpacked_struct slp;

slp.light = GREEN;
slp = '0;            //illegal
slp = '{switch:ON, light:RED, test_bit:1'b0};
//slp = '{test_bit:1'b0, switch:ON, light:RED};
//position independent
```



Packed struct Example

- Can be accessed separately or as a single word
- Entire set of struct members can be assigned via member name (position independent)
  - Similar to unpacked struct
- Operation on the entire packed structure is legal
- Multiple views are possible and allowed
  - Additional memory/runtime overhead to track both views

```
module struct_sv;
typedef enum logic[1:0] {OFF = 2'd0, ON= 2'd3} switch_val_enum;

typedef enum  {RED, GREEN, BLUE} colors_enum;

typedef struct packed{
    switch_val_enum  switch;   // 2  bits
    colors_enum      light;    // 32 bits
    logic            test_bit; // 1 bit
} sw_lgt_pair_packed_struct;

initial begin
sw_lgt_pair_packed_struct slp;

slp.light = GREEN;
slp = '0;
slp = '{switch:ON, light:RED, test_bit:1'b0};
//slp = '{test_bit:1'b0, switch:ON, light:RED};
//position independent
end
endmodule
```

34 … 33	32 … 1	0
switch	light	test_bit

Queues

Queues

```
data_type queue_name[$];
```

- Combines the best of a linked list and arrays
- Elements can be added anywhere in a queue
- A queue is declared with word subscripts containing a dollar sign: [$]
- The elements of a queue are numbered from 0 to $.
- Queue literals only have curly braces, and are missing the initial apostrophe of array literals

Queues - Methods

Function	Description
insert (<index>,<item>)	Inserts an item at a specified index.
- delete(<index>) - delete	-> Deletes an item at a specified index -> Deletes all elements in the queue.
size()	If the queue is not empty, return the number of items in the queue. Otherwise, it returns 0.
push_back(<item>)	Inserts an item at the end of the queue.
pop_back()	Returns and removes the last item of the queue.
push_front(<item>)	Inserts an item at the front of the queue.
pop_front()	Returns and removes the first item of the queue.
shuffle()	Shuffles items in the queue

Queues - Methods

```
int j = 1,
    q2[$] = {3,4},          // Queue literals do not use '
    q[$] = {0,2,3};         // {0,2,3}

initial begin
  q.insert(1, j);           // {0,1,2,3}  Insert j before ele #1
  q.delete(1);              // {0,2,3}    Delete element #1

  // These operations are fast
  q.push_front(6);          // {6,0,2,3}  Insert at front
  j = q.pop_back;           // {6,0,2}    j = 3
  q.push_back(8);           // {6,0,2,8}  Insert at back
  j = q.pop_front;          // {0,2,8}    j = 6
  foreach (q[i])
    $display(q[i]);         //            Print entire queue
  q.delete();               // {}         Delete queue
end
```

Queues – Example

```
module lab2_ex4;
    int j = 1;
    int q1[$] = {3,4};
    int q2[$] = {0,2,3};

    initial begin
        q2 = {q2[0], j, q2[1:$]};
        $display("q2 = %p", q2);
        q2 = {q2[0:2], q1, q2[3:$]};
        $display("q2 = %p", q2);
        q2 = {q2[0],q2[2:$]};
        $display("q2 = %p", q2);
        q2 = {6, q1};
        $display("q2 = %p", q2);
        j = q2[$];
        $display("j = %p", j);
        q2 = q2[0:$-1];
        $display("q2 = %p", q2);
        q2 = {q2, 8};
        $display("q2 = %p", q2);
        j = q2[0];
        $display("j = %p", j);
        q2 = q2[1:$];
        $display("q2 = %p", q2);
        q2 = {};
        $display("q2 = %p", q2);
    end
endmodule
```

```
Compiler version S-2021.09; Runtime version S-2021.09;
q2 = '{0, 1, 2, 3}
q2 = '{0, 1, 2, 3, 4, 3}
q2 = '{0, 2, 3, 4, 3}
q2 = '{6, 3, 4}
j =           4
q2 = '{6, 3}
q2 = '{6, 3, 8}
j =           6
q2 = '{3, 8}
q2 = '{}
```

Lab 2: (~90 mins)

Lab 2 Exercise 1

- Implement a memory model using a multidimensional array with 64 locations and 32 bits entry size, where:
  - The array contents are initialized randomly and then printed.
  - The last 3 bytes of each entry are accessed and then complemented.
  - Store the entries at the same location and print the modified array.

Lab 2 Exercise 2

- Design a sequence detector to detect 011
- Create a testbench to test your DUT
- **Driver**: create an input queue to drive your DUT
  - bit input_queue[$]= {0, 0, 1, 1, 0, 0, 0, 1, 1, 0};
- **Monitor**: create a queue to capture output
  - bit detect_design_queue[$];
- **Scoreboard**: compare the output queue with the queue from the golden model
  - bit detect_GoldenModel_queue[$]= {0, 0, 0, 1, 0, 0, 0, 0, 1, 0};

Lab 2 Exercise 2

seq.v

```verilog
module seq (
    input clk,
    input reset,
    input sig_in,
    output detect
);

reg [2:0] arr;

always @(posedge clk, posedge reset) begin
    if(reset)
        arr<=0;
    else
        arr<={sig_in, arr[2:1]};
    end

    assign detect = (arr==3'b110) ? 1:0;
endmodule
```