Chapter 2 Data Types Topics

- Why SystemVerilog?
- The logic type
- 2-state logic
- Fixed size arrays
- for/foreach loops
- packed/unpacked arrays
- Dynamic arrays
- Queues
- Associative arrays
- Array Methods
- Choosing a storage type
- Structures
- User defined types
- Type conversion
- Streaming operators
- Enumerated types
- Strings
Why SystemVerilog?

• Gains in productivity/maintainability/thoroughness
  • Recall graphs of directed versus random testing.
• Automatic generation and direction of stimulus
• Functional Coverage
• Higher level RTL
• Required for complex designs
Current Version of Verilog/SystemVerilog

• Last Verilog standard is IEEE Std 1364-2005
• Verilog (1364-2005) and SystemVerilog (1800) merged to create IEEE Std 1800-2009
• Verilog RTL synthesis standard is IEEE 1364.1-2005
Tool support for SystemVerilog

• Cadence
  • Minimum is Incisive Design Team Simulator v6.1
  • Much better SystemVerilog support in v6.2
• Mentor
  • Need QuestaSim
  • ModelSim SE or PE does not fully support SystemVerilog
  • Minimum is QuestaSim 6.3a
  • Recommend QuestaSim 6.3d or higher
• Synopsys
  • VCS 2009.12
  • Recommend Design Compiler 2007.12-SP1 or higher
  • Synthesis of SystemVerilog supported in Xilinx ISE 12.1
  • Synplicity and Precision support is limited but improving.
2.1.1: The Logic Type

• Verilog has 2 data types
  • wire
  • reg
• SystemVerilog introduces the logic data type
  • Still 4-state (0, 1, X, and Z)
  • Replaces both wire and reg
    • Cannot have multiple drivers
    • For a bidirectional bus use the wire type.

```verbatim
logic reset;
logic [15:0] data;
```

Can initialize logic and reg upon declaration (used with FPGAs):

```verbatim
logic reset = 0;
logic [15:0] data = 16'hAB;
```
2.1.2: 2-State Data Types

• Why?
  • Faster simulation and less memory than 4-state logic
  • Easier to interface to C/C++ functions

  • bit byte shortint int longint

```c
bit [15:0] my_reg; // 2-state 16 bit unsigned
shortint my_reg2; // 2-state 16 bit signed
```

• Caveats
  • bit is not signed
  • byte, shortint, int, longint are signed
  • Initial value is 0
    • Can initialize to value upon declaration
  • X’s and Z’s resolve to a 0
  • bit [16] my_reg_error; // Illegal
2.1.2: 2-State Data Types (cont.)

- 4-state logic types are x at beginning of simulation
- 2-state logic types are 0 at beginning of simulation
- Can assign 4-state to 2-state.
- Can use $isunknown to check for x or z

```verbatim
if ($isunknown(iport) == 1)
  $display("@%0t: 4-state value detected on iport %b", $time, iport);
```

- Use to check DUT outputs
2-State Data Types Exercise

Given the following code sample:

```vhdl
byte my_byte;
integer my_integer = 32'b0000_1111_xxxx_zzzz;
int my_int = my_integer;
bite [15:0] my_bit = 16'h8000;
shortint my_short_int1 = my_bit;
shortint my_short_int2 = my_short_int1-1;
```

What is the decimal range of values my_byte can take?  **-128 to 127**

What is the value of:

- `my_integer`: 32'b0000_1111_xxxx_zzzz (4-state signed)
- `my_int` in hex: 32'h0000_0F00 (2-state signed)
- `my_bit` in decimal: 32768 (2-state unsigned)
- `my_short_int1` in decimal: -32768 (2-state signed)
- `my_short_int2` in decimal: 32767 (2-state signed)
2.2 Fixed-Size Arrays

• C-style array declaration

```c
int lo_hi[0:15];  // 16 ints [0]..[15]
int c_style_lo_hi[16];  // 16 ints [0]..[15]
bit [7:0] my_bitmem [128];
logic [3:0] my_logicmem [64];
logic [4] my_logicmem [64];  // Incorrect
```

• Out of bounds writes ignored
• Out of bounds read:
  • 4-state returns an X
  • 2-state returns a 0
2.2.2 The Array Literal

• Verilog 2001
  • Could not initialize arrays at declaration
  • No way to set default value of array

• SystemVerilog
  • Initialize with apostrophe (‘)

    ```
    int ascend[4] = '{0,1,2,3};
    ascend[0:2]='{5,6,7};
    ```

  • Replicate values with replication operator

    ```
    ascend = '{4{8}};
    ```

  • Specify defaults

    ```
    ascend = '{default:-1};
    ```

  • Print using %p format specifier

    ```
    $display("%p", ascend);
    ```
Array Exercise

Given the following code sample:

```verilog
logic [3:0] my_logicmem [4] = '{0,1,2,3};
logic [3:0] my_logic = 4'hF;
my_mem [2] = my_logicmem[4]; '{A5, A5, 0}
my_logic = my_logicmem[4]; x
my_logicmem[3] = my_mem[3]; '{0, 1, 2, 0}
my_mem[3] = my_logic; '{A5, A5, 0}
my_logic = my_logicmem[1]; 1
my_logic = my_mem[1]; 5
my_logic = my_logicmem[my_logicmem[4]]; x
```
2.2.3 Basic Array Ops – for/foreach

• For loop
  • Declare index local to loop
  • use $size function to return size of the array
    
    ```
    initial begin
    bit [31:0] src[5];
    for (int i=0; i<$size(src); i++)
      src[i]=i;
    end
    ```

• Foreach loop
  • Simply loops through each item of array
  • Index variable is automatically declared
    
    ```
    initial begin
    bit [31:0] dst[5];
    foreach(dst[j])
      dst[j]=j*2;
    end
    ```
2.2.3 Multi-dimensional arrays

• Initialization:

```c
int md[2][3] = {{{0,1,2}, {3,4,5}};}
```

• Foreach loop
  • Loop through every element
    ```c
    foreach (md[i,j]) $display("%d ", md[i][j]);
    ```

  • Loop through the first dimension
    ```c
    foreach (md[i]) $display("%d ", md[i][0]);
    ```

  • Loop through the 2\textsuperscript{nd} dimension
    ```c
    foreach (md[,j]) $display("%d ", md[1][j]);
    ```
2.2.4 Basic array ops – Copy and Compare

• Comparison and copy only
• No arithmetic – use loops

```verbatim
initial begin
  bit [31:0] src[5] = '{0,1,2,3,4},
      dst[5] = '{5,4,3,2,1};
  if(src==dst)
    $display("src==dst");
  else
    $display("src!=dst");
  dst=src;
  src[0]=5;
  if(src[1:4]==dst[1:4])
    $display("src==dst");
  else
    $display("src!=dst");
end
```
2.2.5 Bit and Array Subscripts

- Illegal in Verilog-1995 to select a bit from an element in an array

```verilog
brit [31:0] src[5] = '{5{5}};
$displayb(src[0],, src[0][0],, src[2][2:1]);
```

```
# 00000000000000000000000000000101 1 10
```
Array Operations Exercise

Write the SystemVerilog code to:

1) Declare a 2-state array, `my_array`, that holds four 12-bit values

2) initialize `my_array` so that:
   1) `my_array[0] = 12'h012`  \(12'b0000_0001_0010\)
   2) `my_array[1] = 12'h345`  \(12'b0011_0100_0101\)
   3) `my_array[2] = 12'h678`  \(12'b0110_0111_1000\)
   4) `my_array[3] = 12'h9AB`  \(12'b1001_1010_1011\)

3) Traverse `my_array` and print out bits [5:4] of each 12-bit element


   1) Using a for loop
      
      ```systemverilog
      for(int i=0; i<$size(my_array); i++)
      $display("%b", my_array[i][5:4]);
      ```

   2) Using a foreach loop
      
      ```systemverilog
      foreach(my_array[i]) $display("%b", my_array[i][5:4]);
      ```
2.2.1 Fixed size arrays - unpacked arrays

- Unpacked arrays
  - **Store** `byte, shortint, int` in a single 32-bit word
  - Identified by the declaration. Depth is declared *after* the variable

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

- Can consider `up_array` only as an array of bytes.

```
$displayh(up_array);
```

- Cannot trigger an event on `up_array`

```
always @(up_array);
wait(up_array);
```

Total memory consumed = 3*32 = 96 bits
2.2.6 Packed Arrays

• Stored as contiguous bits
• Identified by the declaration. Size is declared *before* the variable

```
bit [3:0][7:0]bytes = 32’hcafe_dada;
```

| ca | fe | da | da |

Total memory consumed = 4*8 = 32 bits

• Array bytes can be considered as a 32-bit word or 4 bytes

```
$displayh(bytes,, bytes[3],, bytes[3][7]);
# cafedada ca 1
```

• However, cannot fully consider array bytes as a 32-bit word

```
$displayb(bytes[31]);
```

• Can trigger an event on bytes

```
always @(bytes);
wait(bytes);
```
Packed Array Example

```verbatim
bit [1:0] [2:0] [3:0] barray;
barray = '{'4'h6, 4'h5, 4'h4}, '{4'h3, 4'h2, 4'h1}'};
$displayh(barray,, barray[1],, barray[1][2],, barray[1][2][3]);
```
Mixing Packed and Unpacked dimensions

- Dimensions of the array declared *before* the variable are packed
- Dimensions of the array declared *after* the variable are unpacked

```vhdl
bit [3:0] [7:0] array [3];
```

### Packed Example
```
array[0][3] = 7654;  
array[1][6] = 3210;
```

### Unpacked Example
```
array[0] = 7654;  
array[1] = 3210;  
array[2] = 7654;
```
Indexing Mixed Arrays

• Unpacked dimensions are referenced first from the left-most to the right-most dimension
• Packed dimensions are referenced second from the left-most dimension to the right-most dimension

```verilog
logic [3:0][7:0] mixed_array [0:7] [0:7] [0:7];
mixed_array [0] [1] [2] [3] [4] = 1'b1;
```
Packed and Unpacked Exercise

• Declare a 5 by 31 multi-dimensional unpacked array, my_array1. Each element of the unpacked array holds a 4-state value.
• Which of the following assignments are legal and not out-of-bounds?

```markdown
my_array1[4][30] = 1'b1;
my_array1[29][4] = 1'b1;
my_array1[3] = 31'b1;
```

• Declare a 5 by 31 multi-dimensional packed array, my_array2. Each element of the packed array holds a 2-state value.
• Which of the following assignments are legal and not out-of-bounds?

```markdown
my_array2[4][30] = 1'b1;
my_array2[29][4] = 1'b1;
my_array2[3] = 31'b1;
```

• Draw each array, filling in the values after assignment.
my_array1:
xxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx
xxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx
xxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx
000_0000_0000_0000_0000_0000_0000_0000_0001
1xx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx

my_array2:
000_0000_0000_0000_0000_0000_0000_0000_0000
000_0000_0000_0000_0000_0000_0000_0000_0000
000_0000_0000_0000_0000_0000_0000_0000_0000
000_0000_0000_0000_0000_0000_0000_0000_0000
000_0000_0000_0000_0000_0000_0000_0000_0001
100_0000_0000_0000_0000_0000_0000_0000_0000
2.3 Dynamic Arrays

• Fixed-size array size is set at compile time
• Dynamic array size is set at run-time – initially empty
• Can be allocated and resized during simulation
• Declared with empty subscripts [ ]
  
  ```
  int dyn[], d2[];
  ```

• `new[]` operator allocates space, passing the number of entries
  
  ```
  dyn = new[5];
  ```

• Can also pass name of an array to copy values
  
  ```
  d2=new[20](dyn);
  dyn=new[100];
  ```

• Mixing fixed/dynamic – only *unpacked* dimensions can be dynamic
  
  ```
  reg [7:0][3:0] b_reg_array[];
  reg [][3:0] b_reg_array2 [4:0];
  ```

• Cannot view dynamic arrays in Questa’s waveform viewer
2.3 Dynamic Arrays (cont.)

- Growing dynamic array:

  ```java
  dyn = `\{dyn, 5\};
  ```

  Adds element 5 to the array. The size is now 6

- Shrinking dynamic array:

  ```java
  dyn = dyn[1:3];
  ```

  Reduces array size to 3

- What happens below?

  ```java
  dyn = new[5];
  dyn = new[4](dyn);
  ```

  First 4 elements are copied to new array and size shrinks to 4
2.3 Dynamic Arrays (cont.)

• When is this useful?
  • Creating a random # of transactions using an array
  • You don’t feel like counting the # of elements you put in an array.

  ```
  bit [7:0] mask[] = '{8'h12, 8'h23, 8'h45};
  ```

• Built-in methods
  • size
    ```
    $size(mask);
    mask.size();
    ```
  • delete
    ```
    mask.delete();
    ```

• Assignment
  • Between fixed-size and dynamic elements if base type is the same
  • dynamic to fixed-size if current size is the same
  • fixed-size to dynamic if current size is the same: new[] is called.
2.4 Queues

- Provides easy sorting and searching
- Can add and remove elements from anywhere
- Can dynamically grow and shrink. No new[].
- Can copy contents of fixed or dynamic arrays to the queue

```plaintext
string coworkers[$];
reg [7:0] address[$];
reg [7:0] j;
coworkers = {“Willy”, “Rob”};
address = {8’h3F, 8’hA5};
coworkers.insert(1, “Holger”);  // {“Willy”, “Holger”, “Rob”};
address.push_back(8’h00);  // {8’h3F, 8’hA5, 8’h00};
j = address.pop_front();  // {8’hA5, 8’h00}, j = 8’h3F
j = address.size();  // j = 2
address.delete();
```
Queues Exercise

Given the following code, determine what will be displayed.

```vhdl
`default_nettype none
module test;
    string street[] = {"Tejon", "Bijou", "Boulder");
    initial begin
        $display("Street[0] = %s", street[0]);  Tejon
        street.insert(2, "Platte");             Platte
        $display("Street[2] = %s", street[2]);  Bijou
        street.push_front("St. Vrain");        Boulder
        $display("Street[2] = %s", street[2]);
        $display("pop_back = %s", street.pop_back);  Tejon
        $display("street.size = %d", street.size); 4
    end
endmodule // test
```
2.5 Associative Arrays

• Suppose a processor you are modeling has a 4GB memory space.
  • A fixed/packed/dynamic array will take up too much memory
  • Use an associative array to model sparse memories
    • Declared with a data type in square brackets, such as [int] where int is the index
  • Only allocates memory for an element when written to.

```
byte unsigned assoc[int], idx = 1;
initial begin
  do begin
    assoc[idx] = idx;
    idx = idx << 1;
  end
  while (idx != 0);
  foreach (assoc[i])
    $display("assoc[%h] = %h", i, assoc[i]);
end
```
2.5 Associate Arrays (cont.)

Examples of associative array declarations:

```c
int assoc1[*]; // Wildcard index – not recommended
int assoc2[string]; // String index
int assoc3[integer]; // Integer index
int assoc4[a_class]; // Class object index
```
2.5 Associate Arrays (cont.)

Suppose we want to keep track of the reset values, address, etc. of a bank of configuration registers.

```
int address[string]; // string is the array index

initial begin
    address["ADC_REG"] = 0;
    address["DAC_REG"] = 1;
    address["LCD_REG"] = 2;
    address["LED_REG"] = 3;
    address["I2C_REG"] = 4;
    $display("Address of LCD_REG is %0d", address["LCD_REG"]);
end
```

# Address of LCD_REG is 2
Associative Array Methods

From previous slide:

```plaintext
initial begin
    address["ADC_REG"] = 0;
    address["DAC_REG"] = 1;
    address["LCD_REG"] = 2;
    address["LED_REG"] = 3;
    address["I2C_REG"] = 4;
end

string s; // The index
$display("The # of array entries is %0d", address.num());
# The # of array entries is 5

if (address.exists("DAC_REG")) $display("address[DAC_REG] exists");
# address[DAC_REG] exists

if (address.first(s)) $display("First address = %0d", address[s]);
# First address = 0
s = ADC_REG
```
Associative Array Methods

From previous slide:

```verilog
initial begin
    address["ADC_REG"] = 0;
    address["DAC_REG"] = 1;
    address["LCD_REG"] = 2;
    address["LED_REG"] = 3;
    address["I2C_REG"] = 4;
end
```

```verilog
define (address.last(s)) $display("Last address = %0d", address[s]);
# Last address = 3
s = LED_REG

s="LCD_REG";
define (address.next(s)) $display("next address = %0d", address[s]);
# next address = 3

address.delete("DAC_REG");
$display("The # of array entries is %0d", address.num);
# The # of array entries is 4
```
Write code for the following problems.

a) Create memory using an associative array for a processor with a word width of 24 bits and an address space of $2^{20}$ words. Assume the PC starts at address 0 at reset. Program space starts at 0x400. The ISR is at the maximum address.

b) Fill the associated array with the following instructions:
24'hA50400; // Jump to location 0x400 for the main code
24'h123456; // Instruction 1 located at location 0x400
24'h789ABC; // Instruction 2 located at location 0x401
24'h0F1E2D; // ISR = Return from interrupt

c) Print out the elements and the number of elements in the array.
Associative Array Exercise (cont.)

`default_nettype none
module test;

    bit [23:0] assoc[int];

initial begin

    assoc[0] = 24'hA50400; // Jump to location 0x400 for the main code
    assoc[32'h400] = 24'h123456; // Instruction 1
    assoc[32'h401] = 24'h789ABC; // Instruction 2
    assoc[32'hF_FFFF] = 24'h0F1E2D; // Return from interrupt

    $display("The # of entries is %d", assoc.num);

    foreach (assoc[i]) begin
        $display("assoc[%h] = %h", i, assoc[i]);
    end

end

endmodule // test

Results:

# The # of entries is 4
# assoc[00000000] = a50400
# assoc[00000400] = 123456
# assoc[00000401] = 789abc
# assoc[000fffff] = 0f1e2d
2.6 Array Methods

- Operate on any *unpacked* array types
  - fixed size
  - dynamic
  - queue
  - associative
- **Reduction**: sum, product, and, or, xor
- **Locator**: find, find_index, find_first, find_first_index, find_last, find_last_index, min, max, unique, unique_index
- **Ordering**: reverse, sort, rsort, shuffle

*except associative*
2.6.1 Array Reduction Methods

- The result bit width is the same as the element bit widths
- Operate on single bit elements, get a single bit result
- Assigning to a variable of sufficient width doesn’t help.

```verbatim
bit one[10] = '{0,1,0,1,0,1,0,1,0,1};
int total;
$display("one.sum = %0d", one.sum); // # one.sum=1
total = one.sum;
$display("total = %0d", total); // # total = 1
```

- `product()` returns the product of all array elements
  ```verbatim
  $display("one.product = %0d", one.product); // # one.product=0
  ```

- `and`, `or`, `xor` returns the bitwise operation of all array elements
  ```verbatim
  byte byte_array[2] = '{8'h35, 8'hFA};
  $display("OR = %b", byte_array.or); // # OR = 11111111
  $display("AND = %b", byte_array.and); // # AND = 00110000
  $display("XOR = %b", byte_array.xor); // # XOR = 11001111
  ```
Picking a random element from an associative array

```vhdl
int idx, element, count;

element = $urandom_range(aa.size()-1);
foreach(aa[i]) begin
    if (count++ == element) begin
        idx = i;  // Save the associative array index
        break;  // and quit
    end
end

$display("element#%0d aa[%0d] = %0d", element, idx, aa[idx]);
```

Example return:
element#1 aa[5] = 2
2.6.2 Array Locator Methods

- Result returned is always a queue
- `min()` returns the minimum element
- `max()` returns the maximum element
- `unique()` returns queue of unique values, duplicates not included
- `unique_index()` returns the indexes of all elements with unique values

```plaintext
int f[6] = '{1,6,2,6,8,6}; // Fixed size array
$display("f.min = %p", f.min);
$display("f.max = %p", f.max);
$display("f.unique = %p", f.unique);
$display("f.unique_index = %p", f.unique_index);

# f.min = '{1}
# f.max = '{8}
# f.unique = '{1, 2, 6, 8} // No duplicate 6's
# f.unique_index = '{0, 2, 1, 4} // Only 1st 6
```
2.6.2 Array Locator Methods (cont.)

- Array locator methods find* all require a with clause.
- The with clause is used to identify what you want to find.
- find() returns all the elements satisfying the expression.
- find_first() returns the first element satisfying the expression.
- find_last() returns the last element satisfying the expression.
- find*_index methods return the index satisfying the expression.

```plaintext
int f[6] = '1,6,2,6,8,6;
$display("f.find w/ (item >3) = %p", f.find with (item > 3));
$display("f.find_index w/ (item >3) = %p", f.find_index with (item > 3));
$display("f.find_first_index w/ (item >3) =%p", f.find_first_index with (item >3));
$display("f.find_last_index w/ (item >3) = %p", f.find_last_index with (item>3));

# f.find w/ (item >3) = '{6, 6, 8, 6}
# f.find_index w/ (item >3) = '{1, 3, 4, 5}
# f.find_first_index w/ (item >3) = '{1}
# f.find_last_index w/ (item >3) = '{5}
```
Creating the sum of an array of single bits

```plaintext
bit one[10] = '{0,1,0,1,0,1,0,1,0,1};
int total;
initial begin
    // Compute the single-bit sum
    total = one.sum(); // total = 1
    // Compute with 32-bit signed arithmetic
    total = one.sum() with (int'(item)); // total = 5
end
```
2.6.3 Array Sorting and Ordering

- Modifies the original array
- reverse() reverses the elements of the array
- sort() sorts the array in ascending order
- rsort() sorts the array in descending order
- shuffle() randomizes the order of the elements in the array.
- with clause can be used with sort or rsort

```plaintext
int f[6] = '{1,6,2,6,8,6};
f.reverse;
$display("f = %p", f);
f.sort;
$display("f = %p", f);
f.rsort;
$display("f = %p", f);
f.shuffle;
$display("f = %p", f);
```

```
# f = '{6, 8, 6, 2, 6, 1}
# f = '{1, 2, 6, 6, 6, 8}
# f = '{6, 6, 8, 1, 2, 6}
# f = '{8, 6, 6, 6, 2, 1}
# f = '{6, 6, 8, 1, 2, 6}
```
Array Method Exercise

Create the SystemVerilog code for the following requirements

a) Create a 3-byte queue and initialize it with 2, -1, and 127
b) Print out the sum of the queue in decimal
c) Print out the min and max values in the queue
d) Sort all values in the queue and print out the resulting queue
e) Print out the index of any negative values in the queue
f) Print out the positive values in the queue
g) Reverse sort all values in the queue and print out the resulting queue
Array Method Exercise (cont.)

a) Create a 3-byte queue and initialize it with 2, -1, and 127
   
   ```
   byte myQ[] = {2, -1, 127};
   ```

b) Print out the sum of the queue in decimal
   
   ```
   $display("Sum of myQ is: %0d", myQ.sum() with (int'(item))); // 128
   ```

a) Print out the min and max values in the queue
   
   ```
   $display("Min of myQ is: %p", myQ.min); // '{-1}
   $display("Max of myQ is: %p", myQ.max); // '[127}
   ```

b) Sort all values in the queue and print out the resulting queue
   
   ```
   myQ.sort();
   $display("myQ after sort: %p", myQ); // '{-1, 2, 127}
   ```

c) Print out the index of any negative values in the queue
   
   ```
   $display("negative values : %p", (myQ.find with (item <0))); // '{-1}
   ```

d) Print out the positive values in the queue
   
   ```
   $display("positive values : %p", (myQ.find with (item >=0))); // '{2, 127}
   ```

e) Reverse sort all values in the queue and print out the resulting queue
   
   ```
   myQ.rsort();
   $display("myQ after reverse sort: %p", myQ); // '{127, 2, -1}
   ```
2.7 Choosing a Storage Type

• Based on flexibility
  • Fixed size is least flexible. Use if size is known at compile time.
  • Dynamic arrays of same type can be processed by 1 subroutine
    • Make sure element types match (int, string, etc.).
  • Queues of same type can be processed by 1 subroutine
    • Make sure element types match (int, string, etc.).

• Based on memory usage
  • Use 2-state elements when possible
  • Choose multiples of 32-bit data width
  • Used packed if data width is not multiple of 32-bits
  • Fixed size and dynamic arrays are most memory efficient
    • `new []` is an expensive operation however
  • Widely spaced indices: associative array
2.7 Choosing a Storage Type (cont.)

• Based on speed
  • Any element in fixed size/dynamic array has equal access time
  • Inserting/removing elements in queue depends on location
    • First and elements pushed/popped w/ little overhead.
    • Inserting/removing middle elements require shifting
  • Accessing associative arrays is slowest
    • Simulator must search for memory elelments.

• Based on Access
  • Values received all at once
    • Fixed or dynamic size
  • Values received sporadically
    • Add to head/tail of queue
  • Unique and noncontiguous values
    • Associative array
2.7 Choosing a Storage Type (cont.)

- Choosing the best data structure
  - Network packets
    - Fixed size or dynamic array
  - Scoreboard of expected values
    - Queue or possibly associative if inserting/deleting in middle
  - Modeling > 1M entry memories
    - 2-state packed associative
  - Translating opcode names to opcode values
    - Associative using command as string index
2.8 Creating new types with typedef

• Useful if you have a common bit width or type
  Example: Create a unsigned 2-state byte
  ```
typedef bit [7:0] ubyte_t;
```  

• Useful for avoiding bus width errors. For example, if all the busses in your system are 13-bits wide define it.
  ```
typedef logic [12:0] global_bus_t;
```  

• Usage:
  ```
ubyte_t my_data
global_bus_t my_bus;
```
2.9 Creating User-Defined Structures

• A structure is a collection of variables and/or constants that can be accessed separately or as a whole
• Why?
  • Great for enumerating system resources
  • Allows grouping of data

```vhdl
typedef struct {
    reg [7:0] data_in;
    reg [3:0] address;
} mem_bus; // data_in and address are members of mem_bus

mem_bus my_mem_bus = '{8'hA5, 4'hC};
initial begin
    my_mem_bus.data_in = 8'h5A;
    my_mem_bus.address = 4'h3;
end
```
Typedef and Structures Exercise

1. Define a user defined 7-bit type
2. Encapsulate the fields of the following packet in a structure using your new type

```
header  cmd  data  crc
0  6  7  13  14  20  21  27
```

3. Assign the header to 7’h5A;
typedef and Structures Exercise

1. Define a user defined 7-bit type
   
   typedef bit [6:0] bit7_t;

2. Encapsulate the fields of the following packet in a structure using your new type
   
   typedef struct{
   bit7_t header;
   bit7_t cmd;
   bit7_t data;
   bit7_t crc;
   } packet;

1. Assign the header to 7’h5A;
   
   packet my_packet;

   initial begin
   my_packet.header = 7'h5A;
   end
2.9.4 Packed Structures

• Unpacked:
  • struct {bit[7:0] r;
    bit[7:0] g;
    bit[7:0] b;
  } u_pixel;

• Packed:
  • Struct packed {
    bit[7:0] r
    bit[7:0] g:
    bit[7:0] b;
  } p_pixel;
2.10 Packages

• Where to put all this stuff? In a package!

• Packages reduce the need for `include
  • May contain types, variables, tasks, functions, sequences, and properties

```verilog
package ABC;
  parameter int abc_data_width = 32;
  typedef logic [abc_data_width-1:0] abc_data_t;
  parameter time timeout = 100ns;
  string message = "ABC done";
endpackage // ABC
```
Importing Packages

• Recall that package ABC defined \texttt{abc\_data\_width}, \texttt{abc\_data\_t}, \texttt{timeout}, and \texttt{message}.

```verilog
module test;
  import ABC::*;
  abc_data_t data;
  string message = "Test timed out";
  initial begin
    #(timeout);
    $display("Timeout - %s", message);
    $finish;
  end
endmodule
```

• Use \texttt{ABC::message} to use the message variable in package ABC.
2.11 Type Conversion

Static cast converts between 2 types  No bounds checking

```
int i;
real r;
byte b;

initial begin
    i=int'(10.0-0.1);
    $display("i = 0d%0d", i);
    r=real'(42);
    $display("r = %f", r);
    b=byte'(256);
    $display("b = 0d%0d", b);
end
```

The following Verilog functions are still supported: $itor(), $rtoi(), $bitstoreal(), $realtobits(), $signed, $unsigned
2.12 Streaming Operators – unpacked to packed

• `>>` right of the assignment converts unpacked arrays into packed from left to right
• `<<` right of the assignment converts unpacked arrays into packed from right to left

```verilog
type h;
bit [7:0] j[4] = '{8'hA, 8'hB, 8'hC, 8'hD};
$display("j= %p", j);  // j= '{10, 11, 12, 13}
h=j;  // Cannot assign an unpacked type to a packed type
h={ >> {j}};
$display("h= 0x%0h", h);  // h= 0x0a0b0c0d
h={ << {j}};
$display("h= 0x%0h", h);  // i= 0xb030d050
h={ << byte {j}};
$display("h= 0x%0h", h);  // h= 0x0d0c0b0a
```
2.12 Streaming Operators – packed to unpacked

• >> left of the assignment converts packed arrays into unpacked from left to right
• << left of the assignment converts packed arrays into unpacked from right to left

```vhdl
int k=32'h89ABCDEF;
byte b1,b2,b3,b4;
b1=k[0];
$display("b1= 0x%0h", b1);
{>>{b1,b2,b3,b4}}=k;
$display("b1= 0x%0h", b1);
{<<{b1,b2,b3,b4}}=k;
$display("b1= 0x%0h", b1);
```

# b1 = 0x01
# b1 = 0x89
# b1 = 0xF7
• logic data[8] is short for data[0:7] not data[7:0]

```verilog
l logic data[8] = '{1,1,1,1,0,0,0,0};
l logic [7:0] data_new;
initial begin
    $display("data = %p", data); // data = '{1, 1, 1, 1, 0, 0, 0, 0}
    data_new = {>>{data}};
    $display("data_new = %b", data_new); // data_new = 11110000
end
```
Type Conversion and Streaming Exercise

1. Create a user defined type, \texttt{nibble	extunderscore t}, of 4 bits
2. Create a real variable, \texttt{r}, and initialize it to 4.33
3. Create a short int variable, \texttt{i	extunderscore pack}
4. Create an unpacked array, \texttt{k}, containing 4 elements of your user defined type nibble and initialize it to 4’h0, 4’hF, 4’hE, and 4’hD
5. Print out \texttt{k}
6. Stream \texttt{k} into \texttt{i	extunderscore pack} right to left on a bit basis and print out \texttt{i	extunderscore pack}
7. Stream \texttt{k} into \texttt{i	extunderscore pack} right to left on a nibble basis and print out \texttt{i	extunderscore pack}
8. Type convert real \texttt{r} into a nibble, assign it to \texttt{k[0]}, and print out \texttt{k}
Type Conversion and Streaming Exercise

5. Print out $k$
   
   ```
   $display("k=%p", k);
   ```

6. Stream $k$ into $i\_pack$ right to left on a bit basis and print out
   ```
   i\_pack
   i\_pack = { << \{k\} };
   $display("i\_pack=%0h", i\_pack);
   ```

7. Stream $k$ into $i\_pack$ right to left on a nibble basis and print out
   ```
   i\_pack
   i\_pack = { << nibble\_t \{k\} };
   $display("i\_pack=%0h", i\_pack);
   ```

8. Type convert real $r$ into a nibble, assign it to $k[0]$, and print out $k$
   ```
   k[0] = nibble\_t'(r);
   $display("k=%p", k);
   ```
Type Conversion and Streaming Exercise

1. Create a user defined type, \texttt{nibble	extunderscore t}, of 4 bits
   typedef bit [3:0] nibble	extunderscore t;

1. Create a real variable, \texttt{r}, and initialize it to 4.33
   real r = 4.33;

2. Create a short int variable, \texttt{i	extunderscore pack}
   shortint i	extunderscore pack;

1. Create an unpacked array, \texttt{k}, containing 4 elements of your user
defined type nibble and initialize it to 4’h0, 4’hF, 4’hE, and 4’hD
   nibble	extunderscore t k[4] = '{4'h0, 4'hf, 4'he, 4'hd};

2. Print out \texttt{k}

3. Stream \texttt{k} into \texttt{i	extunderscore pack} right to left on a bit basis and print out
   \texttt{i	extunderscore pack}

4. Stream \texttt{k} into \texttt{i	extunderscore pack} right to left on a nibble basis and print out
   \texttt{i	extunderscore pack}

5. Type convert real \texttt{r} into a nibble, assign it to \texttt{k[0]}, and print out \texttt{k}
2.13 Enumerations

• Create list of constant names

```plaintext
enum {CFG, ADC_REG, CTRL} reg_e;
    0   1   2
enum {CFG=5, ADC_REG=6, CTRL} reg2_e;
    5   6   7
```

• Easier than:

```plaintext
localparam CFG = 2'b00,
    ADC_REG = 2'b01,
    CTRL = 2'b10;
```

• Easier to add/delete registers

• Register name is not visible in waveform

• Usage

```plaintext
bit [1:0] my_reg;
my_reg = CFG;
```
2.13 Enumerated Types

• Creates user defined type of the enumeration
• Value of register is visible in waveform

```verbatim
typedef enum {CFG, ADC_REG, CTRL} reg_e;
reg_e my_reg;
initial begin
    my_reg = CFG;
    #30ns;
    my_reg = ADC_REG;
    #10ns;
    my_reg = CTRL;
    #10ns;
    my_reg = CFG;
end
```
typedef enum {ST0, ST1, ST2} state_e;
state_e state;
initial begin
    $display("first = 0x%0h", state.first);
    $display("next(1) = 0x%0h", state.next(1));
    $display("prev(1) = 0x%0h", state.prev(1));
    $display("num = 0x%0h", state.num);
    $display("name = %s", state.name);
    $display("last (hex) = 0x%0h", state.last);
    $display("last (string)=%s", state.last);
    $finish;
end
Stepping through all enumerated members

typedef enum {RED, BLUE, GREEN} color_e;
for (color_e color=color.first; color!=color.last; color= color.next)
    $display("Color = %0d/%s", color, color.name);

# Color = 0/RED
# Color = 1/BLUE

typedef enum {RED, BLUE, GREEN} color_e;
color_e color;
color = color.first;
do
    begin
        $display("Color = %0d/%s", color, color.name());
color = color.next;
    end
while (color != color.first);

# Color = 0/RED
# Color = 1/BLUE
# Color = 2/GREEN
Example of usage in testbench

```verilog
module test ();
    reg clk, reset;
    typedef enum {CFG, ADC_REG, CTRL} reg_e;
    reg_e  my_reg;
    initial begin
        clk = 'b0;
        forever #500 clk = ~clk;
    end
    initial begin
        reset = 1'b0;  #1000ns;
        @(negedge clk); reset = 1'b1;  my_reg = CFG;
        @(negedge clk);  my_reg =  my_reg.next;
        @(negedge clk);  my_reg =  my_reg.next;
        @(negedge clk);  my_reg =  my_reg.next;
    end
    config_reg config_reg(.address(my_reg));
endmodule
```
Waveform of usage in testbench
Enumerated Type Exercise

An ALU has the following opcodes.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>2'b00</td>
</tr>
<tr>
<td>sub</td>
<td>2'b01</td>
</tr>
<tr>
<td>bitwise invert A</td>
<td>2'b10</td>
</tr>
<tr>
<td>reduction OR B</td>
<td>2'b11</td>
</tr>
</tbody>
</table>

Write a testbench that performs the following tasks.
1. Create an enumerated type of the opcodes, opcode_e
2. Create a variable, opcode, of type opcode_e
3. Loop through all the values of variable opcode every 10ns
4. Instantiate an ALU with one 2-bit input opcode
Enumerated Type Exercise (cont.)

1. Create an enumerated type of the opcodes, `opcode_e`
   ```cpp
   typedef enum {Add, Sub, Not_A, ReductionOR_B} opcode_e;
   ```

2. Create a variable, `opcode`, of type `opcode_e`
   ```cpp
   opcode_e opcode;
   ```

3. Loop through all the values of variable `opcode` every 10ns
   ```cpp
   opcode = opcode.first();
   do
   begin
     $display("opcode = %s", opcode);
     opcode = opcode.next(1);
     #10ns;
   end
   while (opcode != opcode.first());

4. Instantiate an ALU with one 2-bit input `opcode`
   ```cpp
   alu alu(.opcode(opcode));
   ```
Enumerated Types in RTL

• Example:

```plaintext
typedef enum {ST0, ST1, ST2} state_e;
state_e current_state, next_state;
```

• Why? Easier than:

```plaintext
localparam ST0 = 2’b00, ST1 = 2’b01, ST2 = 2’b10;
reg [191:0] ASCII_current_state, ASCII_next_state;

  case (current_state)
    ST0: ASCII_current_state = "ST0";
    ST1: ASCII_current_state = "ST1";
    ST2: ASCII_current_state = "ST2";
  endcase
```
module enum_type (input wire clk, input wire reset);
    enum {ST0, ST1, ST2} state_e;
    reg [1:0] current_state, next_state;
    always @(posedge clk or negedge reset) begin
        if (!reset)
            current_state <= #1 ST0;
        else
            current_state <= #1 next_state;
    end
    always @* begin
        next_state = ST0;
        case (current_state)
            ST0: next_state = ST1;
            ST1: next_state = ST2;
            ST2: next_state = ST0;
        endcase
    end
endmodule
Simulation of State Machine
module enum_type (input wire clk, input wire reset);
    typedef enum {ST0, ST1, ST2} state_e;
    state_e current_state, next_state;
    always @(posedge clk or negedge reset) begin
        if (!reset)
            current_state <= #1 ST0;
        else
            current_state <= #1 next_state;
    end
    always @* begin
        next_state = ST0;
        case (current_state)
            ST0: next_state = ST1;
            ST1: next_state = ST2;
            ST2: next_state = ST0;
        endcase // case (current_state)
    end
endmodule
Simulation of State Machine
Base Type of Enumerated Types

Replace

```c
typedef enum {ST0, ST1, ST2} state_t;
```

with

```c
typedef enum reg [1:0] {ST0, ST1, ST2} state_t;
```

![Timing diagram](image-url)

 clk | ST0 | ST1 | ST2 | ST0 | ST1 | ST2
 current_state: | | | | | | |
 next_state: | ST1 | ST2 | ST0 | ST1 | ST2 | ST0
 reset: | | | | | | |
2.13.3 Converting to/from enum types

From enum type to non-enum type with simple assignment

```haskell
typedef enum {Add, Sub, Not_A, ReductionOR} opcode_e;
opcode_e opcode;
int i;
i = Not_A;
$display("i=%0d", i);
```

# i=2
2.12.3 Converting to/from enum ... (cont.)

From non-enum type to enum type requires static cast or $cast

```verilog
i=3;
if (!$cast(opcode,i))
  $display("Cast failed for i=%0d", i);
$display("opcode=%s", opcode);
i=4;
if (!$cast(opcode,i))
  $display("Cast failed for i=%0d", i);
$display("opcode=%s", opcode);
opcode = opcode_e'(i);
$display("opcode=%s", opcode);
```

# opcode=ReductionOR
# Cast failed for i=4
# opcode=ReductionOR
# opcode=4
2.15 Strings

Why?
- Much simpler string manipulation
- No more packing characters into a reg variable

```plaintext
string s;
s = "AMI Semiconductor";
$dbdisplay(s.toupper());
$dbdisplay(s.substr(3, 7));

s = {"ON ", s.substr(s.len()-13, s.len()-1)};
$dbdisplay(s);
s = {s, " 2008"};
$dbdisplay(s);
```

- Two ways to compare:
  s1 == s2
  - Returns 1 if identical,
  - Returns 0 if different
  s1.compare(s2)
  - Returns 1 if s1 greater than s2
  - Returns 0 if equal
  - Returns -1 if s1 less than s2