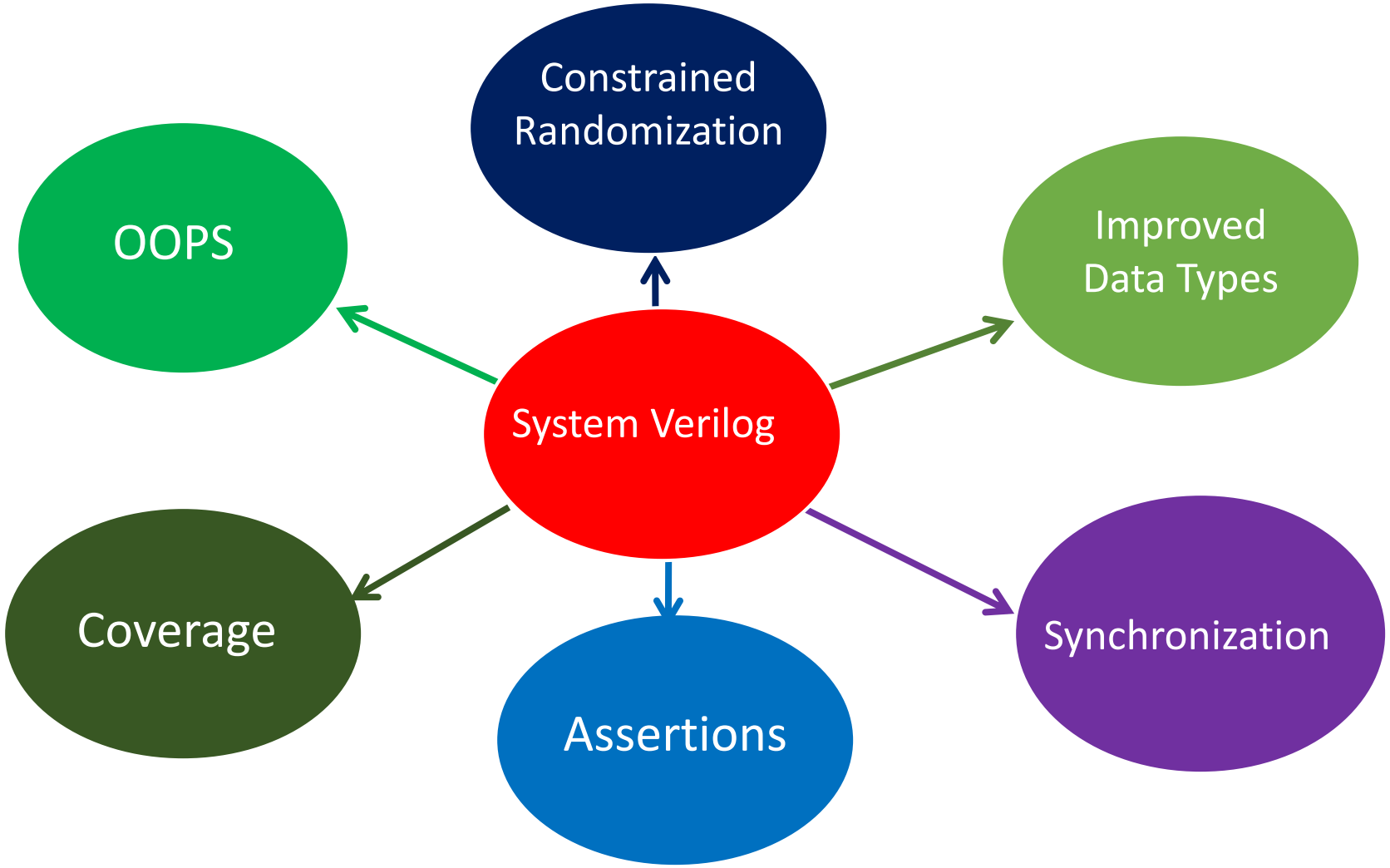
Introduction to System Verilog (Data Types & Arrays)

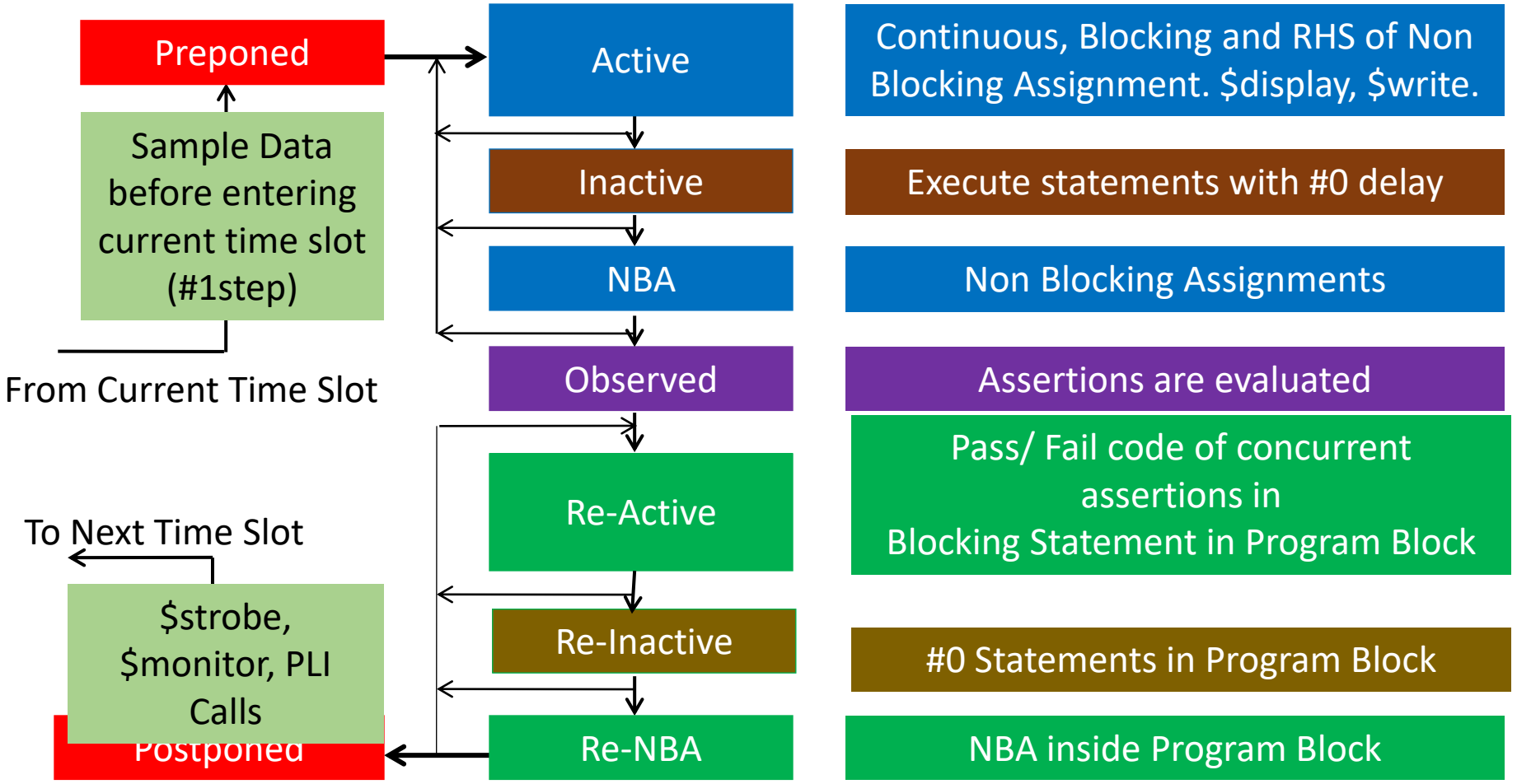
What is SV?

- System Verilog is a hardware description and Verification language (HDVL).
- The bulk of the verification functionality is based on the OpenVera language.
- System Verilog (IEEE Standard 1800-2017).
- It inherits features from Verilog, VHDL, C and C++.

Features of SV



Regions in SV



Data Type

- System Verilog offers following Data Types:
  - 4-State Type
  - 2-State Type
  - Real
  - Arrays
  - User Define
  - Structures
  - Unions
  - Strings
  - Enumerated Type
  - Class

4-State Type

Allowed values are 0, 1, X and Z.

Following 4-State types are included from Verilog:

- wire //Size: 1-bit Value: Z
- reg //Size: 1-bit Value: X
- integer // Size: 32-bit Value: X
- time // Size: 64-bit Value: X

User can define size for wire and reg.

integer is signed, all others are unsigned.

4-State Type

- Addition to System Verilog w. r. t. Verilog
  - logic //Size: 1-bit Value: X
- User can define size for logic.
- Logic is improved reg data type.
- Logic can be driven by continuous as well as procedural assignments.
- Logic has a limitation that it cannot be driven by multiple drivers in such case use wire.

Logic

Example1:

```
module and_gate ( input logic a, b, output logic c);
assign c= a & b;  //driving logic using continuous assignment
endmodule
```

Example2:

```
module flip_flop ( input logic din, clk, output logic dout);
always @ (posedge clk)
dout<=din;          //driving logic using procedural assignment
endmodule
```

Logic

Example4:

```
module example4 ( input logic a, b, ctrl, output logic c);
assign c= ctrl?a:1'bZ; //driving logic using continuous assignment

assign c= !ctrl?b:1'bZ; //driving logic using continuous assignment
endmodule
```

****E: Compilation error**
(use wire to achieve this functionality).

2-State Type

- Allowed values are 0 and 1 only.
- System Verilog offers following 2-State Data Types :
  - shortint //Size: 16-bit Value: 0
  - int //Size: 32-bit Value: 0
  - longint //Size: 64-bit Value: 0
  - byte //Size: 8-bit Value: 0
  - bit //Size: 1-bit Value: 0
- User can define size for bit.
- All are signed except bit which is unsigned.

Example1

```
module example1;
int a;
int unsigned b;                 //unsigned integer
bit signed [7:0] c;             //same as byte

initial
begin
a=-32'd127;
b='1;                           //SV offers un-sized literal to fill all
c='0;                           // locations with given number
end

endmodule
```

Example2

```
module example2;
int a;
logic [31:0] b='Z;              //b=32'hzzzz_zzzz

initial
begin
a=b;                            //a=32'h0000_0000
b=32'h123x_5678;
if($isunknown(b))
  $display("b is unknown");
else
  $display("b is known");
end

endmodule
```

Void

- void keyword represents non existing data.
- It can be used as return type of functions to indicate nothing is returned.

Example:

```
function  void display;
$display(“Hello”);
endfunction
```

Usage:

```
void=display( );
```

Arrays

- Arrays are used to group elements of same type.

- Arrays can be categorized as following:
  - Fixed Array
    - Packed Array
    - Unpacked Array
  - Dynamic Array
  - Queues
  - Associative Array

Fixed Array

- Array whose size is fixed during compilation time is called as Fixed Array.

- Size of fixed array cannot be modified during run time.

Examples

```
int array1 [15];          //array of int containing 15 elements
//Equivalent to int array1 [0:14]
int array2 [0:14];
logic array3 [7:0];       //array of logic containing 8 elements
```

Unpacked Array

- Unpacked Arrays can be declared by adding size after array name.

- Unpacked Arrays can be made of any data type.

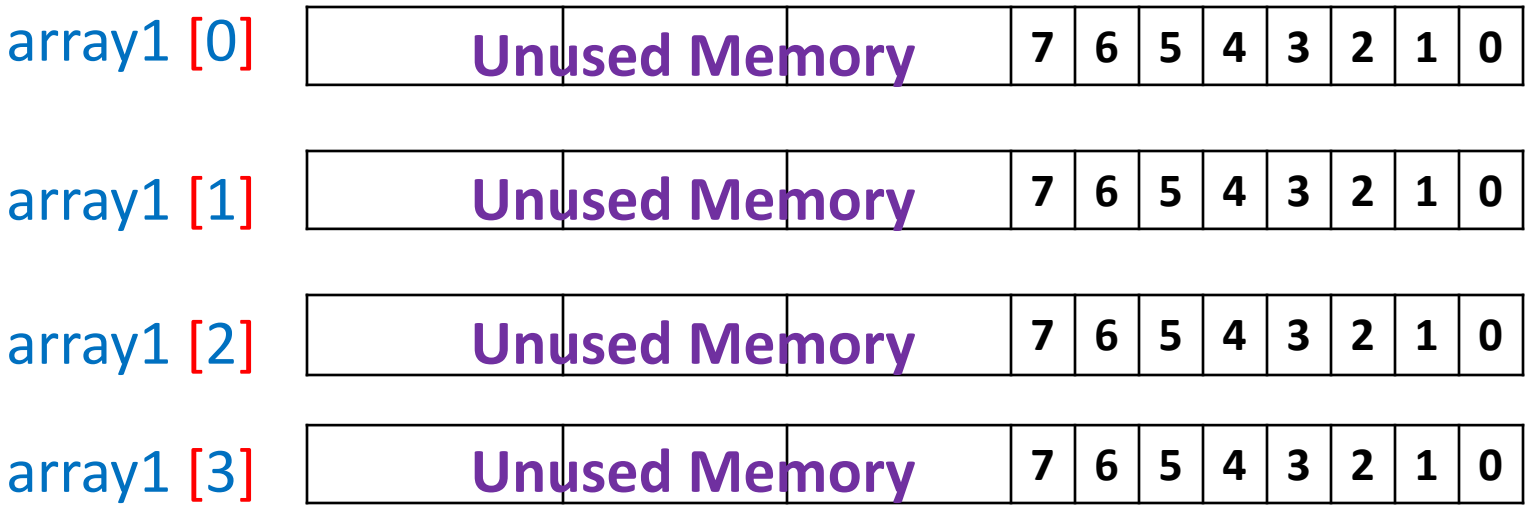
  Example:

```
int array1 [16] [8];        //16 rows , 8 columns
bit array2 [3:0] [7:0];     //4 rows , 8 columns
bit [7:0] array3 [4];       //4 rows each containing 8 bits
```

- System Verilog stores each element of an unpacked array in a longword (32-bit).

Unpacked Array

bit [7:0] array1 [4];



Unpacked Array

Initializing Array:

```
int array1 [2] [4] = ‘{ ‘{ 1, 2, 3, 4 } , ‘{ 5, 6, 7, 8 } };
int array2 [2] [3] = ‘{ ‘{ 1, 3, 6 } , ‘{ 3 {2} };
            // same as ‘{ 1, 3, 6 } , ‘{ 2, 2, 2 }

int array3 [0:5] = ‘{1:5, 3:1, default: 0};
            // same as ‘{0, 5, 0, 1, 0, 0}

int array4 [0:2] [1:4] = ‘{3 { ‘{ 2 {1, 2} } } };
    // same as ‘{ ‘{1, 2, 1, 2} , ‘{1, 2, 1, 2} , ‘{1, 2, 1, 2} }

int array5 [2] [2] [2] = ‘{ ‘{ ‘{4, 5}, ‘{3, 1} }, ‘{ ‘{1, 7}, ‘{2, 5} } };
```

Unpacked Array

Accessing Array

```
int array1 [2] [4];
int array2 [0:5];
byte array3 [0:2] [1:4];
int a, b;
byte c;
```

```
a= array1[1] [3];
b= array2[4];
c= array3[1] [2];
```

Basic Array Operation

- Arrays can be manipulated using for and foreach loop

```
bit [7:0] array1[10], array2[10] ;

initial
begin
for ( int i=0; i <$size(array1); i++)   //$size returns size of array
array1[ i ]= 0;

foreach(array2[ k ])                    //k is defined implicitly
array2[ k ]=$random;

end
```

Basic Array Operation

Example:

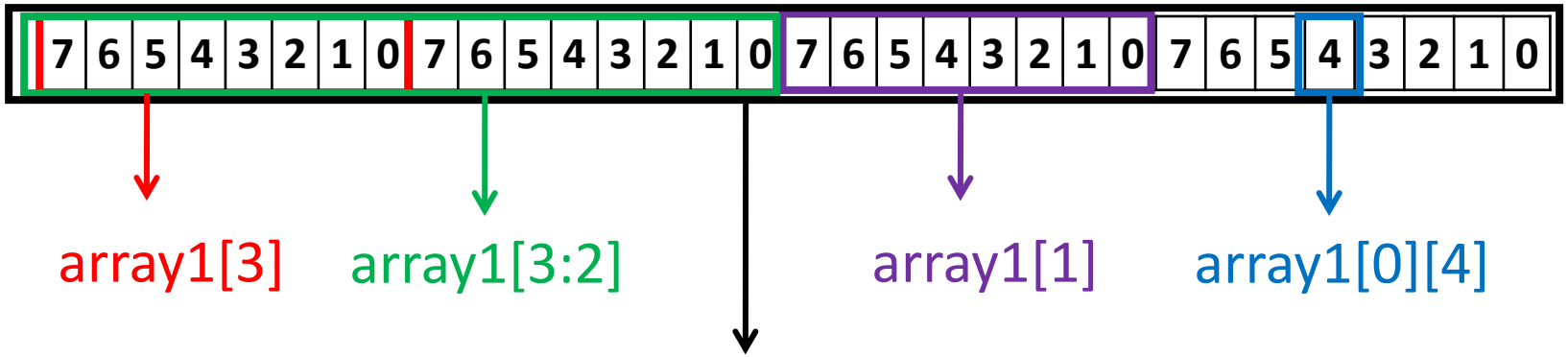
```
bit [7:0] array1[10] [20];

initial
begin
array1='{10 { '{0:2, 1 : 0 , default:$random} } };

foreach(array1[i ,k])
$display("array1[%0d] [%0d]=%0d", i, k, array1[i] [k]);

end
```

Packed Array

- Packed Arrays can be declared by adding size before array name.
- One dimensional packed arrays are also referred as vectors.
- Packed array is a mechanism of subdividing a vector into subfields which can be accessed as array elements.
- Packed array represents contiguous set of bits.
- Packed array can be made of single bit data (logic, bit, reg), enumerated type or other packed arrays.

Packed Array

bit [3:0] [7:0] array1;



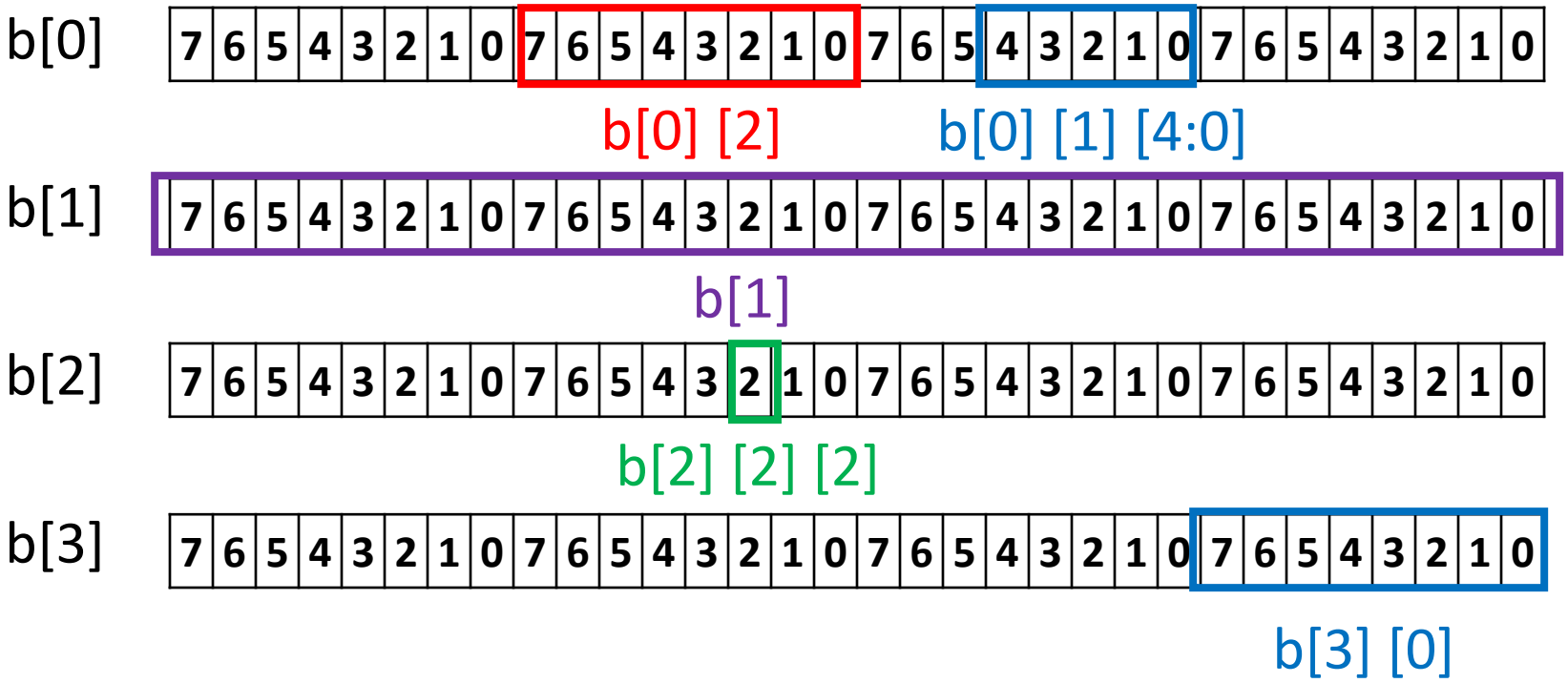

```
bit [7:0] a, b;
bit [15:0] c;
bit d;
bit [31:0] e;
```

```
a=array1[3];
b=array1[1];
c=array1[3:2];
d=array1[0][4];
e=array1;
```

Packed Array

Mixture of Packed and Unpacked Array

bit [3:0] [7:0] b [4];



Packed vs. Unpacked Array

- Packed arrays are handy if user wants to access array with different combination.
- If user want to wait for change in array(i.e. @), in that case packed array will be preferred over unpacked array.
- Only fixed size arrays can be packed. Therefore, it is not possible to pack following arrays:
  - Dynamic Arrays
  - Queues
  - Associative Arrays

Operation on Arrays

```
int A [0:7] [15:0] , B [0:7] [15:0];
```

- Following operation are possible for both packed and unpacked arrays.

- Both array A and B should be of same type and size.

```
A=B;                    //Copy Operation
A[0:3]= B[0:3];           //Slice and Copy
A[1+:4]= B[3+:4];
A[5]=B[5];
A==B                    //Comparison Operations
A[2:4]!=B[2:4];
```

Operation on Arrays

```
bit [3:0] [7:0] A;
```

- Following operation are only allowed in packed arrays:

```
A=0;
A=A + 3;
A=A * 2;
A=‘1;
A=A & 32’d255;
A[3:1]=16’b1101_1110_0000_1010;
```

Dynamic Array

- Dynamic arrays are unpacked arrays whose size can be set and changed during simulation time.
- new constructor is used to set or change size of Dynamic Array.
- size() method returns current size of array.
- delete() method is used to delete all elements of the array.

Dynamic Array

```
int dyn1 [ ];          //Defining Dynamic Array (empty subscript)
int dyn2 [4] [ ];

initial
begin
dyn1=new[10];                              //Allocate 10 elements
foreach (dyn1[ i ]) dyn1[ i ]=$random;    // Initializing Array
dyn1=new[20] (dyn1);                       // Resizing array and
                                  // Copying older values
dyn1=new[50];    // Resizing to 50 elements Old Values are lost
dyn1.delete;                               // Delete all elements
end
```

Dynamic Array

```
int dyn1 [ ]= ‘{5, 6, 7, 8} ;        //Alternative way to define size

initial
begin
  repeat (2)
   if (dyn1.size != 0)
     begin
     foreach(dyn1 [ i ] ) $display(“dyn1[%0d]=%0d”, i, dyn[ i ] );
     dyn1.delete;
     end
  else
     $display(“Array is empty”);
end
```