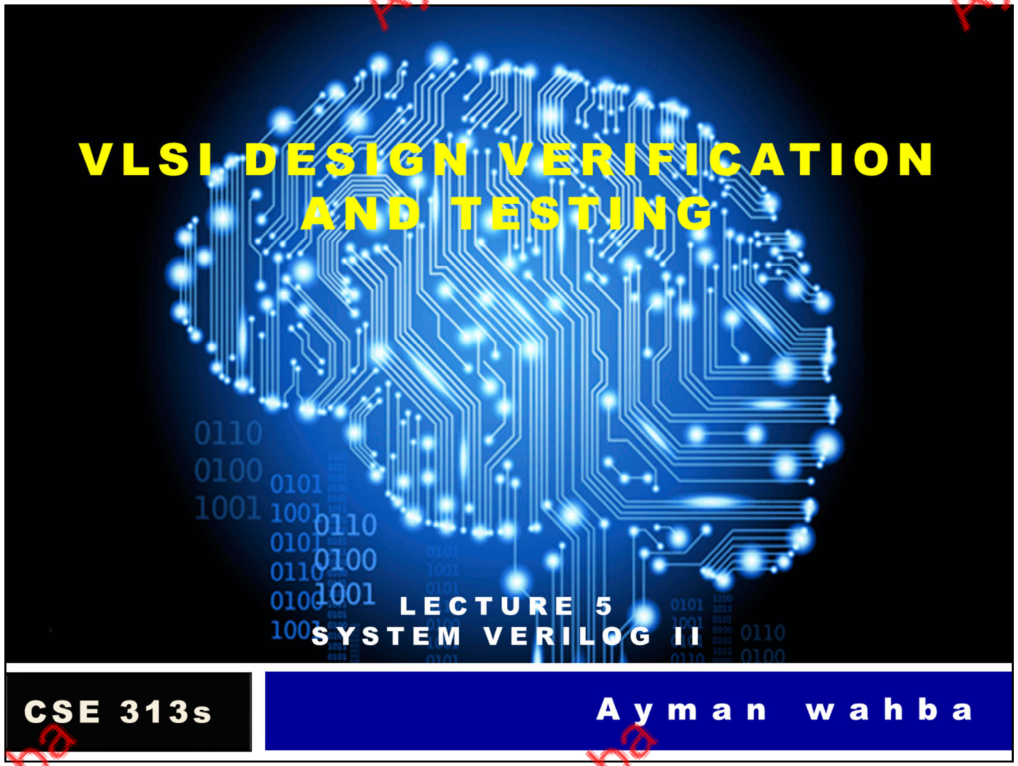
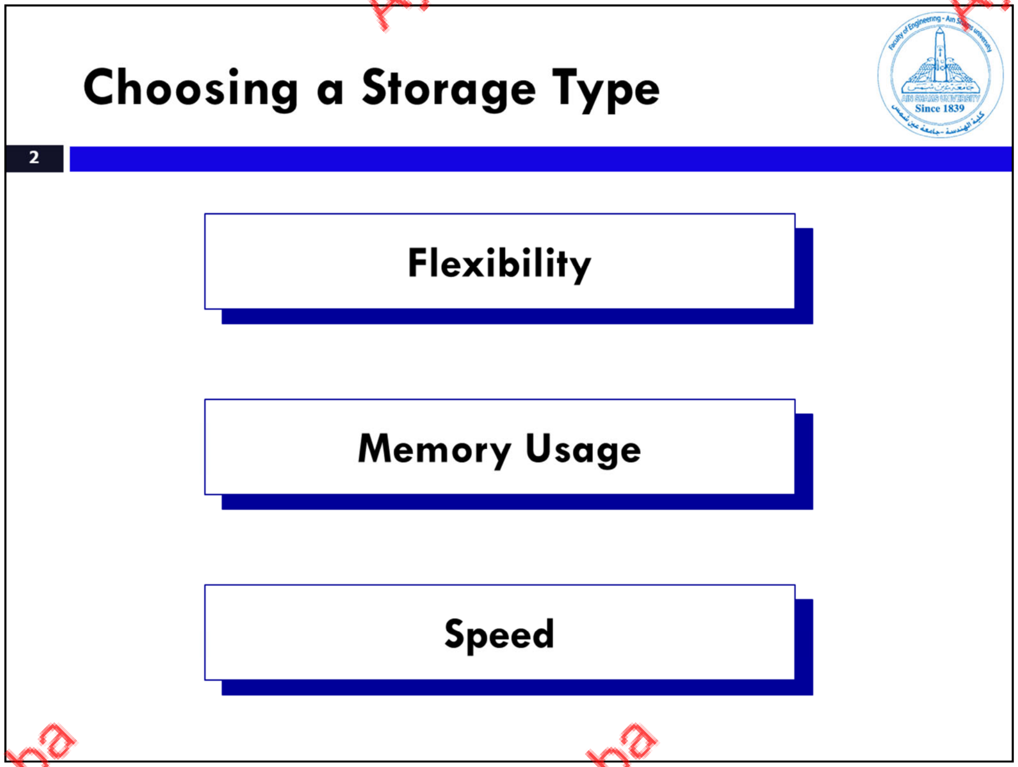


VLSI DESIGN VERIFICATION AND TESTING

LECTURE 5
SYSTEM VERILOG II

CSE 313s

Ayman wahba



Choosing a Storage Type

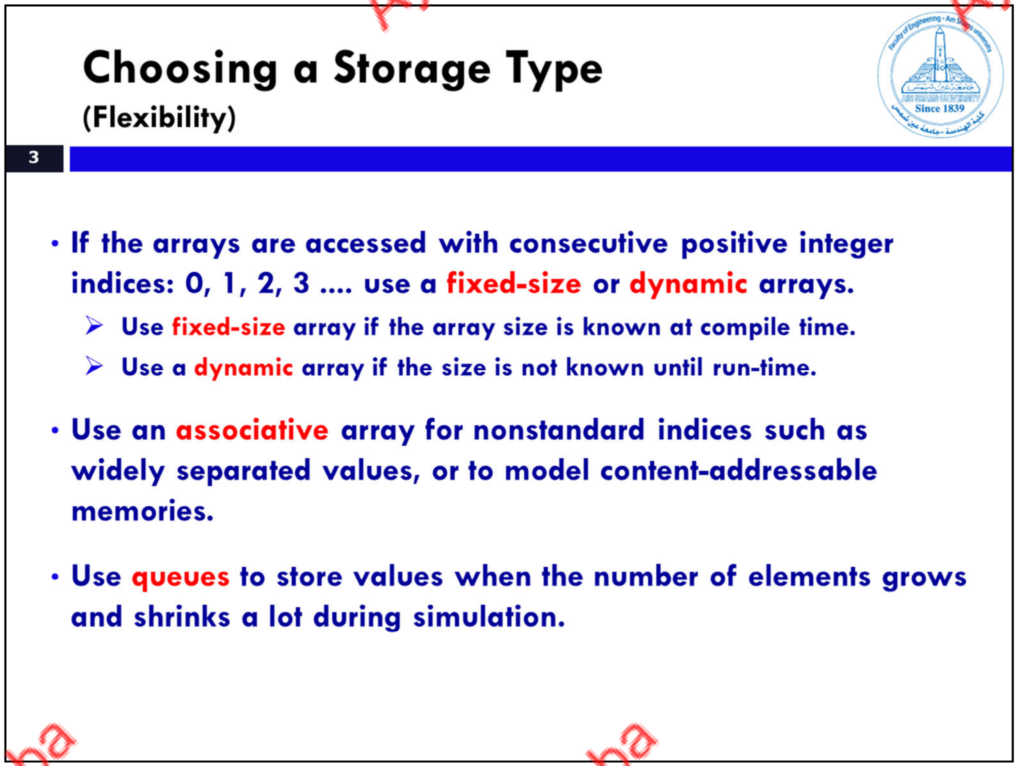
2

Flexibility

Memory Usage

Speed

- Here are some guidelines for choosing the right storage type based on *flexibility*, *memory usage*, *speed*, and *sorting*. These are just rules of thumb, and results may vary between simulators.

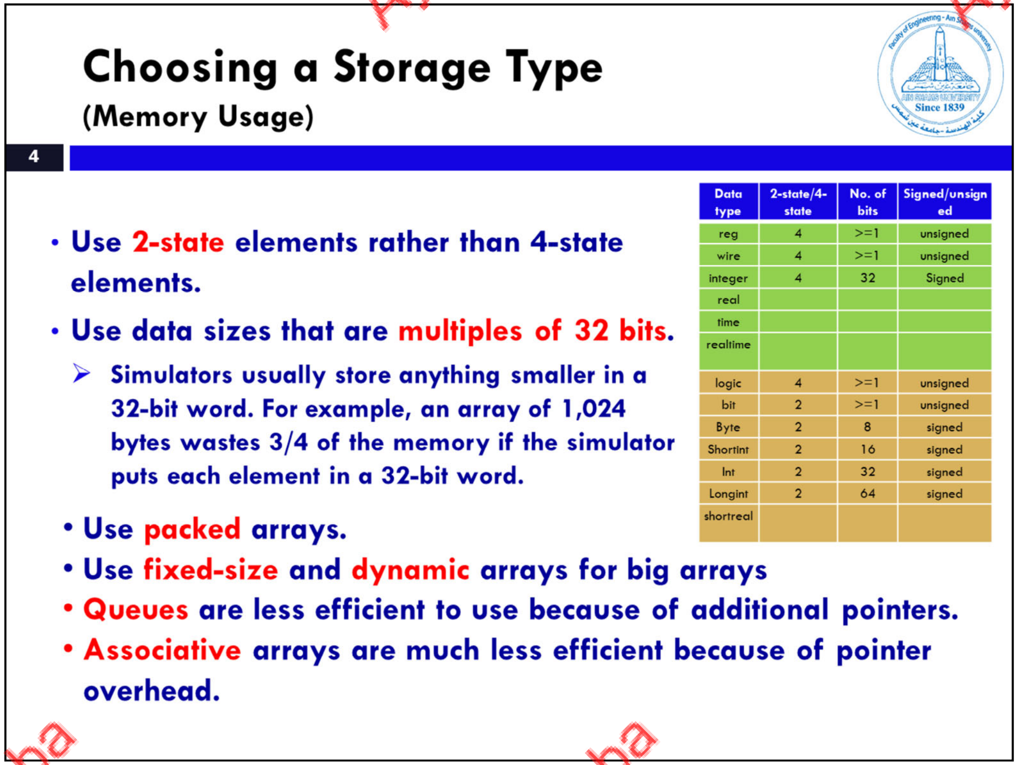


Choosing a Storage Type

(Flexibility)

- If the arrays are accessed with consecutive positive integer indices: 0, 1, 2, 3 .... use a fixed-size or dynamic arrays.
  - Use fixed-size array if the array size is known at compile time.
  - Use a dynamic array if the size is not known until run-time.
- Use an associative array for nonstandard indices such as widely separated values, or to model content-addressable memories.
- Use queues to store values when the number of elements grows and shrinks a lot during simulation.

- Use a *fixed-size* or *dynamic* array if it is accessed with consecutive positive integer indices: 0, 1, 2, 3 ....
- Choose a *fixed-size* array if the array size is known at compile time, or choose a *dynamic* array if the size is not known until run-time.
- Choose *associative* arrays for nonstandard indices such as widely separated values because of random values or addresses. Associative arrays can also be used to model content-addressable memories.
- *Queues* are a good way to store values when the number of elements grows and shrinks a lot during simulation, such as a scoreboard that holds expected values.



Choosing a Storage Type

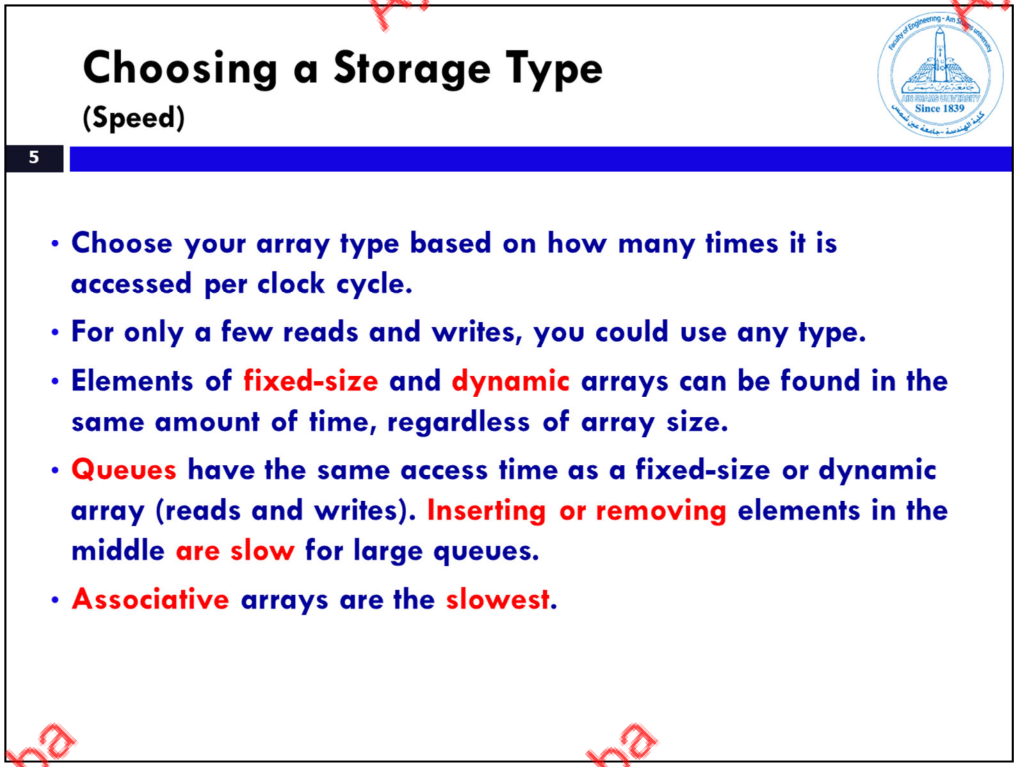
(Memory Usage)

4

Data type	2-state/4-state	No. of bits	Signed/unsigned
reg	4	>=1	unsigned
wire	4	>=1	unsigned
integer	4	32	Signed
real			
time			
realtime			
logic	4	>=1	unsigned
bit	2	>=1	unsigned
Byte	2	8	signed
Shortint	2	16	signed
Int	2	32	signed
Longint	2	64	signed
shortreal			

- Use 2-state elements rather than 4-state elements.
- Use data sizes that are multiples of 32 bits.
  - ➢ Simulators usually store anything smaller in a 32-bit word. For example, an array of 1,024 bytes wastes 3/4 of the memory if the simulator puts each element in a 32-bit word.
- Use packed arrays.
- Use fixed-size and dynamic arrays for big arrays
- Queues are less efficient to use because of additional pointers.
- Associative arrays are much less efficient because of pointer overhead.

- If you want to reduce the simulation memory usage, use *2-state* elements.
- You should choose *data sizes that are multiples of 32 bits* to avoid wasted space. Simulators usually store anything smaller in a 32-bit word.
  - ➢ For example, an array of 1,024 bytes wastes 3/4 of the memory if the simulator puts each element in a 32-bit word.
- Packed arrays can also help conserve memory.
- For arrays that hold up to a thousand elements, the type of array that you choose does not make a big difference in memory usage (unless there are many instances of these arrays).
- For arrays with a thousand to a million active elements, fixed-size and dynamic arrays are the most memory efficient. (You may want to reconsider your algorithms if you need arrays with more than a million active elements).
- *Queues* are slightly *less efficient* to access than fixed-size or dynamic arrays because of *additional pointers*. However, if your data set grows and shrinks often, and you store it in a dynamic memory, you will have to manually call new [] to allocate memory and copy. This is an expensive operation and would wipe out any gains from using a dynamic memory.
- Note that each element in an *associative array* can take several times more memory than a fixed-size or dynamic memory because of pointer overhead.

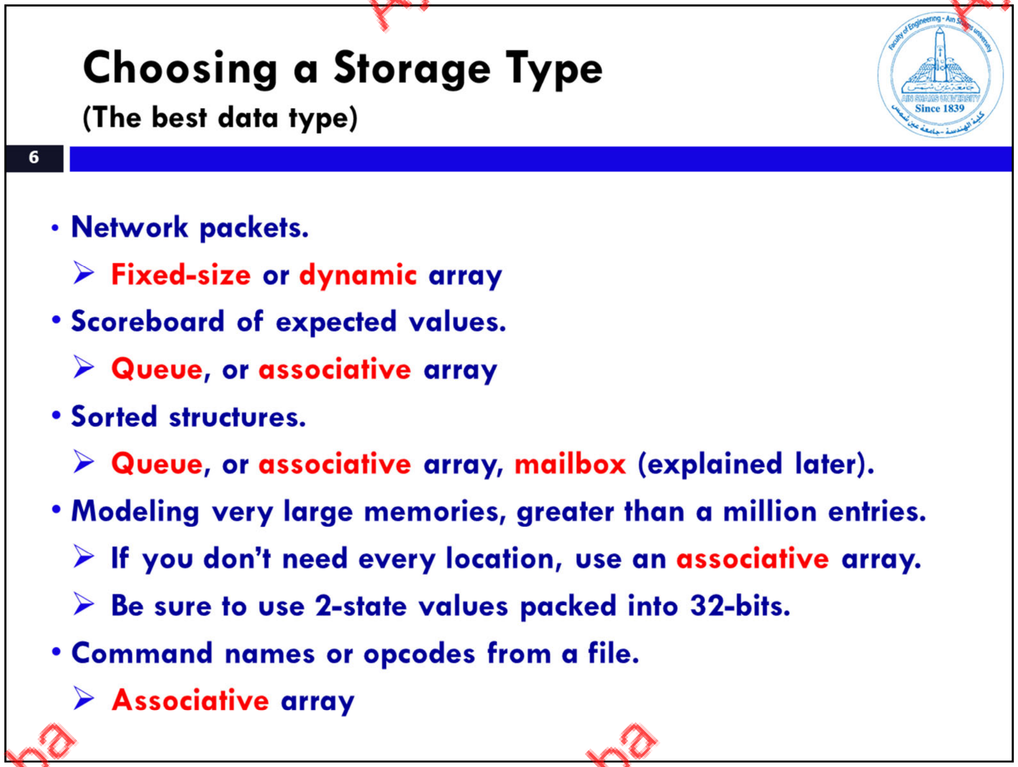


Choosing a Storage Type
(Speed)

5

- Choose your array type based on how many times it is accessed per clock cycle.
- For only a few reads and writes, you could use any type.
- Elements of fixed-size and dynamic arrays can be found in the same amount of time, regardless of array size.
- Queues have the same access time as a fixed-size or dynamic array (reads and writes). Inserting or removing elements in the middle are slow for large queues.
- Associative arrays are the slowest.

- Choose your array type based on how many times it is accessed per clock cycle.
- For only a few reads and writes, you could use any type, as the overhead is minor compared with the DUT. As you use an array more often, its size and type matters.
- *Fixed-size* and *dynamic arrays* are stored in contiguous memory, and so any element can be found in the same amount of time, regardless of array size.
- *Queues* have almost the same access time as a fixed-size or dynamic array for reads and writes. The first and last elements can be pushed and popped with almost no overhead.
    - ➢ Inserting or removing elements in the middle requires many elements to be shifted up or down to make room. *If you need to insert new elements into a large queue, your testbench may slow down*, and so consider changing how you store new elements.
- When reading and writing *associative arrays*, the simulator must search for the element in memory. The LRM does not specify how this is done, but popular ways are hash tables and trees. These require more computation than other arrays, and therefore *associative arrays are the slowest*.



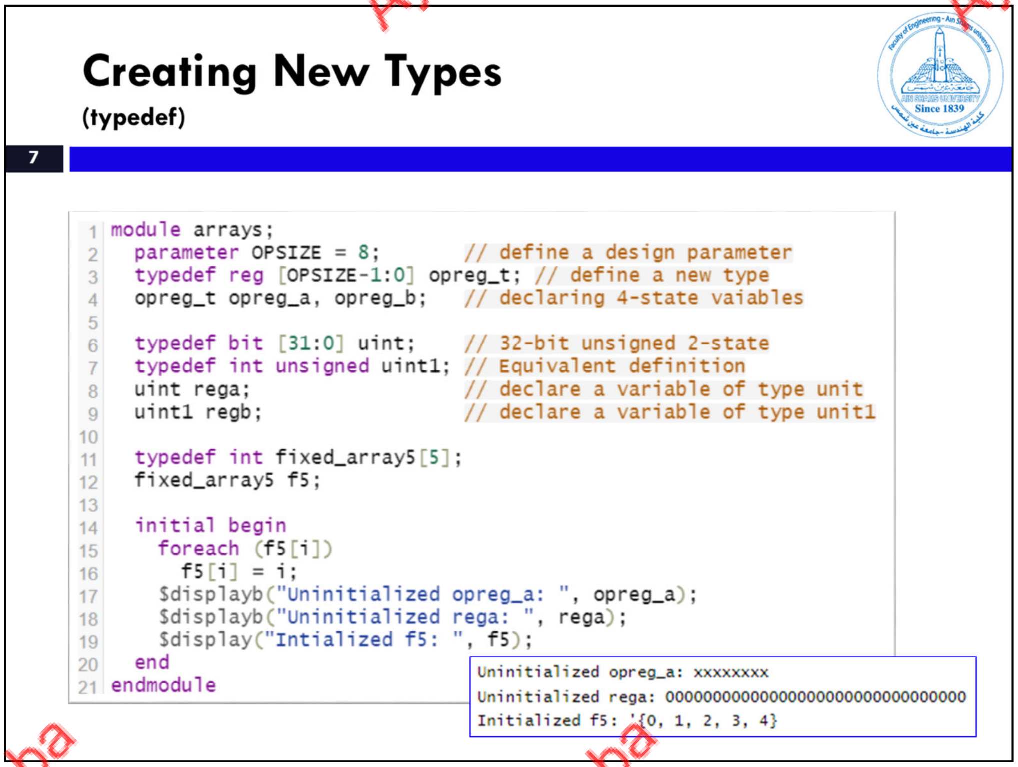
Choosing a Storage Type

(The best data type)

6

- Network packets.
  - ➢ Fixed-size or dynamic array
- Scoreboard of expected values.
  - ➢ Queue, or associative array
- Sorted structures.
  - ➢ Queue, or associative array, mailbox (explained later).
- Modeling very large memories, greater than a million entries.
  - ➢ If you don't need every location, use an associative array.
  - ➢ Be sure to use 2-state values packed into 32-bits.
- Command names or opcodes from a file.
  - ➢ Associative array

- *Network packets*. Properties: fixed size, accessed sequentially.
  - ➢ Use a fixed-size or dynamic array for fixed- or variable-size packets.
- *Scoreboard* of expected values. Properties: size not known until run-time, accessed by value, and a constantly changing size.
  - ➢ In general, use a queue, as you are continually adding and deleting elements during simulation.
  - ➢ If you can give every transaction a fixed id, such as 1,2,3 .... , you could use this as an index into the queue.
  - ➢ If your transaction is filled with random values, you can just push them into a queue and search for unique values.
  - ➢ If the scoreboard may have hundreds of elements, and you are often inserting and deleting them from the middle, an associative array may be faster.
- *Sorted structures*.
  - ➢ Use a queue if the data comes out in a predictable order.
  - ➢ Use an associative array if the order is unspecified.
  - ➢ If the scoreboard never needs to be searched, just store the expected values in a *mailbox*, as will be shown later.
- *Modeling very large memories, greater than a million entries*.
  - ➢ If you don't need every location, use an associative array as a sparse memory.
  - ➢ Be sure to use 2-state values packed into 32-bits.
- *Command names or opcodes from a file*. Property: translate a string to a fixed value.
  - ➢ Read string from a file, and then look up the commands or opcodes in an associative array using the command as a string index.



Creating New Types

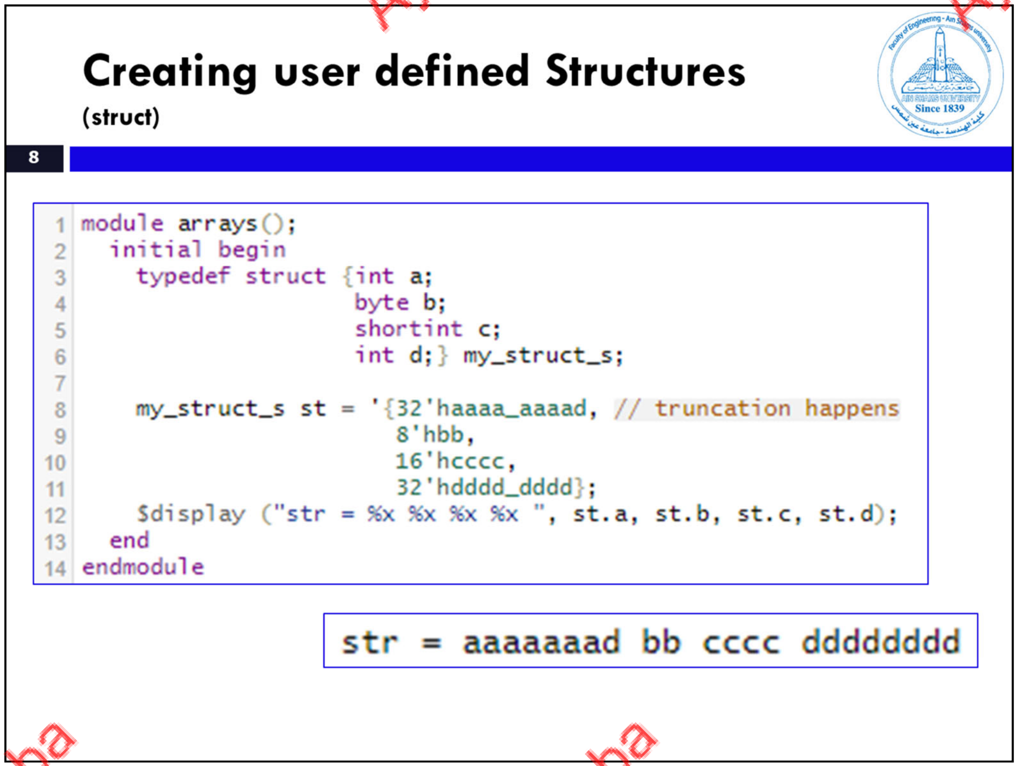
(typedef)

7

```
module arrays;
  parameter OPSIZE = 8;         // define a design parameter
  typedef reg [OPSIZE-1:0] opreg_t; // define a new type
  opreg_t opreg_a, opreg_b;     // declaring 4-state vaiables

  typedef bit [31:0] uint;      // 32-bit unsigned 2-state
  typedef int unsigned uint1;   // Equivalent definition
  uint rega;                    // declare a variable of type unit
  uint1 regb;                   // declare a variable of type unit1

  typedef int fixed_array5[5];
  fixed_array5 f5;

  initial begin
    foreach (f5[i])
      f5[i] = i;
    $displayb("Uninitialized opreg_a: ", opreg_a);
    $displayb("Uninitialized rega: ", rega);
    $display("Intialized f5: ", f5);
  end
endmodule
```

```
Uninitialized opreg_a: xxxxxxxx
Uninitialized rega: 00000000000000000000000000000000
Initialized f5: '{0, 1, 2, 3, 4}
```

- You can create new types using the *typedef* statement.
- In SystemVerilog you create a new type as shown in the above code.
- It is recommended to distinguish between the names of user defined types and standard types by adding for example "_t" to the type name.
- SystemVerilog lets you copy between these basic types with no warning, either extending or truncating values if there is a width mismatch.
- Note that parameter and typedef statements can be put in a package so that they can be shared across the design and testbench. (packages are explained later).
- One of the most useful types you can create is an unsigned, 2-state, 32-bit integer. Most values in a testbench are positive integers such as field length or number of transactions received, and so having a signed integer can cause problems.



Creating user defined Structures

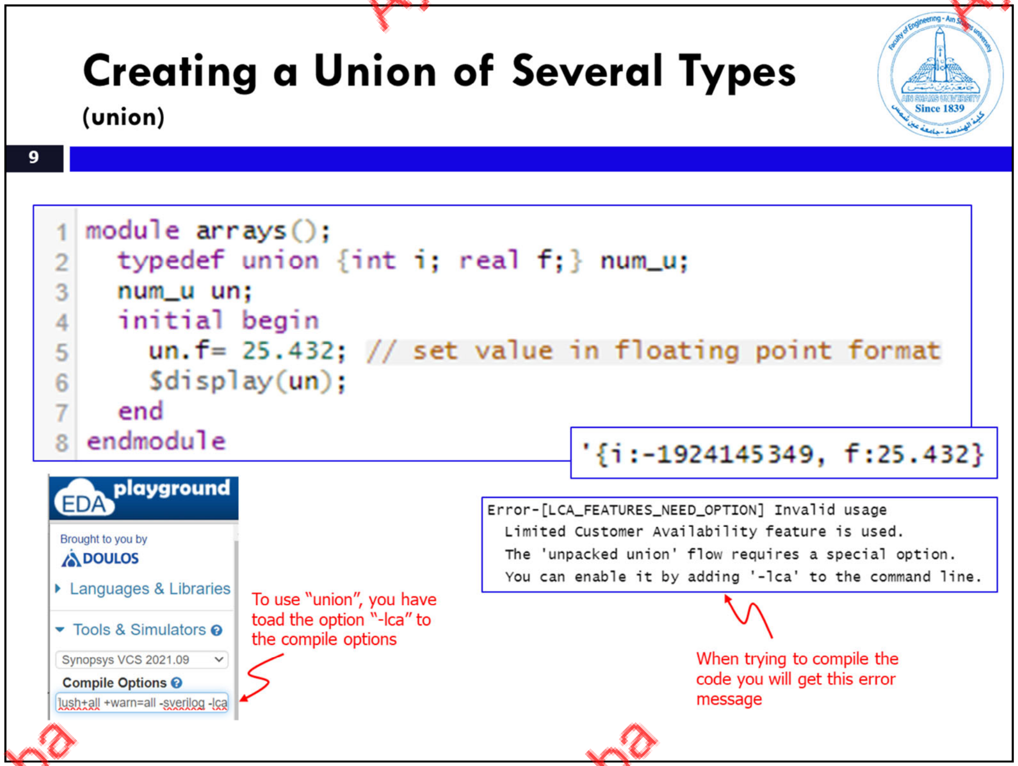
(struct)

8

```
module arrays();
  initial begin
    typedef struct {int a;
                    byte b;
                    shortint c;
                    int d;} my_struct_s;

    my_struct_s st = '{32'haaaa_aaaad, // truncation happens
                       8'hbb,
                       16'hcccc,
                       32'hdddd_dddd};
    $display ("str = %x %x %x %x ", st.a, st.b, st.c, st.d);
  end
endmodule
```

```
str = aaaaaaad bb cccc dddddddd
```

- One of the biggest limitations of Verilog is the lack of data structures. In System verilog you can create a structure using the struct statement similar to what is available in C.
- A struct groups data fields together. The *data fields can be of different types.*
- If you want to model a complex data type, put it in a struct.
- It is recommended to use the suffix "_s" when declaring a struct. This makes it easier to spot user defined types, simplifying the process of sharing and reusing code.
- You can assign multiple values to a struct just like an array, either in the declaration or in a procedural assignment. Just surround the values with an apostrophe and braces as shown above.
- Note that %x is sometimes used instead of %h.



Creating a Union of Several Types

(union)

9

```
module arrays();
  typedef union {int i; real f;} num_u;
  num_u un;
  initial begin
    un.f= 25.432; // set value in floating point format
    $display(un);
  end
endmodule
```

```
'{i:-1924145349, f:25.432}
```

```
Error-[LCA_FEATURES_NEED_OPTION] Invalid usage
  Limited Customer Availability feature is used.
  The 'unpacked union' flow requires a special option.
  You can enable it by adding '-lca' to the command line.
```

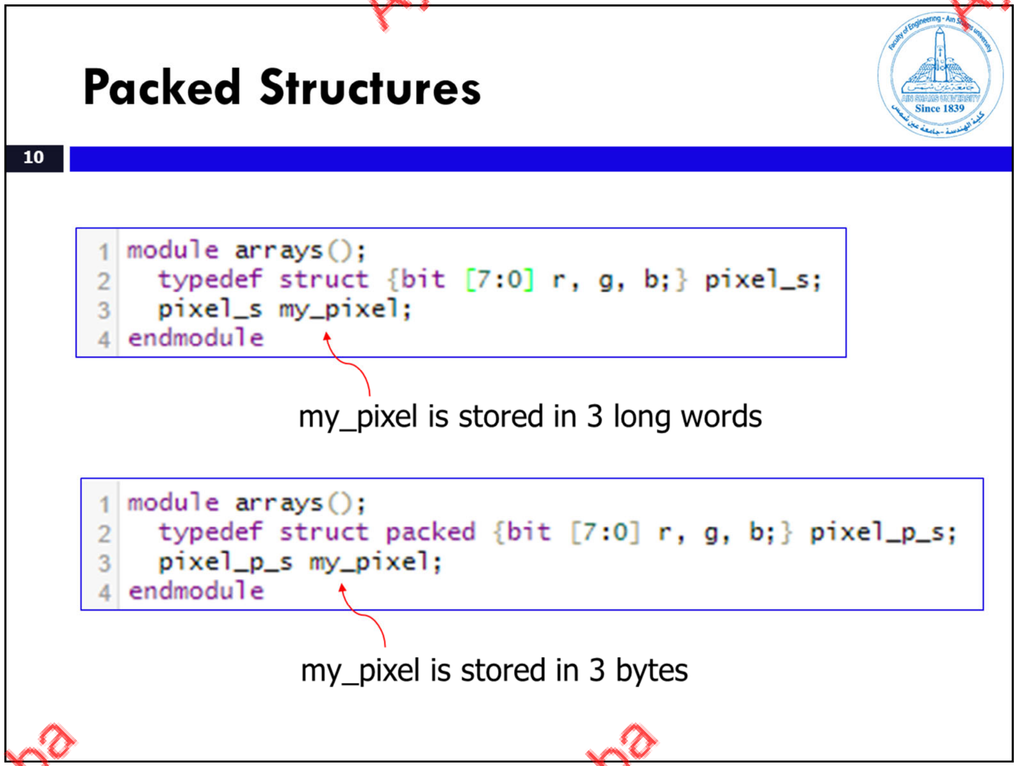
Synopsys VCS 2021.09

Compile Options: -lush+all +warn=all -sverilog -lca

To use “union”, you have toad the option “-lca” to the compile options

When trying to compile the code you will get this error message

- A union is similar to struct in its declaration, but is different.
- Struct has different elements, and its size is the sum of the sizes of its elements.
- A union is a special data type available that allows storing different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple purposes. Its size is the size of the largest data type of its elements.
- The shown code stores both the integer *i* and the real *f* in the same location.
- Unions are useful when you frequently need to read and write a register in several different formats.
- Unions may help squeeze a few bytes out of a structure, but at the expense of having to create and maintain a more complicated data structure.
- To use unions you must add the argument –lca (Limited Customer Availability) to the command line of the compiler.



Packed Structures

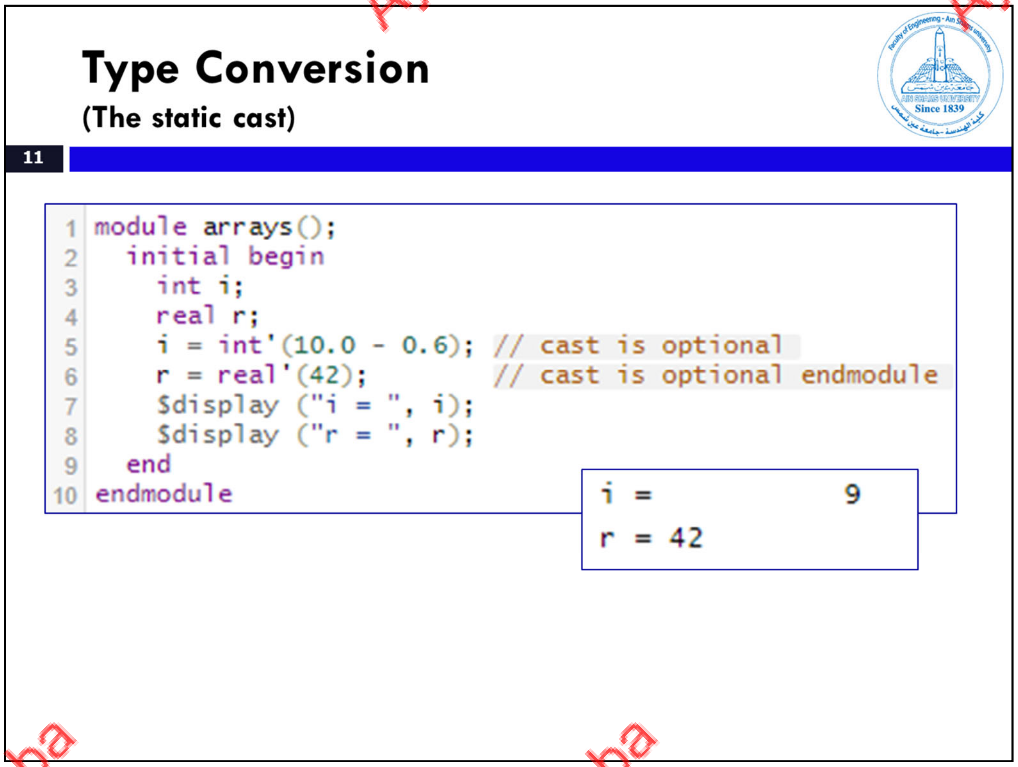
```
module arrays();
  typedef struct {bit [7:0] r, g, b;} pixel_s;
  pixel_s my_pixel;
endmodule
```

my_pixel is stored in 3 long words

```
module arrays();
  typedef struct packed {bit [7:0] r, g, b;} pixel_p_s;
  pixel_p_s my_pixel;
endmodule
```

my_pixel is stored in 3 bytes

- SystemVerilog allows you more control on how bits are laid out in memory by using packed structures.
- A packed structure is stored as a contiguous set of bits with no unused space.
- The struct for a pixel in the first code uses three values, and so it is stored in three longwords, even though it only needs three bytes.
- You can specify that it should be packed into the smallest possible space, as shown in the second code.
- Packed structures are used when the underlying bits represent a numerical value, or when you are trying to reduce memory usage. For example, you could pack together several bit-fields to make a single register. Or you might pack together the opcode and operand fields to make a value that contains an entire processor instruction.
- When you are trying to choose between packed and unpacked structures, consider how the structure is most commonly used, and the alignment of the elements.
  - ➢ If you plan on making aggregate operations on the structure, such as copying the entire structure, a packed structure is more efficient.
  - ➢ However, if your code accesses the individual members more than the entire structure, use an unpacked structure.



Type Conversion

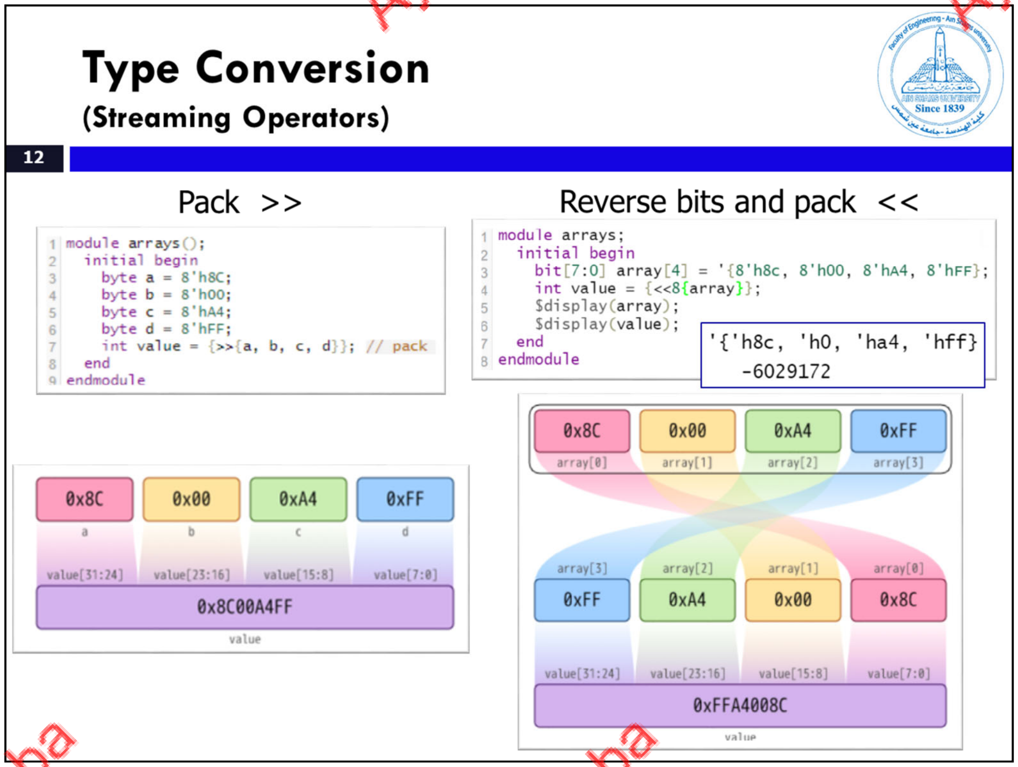
(The static cast)

11

```
module arrays();
  initial begin
    int i;
    real r;
    i = int'(10.0 - 0.6); // cast is optional
    r = real'(42);        // cast is optional endmodule
    $display ("i = ", i);
    $display ("r = ", r);
  end
endmodule
```

```
i =           9
r = 42
```

- The static cast operation converts between two types *with no checking of values*. You specify the destination type, an apostrophe, and the expression to be converted as shown.
- Note that verilog has always implicitly converted between types such as integer and real, and also between different width vectors.



Type Conversion
(Streaming Operators)

12

Pack >>

```
module arrays();
  initial begin
    byte a = 8'h8C;
    byte b = 8'h00;
    byte c = 8'hA4;
    byte d = 8'hFF;
    int value = {>>{a, b, c, d}}; // pack
  end
endmodule
```

Reverse bits and pack <<

```
module arrays;
  initial begin
    bit[7:0] array[4] = '{8'h8c, 8'h00, 8'hA4, 8'hFF};
    int value = {<<8{array}};
    $display(array);
    $display(value);
  end
endmodule
```

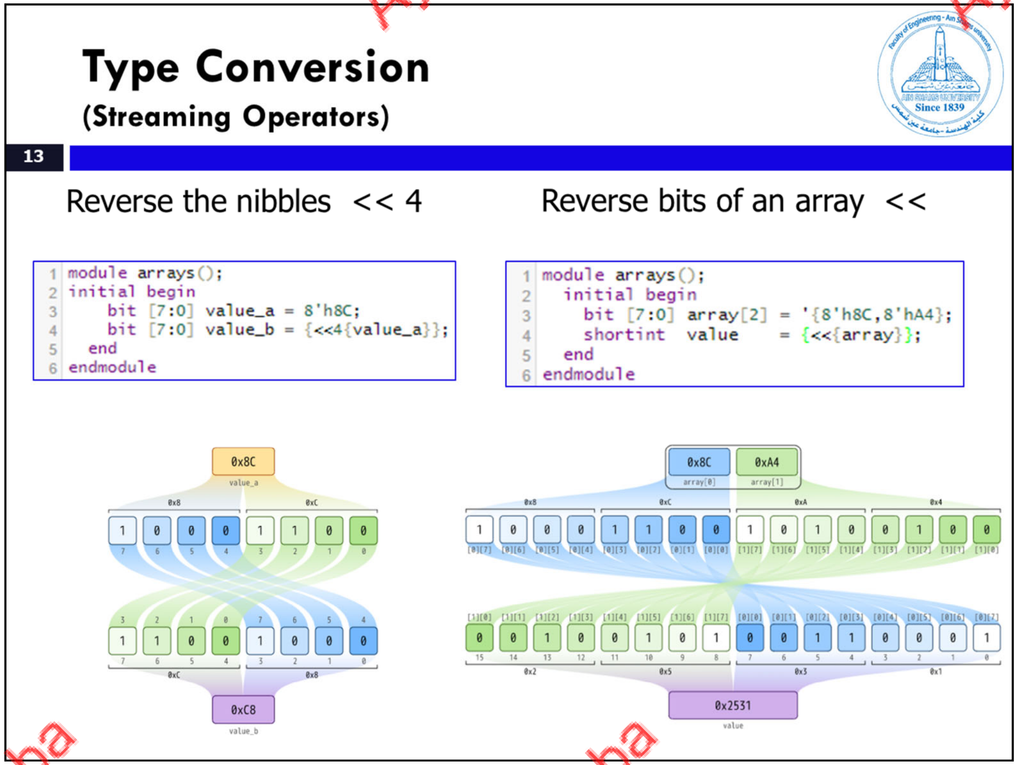
```
'{'h8c, 'h0, 'ha4, 'hff}
 -6029172
```

- The streaming operators perform packing of bit-stream into a sequence of bits in a user-specified order.
- The stream_operator << or >> determines the order in which blocks of data are streamed:
  - "\>>" causes blocks of data to be streamed in left-to-right order
  - "<<" causes blocks of data to be streamed in right-to-left order.
- Right-to-left streaming using << shall reverse the order of blocks in the stream, preserving the order of bits within each block.
- For right-to-left streaming using <<, the stream is sliced into blocks with the specified number of bits, starting with the right-most bit. If as a result of slicing the last (left-most) block has fewer bits than the block size, the last block has the size of the remaining bits; there is no padding or truncation.
- Note that FFA4008C (after packing) is

  1111 1111 1010 0100 0000 0000 1000 1100

  When translated a a signed number, the result is:

  $-2^{31}+2^{30}+2^{29}+2^{28}+2^{27}+2^{26}+2^{25}+2^{24}+2^{23}+2^{21}+2^{18}+2^{7}+2^{3}+2^{2} = -6029172$



Type Conversion
(Streaming Operators)

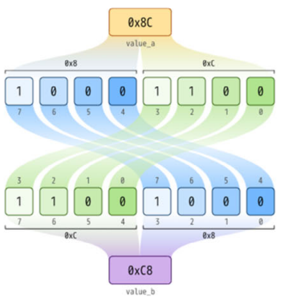
13

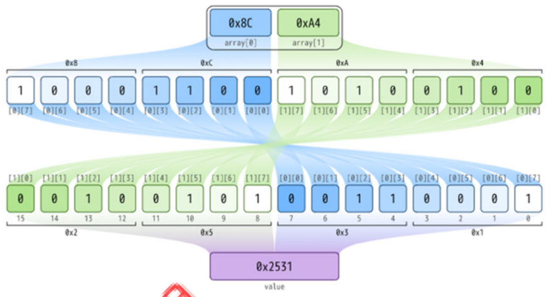
Reverse the nibbles << 4

```
module arrays();
initial begin
    bit [7:0] value_a = 8'h8C;
    bit [7:0] value_b = {<<4{value_a}};
  end
endmodule
```

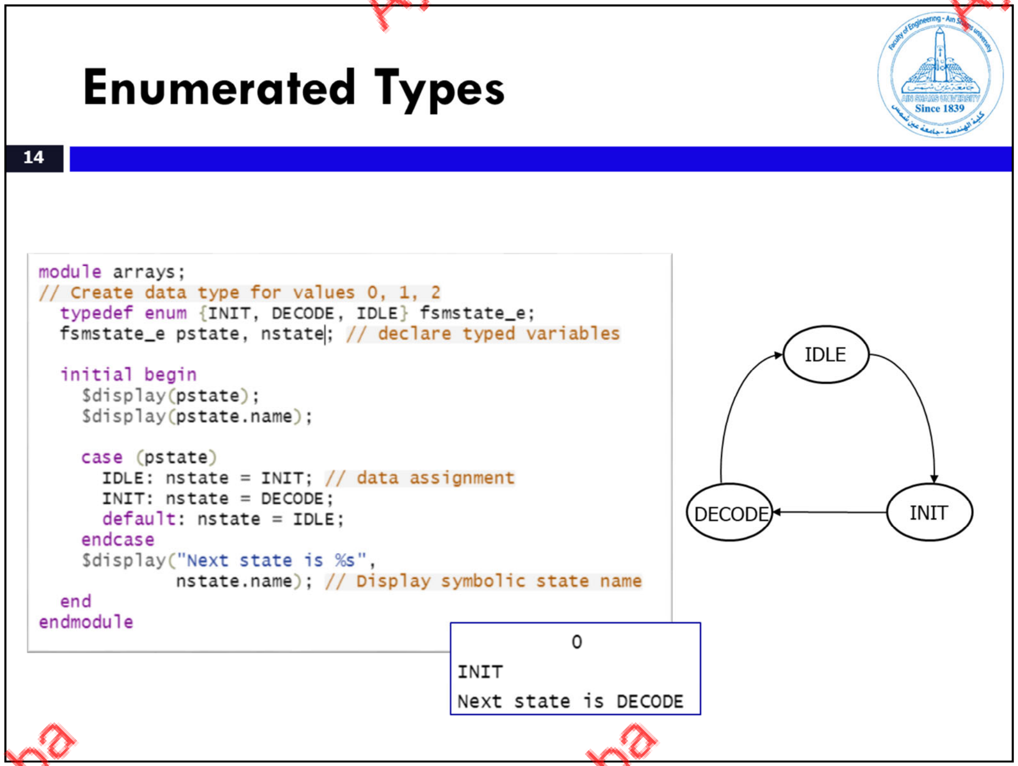
Reverse bits of an array <<

```
module arrays();
  initial begin
    bit [7:0] array[2] = '{8'h8C,8'hA4};
    shortint  value    = {<<{array}};
  end
endmodule
```





- The slice_size determines the size of each block, measured in bits. If a slice_size is not specified, the default is 1.
- Left-to-right streaming using >> shall cause the slice_size to be ignored and no re-ordering performed.

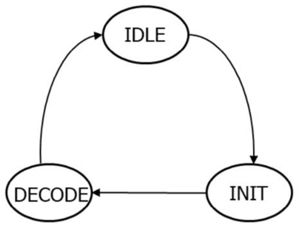


Enumerated Types

14

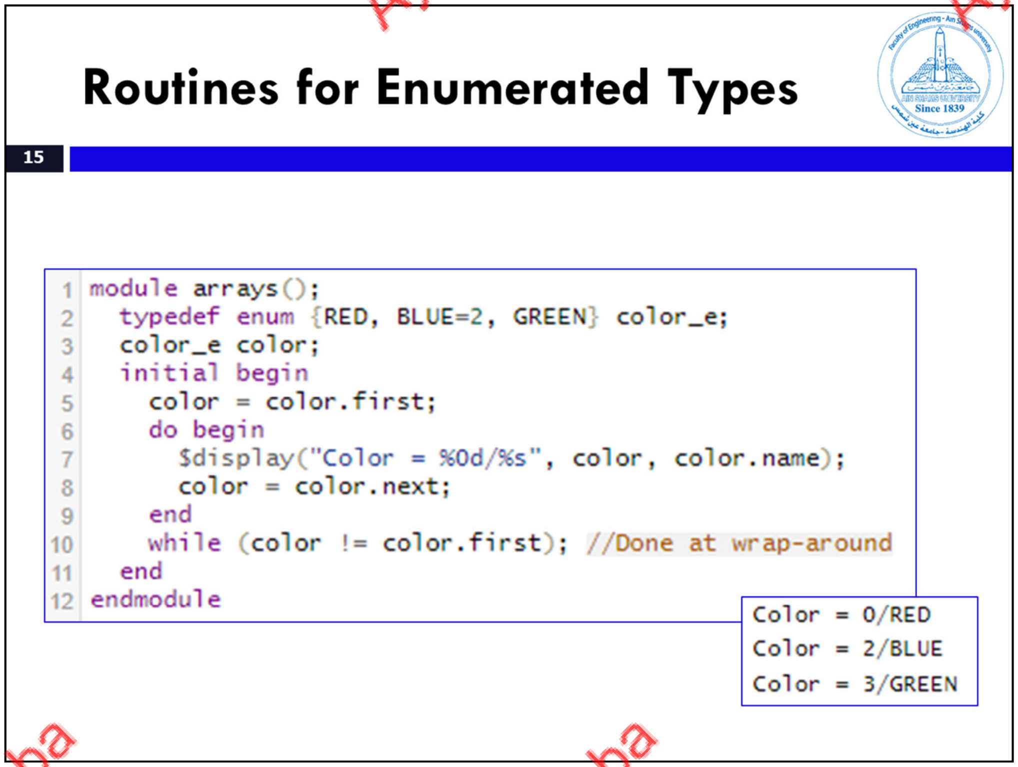
```
module arrays;
// Create data type for values 0, 1, 2
  typedef enum {INIT, DECODE, IDLE} fsmstate_e;
  fsmstate_e pstate, nstate; // declare typed variables

  initial begin
    $display(pstate);
    $display(pstate.name);

    case (pstate)
      IDLE: nstate = INIT; // data assignment
      INIT: nstate = DECODE;
      default: nstate = IDLE;
    endcase
    $display("Next state is %s",
             nstate.name); // Display symbolic state name
  end
endmodule
```

```
          0
INIT
Next state is DECODE
```



- An enumeration creates a variable type that is limited to a set of specified names, such as the instruction opcodes or state machine values. For example, the names ADD, MOVE, or ROTW make your code easier to write and maintain than if you had used literals such as 8’h01.
- The simplest enumerated type declaration contains a list of constant names and one or more variables: enum {RED, BLUE, GREEN} color;
- This creates an anonymous enumerated type, but it cannot be used for any other variables than the ones in this declaration.
- In general you want to create a named enumerated type to easily declare multiple variables, especially if these are used as routine arguments or module ports. You first create the enumerated type using “typedef”, and then declare the variables of this type.
- You can get the string representation of an enumerated variable with the built-in function name().
- Use the suffix "_e" when declaring an enumerated type to facilitate the readability.
- The actual values of the enumerated types are 0, 1, 2 … INIT, IDLE and DECODE are just nicknames for the 0, 1, 2. In the above code, as *pstate* is not initialized, it takes the default value of 0 (i.e INIT).
- However, you can choose your own enumerated values. You can, for example, use the default value of 0 for INIT, then 2 for DECODE, and 3 for IDLE.

  typedef enum {INIT, DECODE=2, IDLE} fsmtype_e;



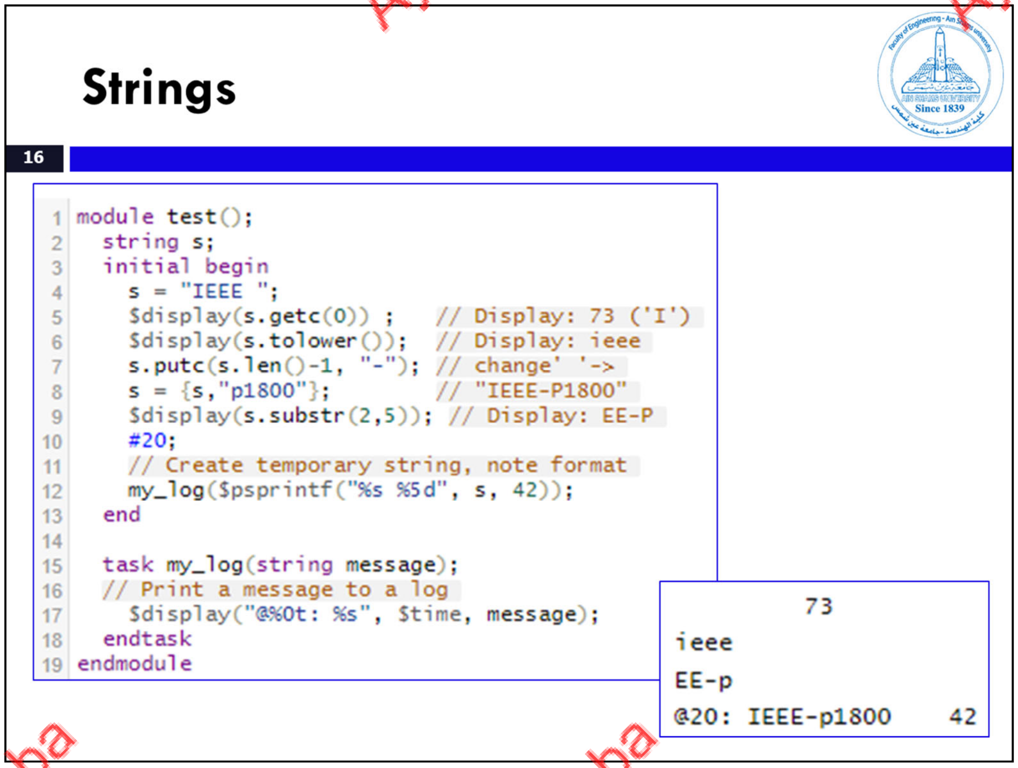
Routines for Enumerated Types

15

```
module arrays();
  typedef enum {RED, BLUE=2, GREEN} color_e;
  color_e color;
  initial begin
    color = color.first;
    do begin
      $display("Color = %0d/%s", color, color.name);
      color = color.next;
    end
    while (color != color.first); //Done at wrap-around
  end
endmodule
```

```
Color = 0/RED
Color = 2/BLUE
Color = 3/GREEN
```

- SystemVerilog provides several functions for stepping through enumerated types.
  - ➢ first(): returns the first member of the enumeration.
  - ➢ last(): returns the last member of the enumeration.
  - ➢ next(): returns the next element of the enumeration.
  - ➢ next (N): returns the N^th next element.
  - ➢ prev (): returns the previous element of the enumeration.
  - ➢ prev (N): returns the N^th previous element.
- The functions next and prev wrap around when they reach the beginning or end of the enumeration.
- Note that there is no easy way to write a for-loop that steps through all members of an enumerated type if you use an enumerated loop variable. The problem resides in how to stop the loop.



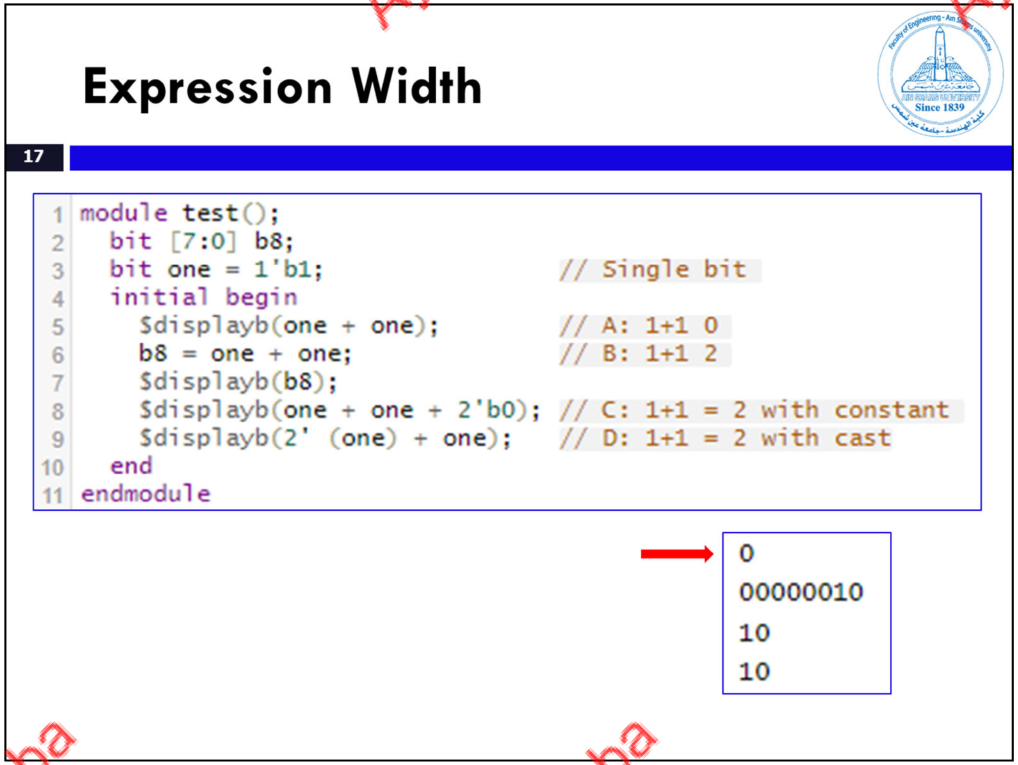
Strings

16

```
module test();
  string s;
  initial begin
    s = "IEEE ";
    $display(s.getc(0)) ;   // Display: 73 ('I')
    $display(s.tolower());  // Display: ieee
    s.putc(s.len()-1, "-"); // change' '->
    s = {s,"p1800"};        // "IEEE-P1800"
    $display(s.substr(2,5)); // Display: EE-P
    #20;
    // Create temporary string, note format
    my_log($psprintf("%s %5d", s, 42));
  end

  task my_log(string message);
  // Print a message to a log
    $display("@%0t: %s", $time, message);
  endtask
endmodule
```

```
          73
ieee
EE-p
@20: IEEE-p1800     42
```

- The SystemVerilog string type holds variable-length strings.
- An individual character is of type byte. The elements of a string of length N are numbered 0 to N-l.
- Note that, unlike C, there is no null character at the end of a string, and any attempt to use the character "\0" is ignored.
- Memory for strings is dynamically allocated, so you do not have to worry about running out of space to store the string.
- The above code shows various string operations.
  - ➢ The function getc(N) returns the byte at location N
  - ➢ The toupper returns an upper-case copy of the string
  - ➢ The tolower returns a lowercase copy.
  - ➢ The curly braces {} are used for concatenation.
  - ➢ The task putc(M, C) writes a byte C (i.e a character) into a string at location M, which must be between 0 and the length as given by len.
  - ➢ The substr(start, end) function extracts characters from location start to end.
- The $psprintf() function returns a formatted temporary string that, as shown above, can be passed directly to another routine. This saves you from having to declare a temporary string and passing it between the formatting statement and the routine call.



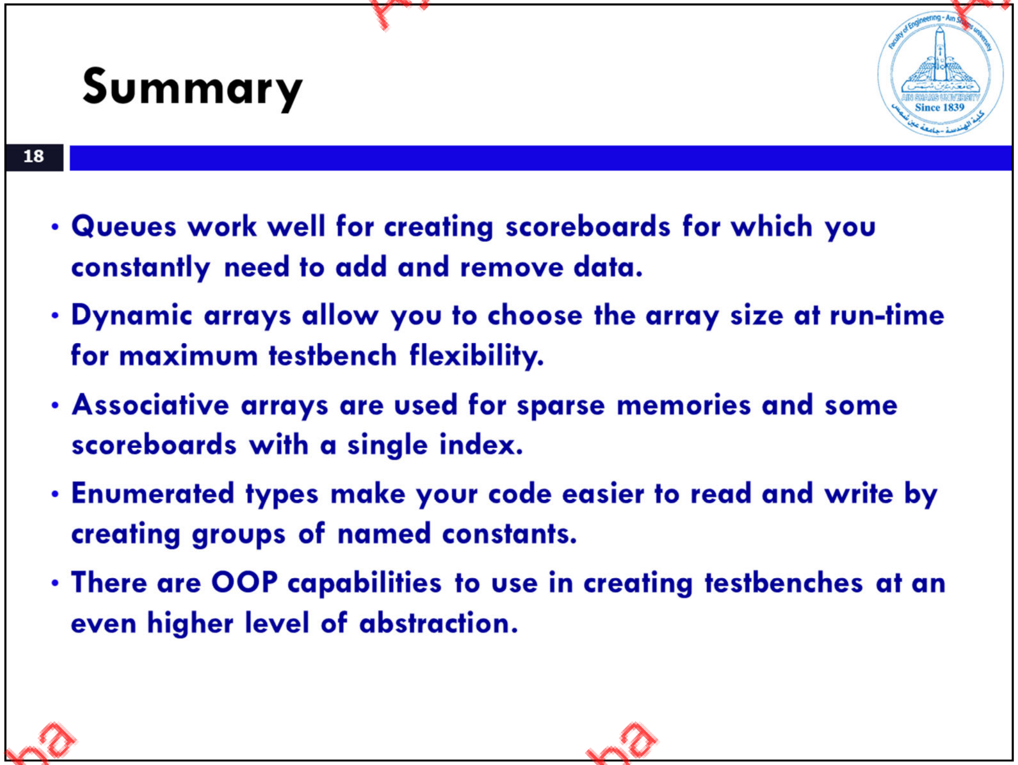
Expression Width

17

```
module test();
  bit [7:0] b8;
  bit one = 1'b1;                 // Single bit
  initial begin
    $displayb(one + one);         // A: 1+1 0
    b8 = one + one;               // B: 1+1 2
    $displayb(b8);
    $displayb(one + one + 2'b0); // C: 1+1 = 2 with constant
    $displayb(2' (one) + one);    // D: 1+1 = 2 with cast
  end
endmodule
```

```
0
00000010
10
10
```

- A prime source for unexpected behavior in Verilog is the width of expressions.
- The above code adds 1 + 1 using four different styles.
- Addition A uses two 1-bit variables, and so with this precision 1 + 1 = 0.
- Addition B uses 8-bit precision because there is an 8-bit variable on the left side of the assignment. In this case, 1 + 1 =2.
- Addition C uses a dummy constant to force SystemVerilog to use 2-bit precision.
- Lastly, in addition D. the first value is cast to be a 2-bit value with the cast operator, and so 1 + 1 =2.
- There are several tricks you can use to avoid this problem. First, avoid situations where the overflow is lost, as in addition A. Use a temporary, such as b8, with the desired width. Or, you can add another value to force the minimum precision, such as 2' b00 Lastly, in SystemVerilog, you can cast one of the variables to the desired precision.



Summary

18

- Queues work well for creating scoreboards for which you constantly need to add and remove data.
- Dynamic arrays allow you to choose the array size at run-time for maximum testbench flexibility.
- Associative arrays are used for sparse memories and some scoreboards with a single index.
- Enumerated types make your code easier to read and write by creating groups of named constants.
- There are OOP capabilities to use in creating testbenches at an even higher level of abstraction.

- SystemVerilog provides many data types and structures so that you can create high-level testbenches without having to worry about the bit-level representation.
- Queues work well for creating scoreboards for which you constantly need to add and remove data.
- Dynamic arrays allow you to choose the array size at run-time for maximum testbench flexibility.
- Associative arrays are used for sparse memories and some scoreboards with a single index.
- Enumerated types make your code easier to read and write by creating groups of named constants.
- Don't go off and create a procedural testbench with just these constructs. There are OOP capabilities of SystemVerilog that you can use in creating testbenches at an even higher level of abstraction, thus creating robust and reusable code.