REAL-TIME DIGITAL SYSTEMS DESIGN AND VERIFICATION WITH FPGAS
ECE 387 – LECTURE 2

PROF. DAVID ZARETSKY

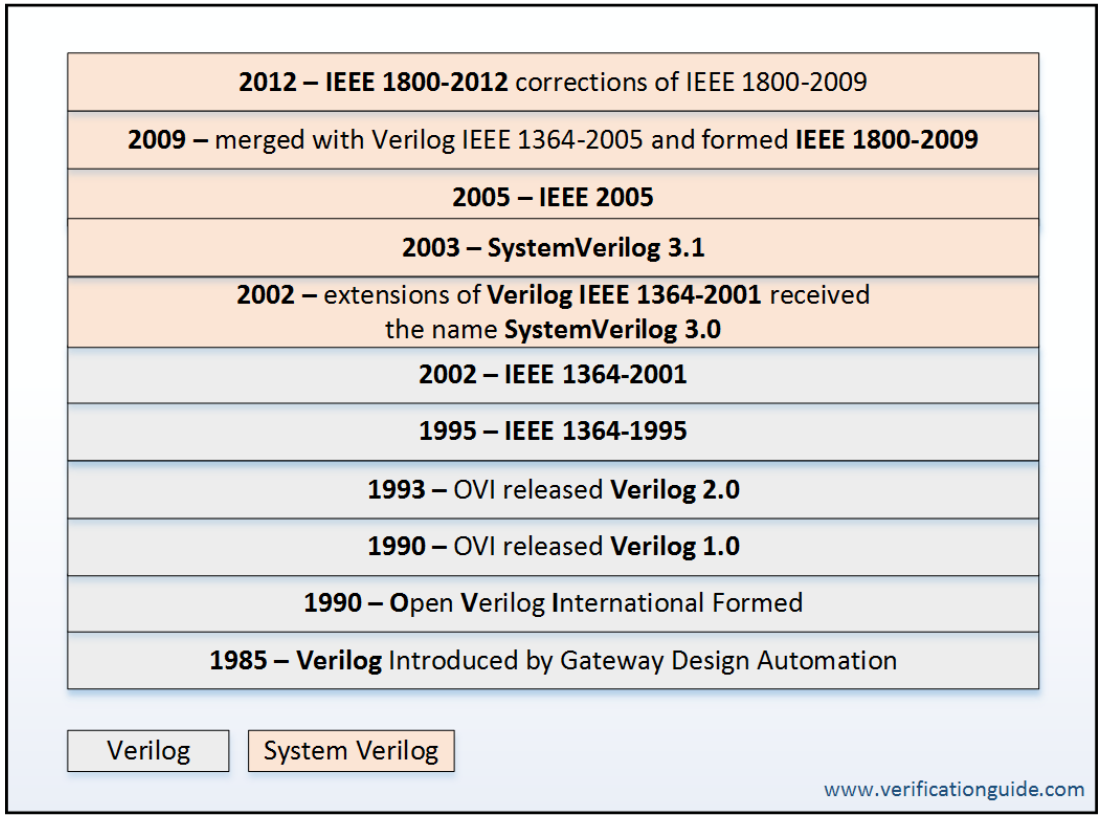
DAVID.ZARETSKY@NORTHWESTERN.EDU

AGENDA

- Introduction to SystemVerilog

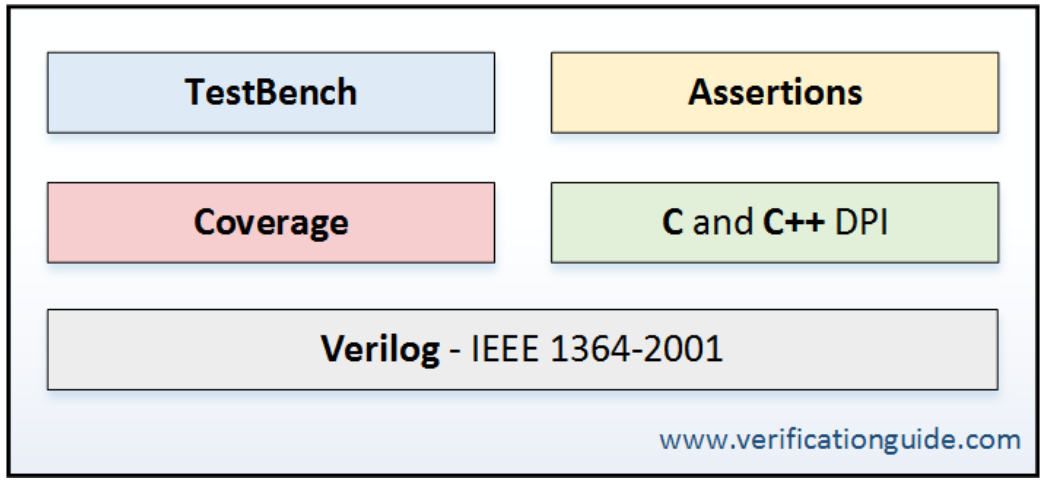
EVOLUTION OF SYSTEM VERILOG

- Gateway Design Automation introduced Verilog in 1985.
- In **1990**, Cadence placed the Verilog language in the public domain, and **O**pen **V**erilog **I**nternational (OVI) formed to manage the language.
- The IEEE working group released a revised standard in 2002, known as **IEEE 1364-2001** (also known as Verilog-2001).
- The new **extensions** to **Verilog** received the name **SystemVerilog 3.0**, which Accellera approved as a standard in **2002**.
- In **2005**, SystemVerilog was adopted as **IEEE** Standard **1800-2005**
- In **2009**, the standard was merged with the base Verilog (IEEE 1364-2005) standard, creating **IEEE** Standard **1800-2009**.
- The current SystemVerilog version is the IEEE standard **1800-2017**, can be divided into two distinct based on its roles:
  - SystemVerilog for **design** is an extension of Verilog-2005
  - SystemVerilog for **verification**



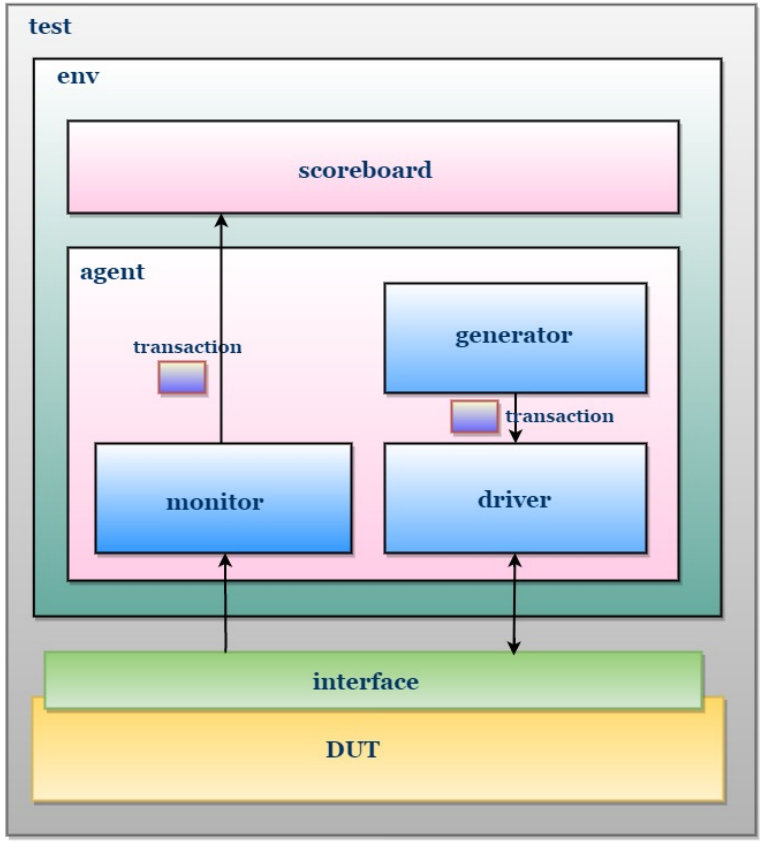
SYSTEM VERILOG COMPONENTS

- Verilog-2005 HDL
- Testbench (UVM)
- Assertions
- Direct Programming Interface (DPI) in C/C++
- Coverage API metrics



HOW DO WE VERIFY WITH SYSTEM VERILOG?

- General Steps:
  - Generate stimulus
  - Apply stimulus to the DUT
  - Capture the response
  - Check for the correctness
  - Measure progress against the overall verification goals
- We will see this in more detail later with the Universal Verification Methodology (UVM)



DATA TYPES

- **wire / reg**
  - Structural data types called nets, which model hardware connections between circuit components. These are now replaced with **logic**.
- **logic**
  - logic is the improved version of reg and wire. It can be driven by continuous assignments, gates, and modules in addition to being a variable.
- **integer**
  - An integer declares one or more variables of type integer. These variables can hold values ranging from -2^31 to (2^31)-1.
- **real**
  - The real variables are stored as 64-bit double-precision floating point values.
- **time**
  - *Time* is a 64-bit quantity that can be used in conjunction with the $time system task to hold simulation time.
- **parameters**
  - *Parameters* represent constants, hence it is illegal to modify their value at runtime. However, parameters can be modified at compilation time with the **defparam** statement, or in the module instance statement.
- **string**
  - A string data type is a dynamically allocated array of bytes.
  - Strings are wrapped in "quotes".
- **event**
  - An event is a handle to a synchronization object that can be passed to routines. Trigger function check whether an event has been triggered.
- **User-defined**
  - Define a new type using *typedef*, as in C.
- **Numbers**
  - Format: [precision]'[basetype][value]
  - Decimal: 32'd145
  - Hexadecimal: 8'hFA
  - Binary: 8'b10111011
  - Integer: 34
  - Real (float): 25.78e-9

LOGIC DATA TYPES

- Logic types may be used as input, output and local signals.
- May be assigned as concurrent assignments or sequential assignments.

SystemVerilog

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2,
                                y3, y4, y5);

  /* five different two input logic
     gates acting on 4 bit busses */
  assign y1 = a & b;     // AND
  assign y2 = a | b;     // OR
  assign y3 = a ^ b;     // XOR
  assign y4 = ~(a & b);  // NAND
  assign y5 = ~(a | b);  // NOR
endmodule
```

~, ^, and | are examples of SystemVerilog *operators*, whereas a, b, and y1 are *operands*. A combination of operators and operands, such as a & b, or ~(a | b), is called an *expression*. A complete command such as assign y4 = ~(a & b); is called

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity gates is
port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
     y1, y2, y3, y4,
     y5:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of gates is
begin
    five different two input logic gates
    acting on 4 bit busses
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```

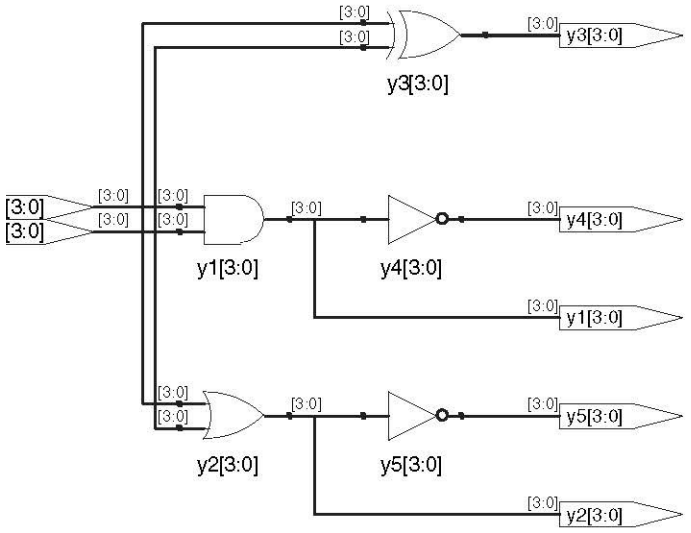


Figure 4.4 gates **synthesized circuit**

OPERATORS

Arithmetic Operators

Op	Description
a + b	a plus b
a - b	a minus b
a * b	a multiplied by b
a / b	a divided by b
a % b	a modulo b
a ** b	a to the power of b

Concatenation

Op	Description
{ }	Concatenation separated by commas
{# { }}	Replication by # times

Relational Operators

Op	Description
a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

Shift Operators

Op	Description
a << b	a shift left by b
a >> b	a shift right b
a <<< b	a arithmetic shift left by b
a >>> b	a arithmetic shift right by b

Equality Operators

Op	Description
a === b	a equal to b, including x and z
a !== b	a not equal to b, including x and z
a == b	a equal to b, result can be unknown
a != b	a not equal to b, result can be unknown

Logic Operators

Op	Description
a && b	evaluates to true if a *and* b are true
a || b	evaluates to true if a *or* b are true
!a	Converts non-zero value to zero

TIME

- Timescale directive specifies the time units and precision for simulations
- Timeformat function
  - $timeformat affects the format of time unit display
  - **$timeformat(unit_number, precision, suffix_string, minimum_field_width);**
  - unit_number is the smallest time precision out of all`timescale directives
  - precision represents the number of fractional digits for the current timescale
  - suffix_string is an option to display the scale alongside the real time values

Unit number	Time unit
-3	1ms
-6	1us
-9	1ns
-12	1ps
-15	1fs

```
`timescale 1ns/1ps
module tb;
  initial begin
    // Change timeformat parameters
    $timeformat(-9, 2, " ns");
    $display("[T=%0t] a=%0b", $realtime, a);
  end
  :
endmodule;
```

MATH FUNCTIONS

Function	Description
$ln(x)	Natural logarithm log(x)
$log10(x)	Decimal Logarithm log10(x)
$exp(x)	Exponential of x (e^x) where e=2.718281828...
$sqrt(x)	Square root of x
$pow(x, y)	x^y
$floor(x)	Floor x
$ceil(x)	Ceiling x
$sin(x)	Sine of x where x is in radians
$cos(x)	Cosine of x where x is in radians
$tan(x)	Tangent of x where x is in radians

Function	Description
$asin(x)	Arc-Sine of x
$acos(x)	Arc-Cosine of x
$atan(x)	Arc-tangent of x
$atan2(x, y)	Arc-tangent of x/y
$hypot(x, y)	Hypotenuse of x and y : sqrt(x^x + y^y)
$sinh(x)	Hyperbolic Sine of x
$cosh(x)	Hyperbolic-Cosine of x
$tanh(x)	Hyperbolic-Tangent of x
$asinh(x)	Arc-hyperbolic Sine of x
$acosh(x)	Arc-hyperbolic Cosine of x
$atanh(x)	Arc-hyperbolic tangent of x

ENUMERATION

- An enumerated type defines a set of named values.
- Enumerated type declaration contains a list of constant names
- You can also assign default values to names
  - `enum { red=10, green=20, blue=30, yellow=40 } Colors;`

```
module enum_datatype;
  //declaration
  typedef enum { red, green, blue, yellow } Colors;
  Colors C;

  //display members of Colors
  initial begin
    C = Colors.first();
    for(int i=0; i<4; i++ ) begin
       $display("Colors :: Value of %0s is = %0d",
              C.name(),C.num());
       C = C.next();
    end
  end
endmodule
```

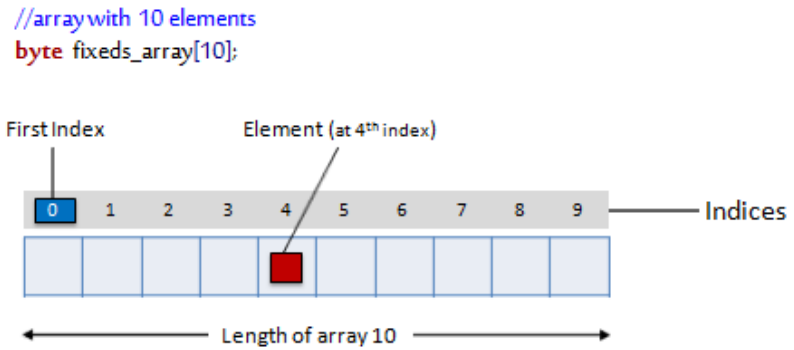
Output:

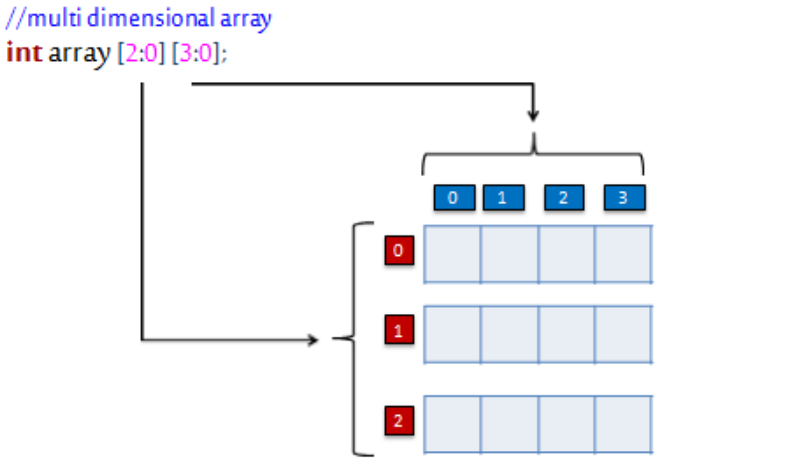
```
Colors :: Value of red is = 0
Colors :: Value of green is = 1
Colors :: Value of blue is = 2
Colors :: Value of yellow is = 3
```

Method	Description
first()	returns the value of the first member of the enumeration
last()	returns the value of the last member of the enumeration
next()	returns the value of next member of the enumeration
next(N)	returns the value of next Nth member of the enumeration
prev()	returns the value of previous member of the enumeration
prev(N)	returns the value of previous Nth member of the enumeration
num()	returns the number of elements in the given enumeration
name()	returns the string representation of the given enumeration value

ARRAYS

- An array is a collection of variables, all of the same type, accessible by one or more indices.
- Single dimensional array
  - `int arr [6];  //Compact declaration`
  - `int arr [5:0];  // Verbose declaration`
- Multidimensional array
  - `int arr [2:0][3:0]; // 2D array verbose declaration`
  - `int arr [2][3];  // 2D array with 2*3 = 6 elements.`
  - `int arr [2][2][2];  // 3D array with 2*2*2 = 8 elements.`
- Array assignment
  - `arr = '{0,1,2,3,4,5};`
  - `arr = '{'{0,1,2,3},'{4,5,6,7},'{8,9,10,11}};`





INFERRING SHIFT REGISTERS

- To infer shift registers, synthesis tools detect a group of shift registers of the same length and convert them to an shift register IP core.
- Use array assignment statement to shift register values
- To be detected, all the shift registers must have the following characteristics:
  - Use the same clock and clock enable
  - Do not have any other secondary signals
  - Have equally spaced taps that are at least three registers apart
- Synthesis recognizes shift registers only for device families that have dedicated RAM blocks, and the software uses certain guidelines to determine the best implementation.

```
module shift_8x64 (
  input logic clk,
  input logic shift,
  input logic [7:0] sr_in
  output logic [7:0] sr_out
);
  reg [63:0] [7:0] sr;
  always @ (posedge clk)
  begin
    if (shift == 1'b1)
    begin
       sr[63:1] <= sr[62:0];
       sr[0] <= sr_in;
    end
  end
  assign sr_out = sr[63];
endmodule
```

ARRAY FUNCTIONS

Array Reduction

Method	Description
sum()	returns the sum of all the array elements
product()	returns the product of all the array elements
and()	returns the bit-wise AND (&) of all the array elements
or()	returns the bit-wise OR (|) of all the array elements
xor()	returns the logical XOR (^) of all the array elements

```
x = '{1,2,3,4};
sum = x.sum(); // sum = 1+2+3+4
```

Array Ordering

Method	Description
reverse()	reverses all the elements of the array(packed or unpacked)
sort()	sorts the unpacked array in ascending order
rsort()	sorts the unpacked array in descending order
shuffle()	randomizes the order of the elements in the array

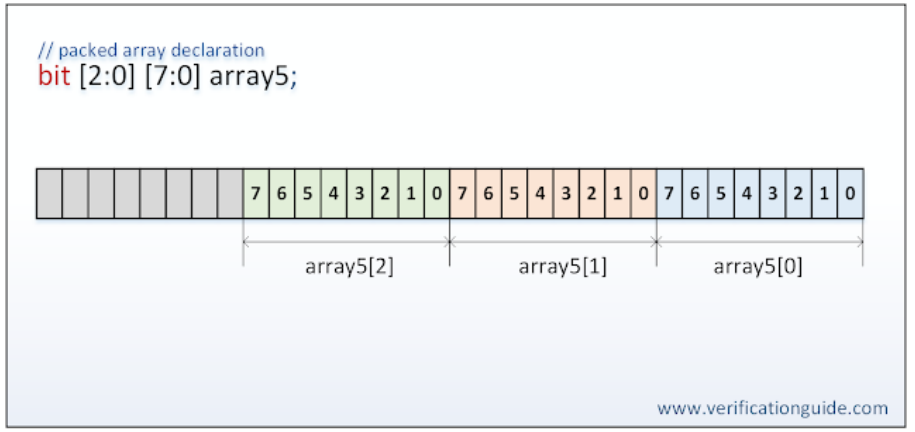
```
x = '{1,2,3,4};
x_r = x.reverse();// x_r={4,3,2,1}
```

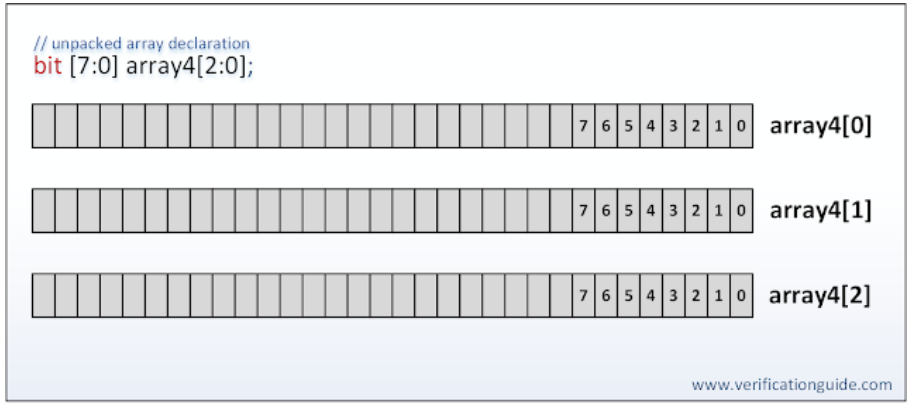
Array Search

Method	Description
find()	returns all elements satisfying the given expression
find_first()	returns the first element satisfying the given expression
find_last()	returns the last element satisfying the given expression
min()	returns the element with the minimum value
max()	returns the element with the maximum value
unique()	returns all elements with unique values
find_index()	returns the index of all elements satisfying the expression
find_first_index()	returns the first index satisfying the given expression
find_last_index()	returns the last index satisfying the given expression
unique_index()	returns all indices with unique values

PACKED AND UNPACKED ARRAYS

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





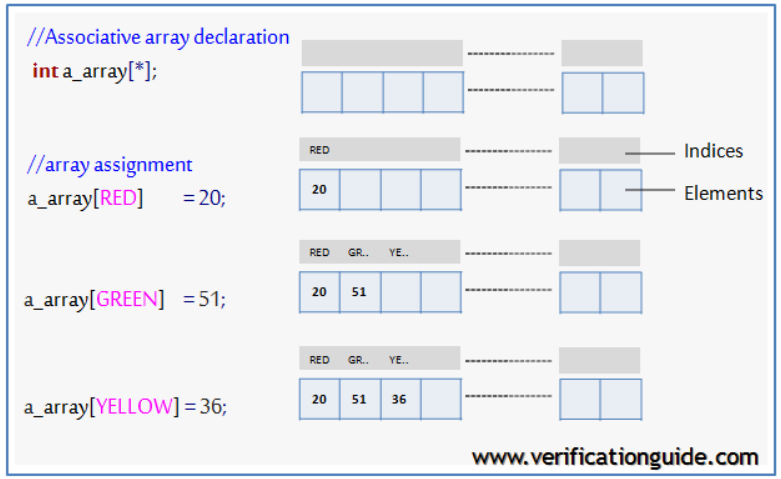
DYNAMIC ARRAYS

- A dynamic array is one dimension of an unpacked array whose size can be set or changed at run-time.
- Dynamic array is Declared using an empty word subscript [ ].
- Dynamic array methods
  - new[ ] : allocates the storage.
  - size( ) : returns the current size of a dynamic array.
  - delete( ) : empties the array, resulting in a zero-sized array.
- Not synthesizeable

```
// declaration
logic [7:0] d_array1[];
int d_array2[];

// memory allocation
d_array1 = new[4]; // dynamic array of 4 elements
d_array2 = new[6]; // dynamic array of 6 elements

// array initialization
d_array1 = {0,1,2,3};
foreach(d_array2[j]) d_array2[j] = j;
```

ASSOCIATIVE ARRAY

- An associative array implements a lookup table of the elements of its declared type.
- In associative array index expression is not restricted to integral expressions, but can be of any type
- The data type to be used as an index serves as the lookup key and imposes an ordering
- Associative array is a better option when the size of the collection is unknown or the data space is sparse
- Associative arrays allocate the storage only when it is used
- Not synthesizable

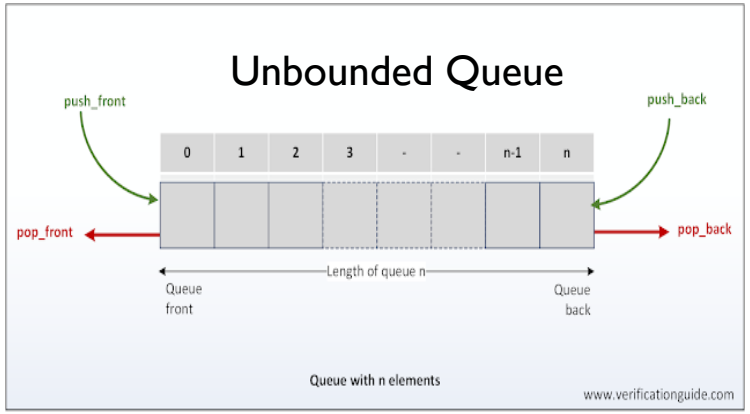


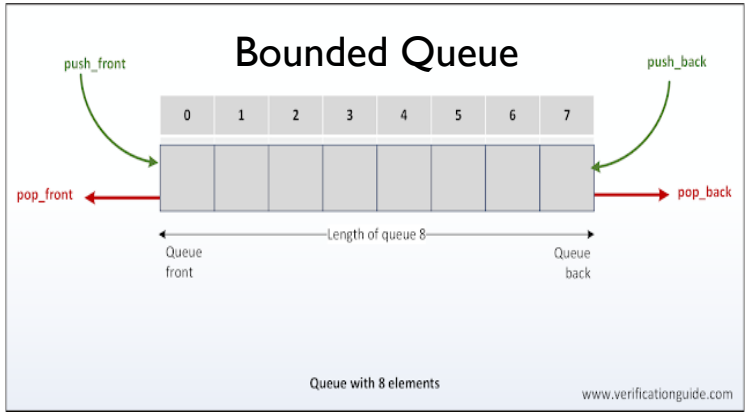
Method	Description
num()	returns the number of entries in the associative array
delete(index)	removes the entry at the specified index.exa_array.delete(index)
exists(index)	returns 1 if an element exists at the specified index else returns 0
first(var)	assigns the value of first index to the variable var
last(var)	assigns the value of last index to the variable var
next(var)	assigns the value of next index to the variable var
prev(var)	assigns the value of previous index to the variable var

QUEUES

- A queue is a variable-size, ordered collection of homogeneous elements.
  - like a dynamic array, queues can grow and shrink
  - queue supports adding / removing elements
- ***Queues*** are declared using the same syntax as unpacked arrays, but using $ as the array size.
- In queue 0 is the first, and $ is the last entry
- A queue can be **bounded** or **unbounded**.
  - bounded queue – number of entries specified
  - unbounded queue – unlimited entries

```
int queue_0[$:255] // bounded queue of 255 elements
int queue_1[$];    // unbounded queue of int
queue_1 = {0,1,2,3};
```





Method	Description
size()	returns the number of items in the queue
insert()	inserts the given item at the specified index position
delete()	deletes the item at the specified index position
push_front()	inserts the given element at the front of the queue
push_back()	inserts the given element at the end of the queue
pop_front()	removes and returns the first element of the queue
pop_back()	removes and returns the last element of the queue

BLOCKING VS. NON-BLOCKING ASSIGNMENTS

- **Blocking Assignment**
  - Blocking assignment statements execute in series order.
  - Blocking assignment blocks the execution of the next statement until the completion of the current assignment execution.
  - Use: a = b;
- **Non-Blocking Assignment**
  - non-blocking assignment statements execute in parallel
  - In the non-blocking assignment, all the assignments will occur at the same time (at the end of simulation cycle)
  - Use: a <= b;

UNIQUE IF

- Unique If
  - Unique if evaluates all the conditions parallel.
  - Simulator issue a run time error/warning if zero or more than one condition is true

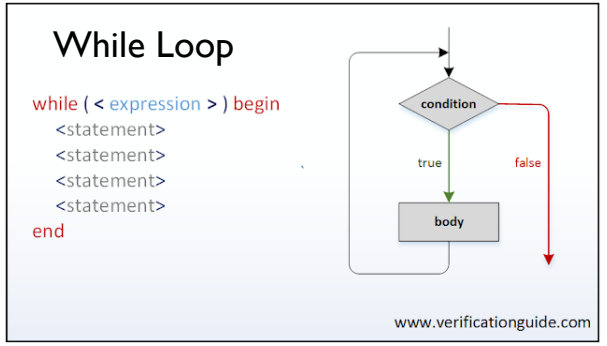
```
// RT Warning: More than one conditions match in 'unique if' statement.
unique if ( a < b ) $display("a is less than b");
else   if ( a < c ) $display("a is less than c");
else                $display("a is greater than b and c");
```

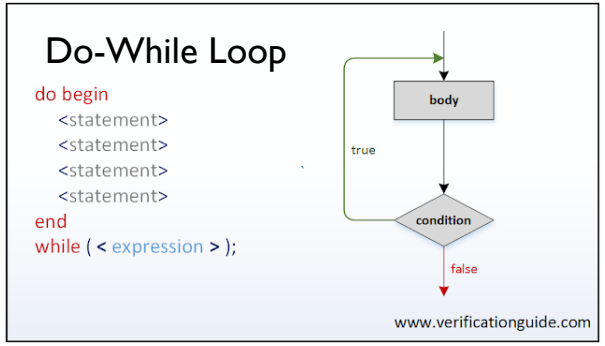
- Priority If
  - Priority if evaluates all the conditions in sequential order.
  - Simulator issue a run time error/warning if no condition is true or no corresponding else

```
// RT Warning: No condition matches in 'priority if' statement.
priority if ( a < 20 ) $display("a is less than b");
else     if ( a < 40 ) $display("a is less than c");
```

LOOPS

- While Loop
  - Condition checked before loop body is executed
  - Loop body is executed 0 or more times.
- Do While Loops
  - Condition checked after the loop body is executed.
  - Loop body executed at least 1 or more times.
- Repeat Loop
  - Executes the loop for a specific number of times.
- Forever Loop
  - Executes the loop forever.
- Break & Continue
  - Use to exit loop or skip body execution






```
repeat(<variable>) begin
  // statement body
end

// example
repeat(10) begin
   if (a % 2 == 1) continue;
   $display("a=%0d",a++);
end
```

```
forever begin
  // statement body
end

// example
forever begin
   #5 $display("a=%0d",a++);
   if (a == 10) break;
end
```

FOR LOOPS

- ForEach Loops
  - Iterates over each element in an array.
  - Loop iterator is defined based on number of elements and dimensions of an array.
- For Loops
  - Verilog
    - Loop iterator must be declared before the loop
    - Only a single initial declaration and step assignment
  - SystemVerilog
    - Loop iterator can be declared in the for-loop
    - One or more initial declaration or assignment within the for loop
    - One or more step assignment or modifier within the for loop

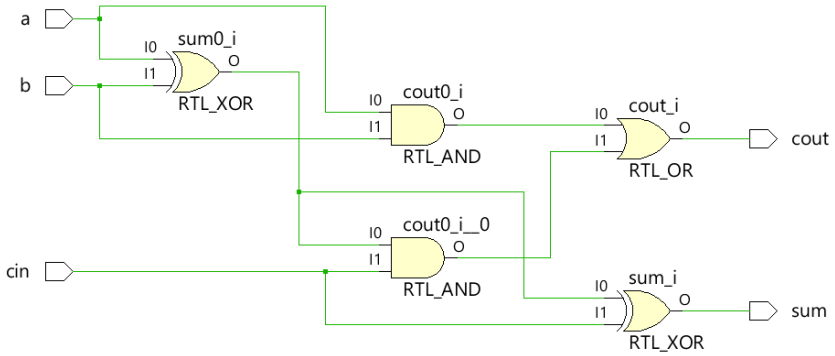
```
foreach ( <variable>[<iterator>] ) begin
  // statements
end

// examples
foreach(a[i]) a[i] = i;
foreach(a[i]) $display("Value of a[%0d]=%0d",i,a[i]);
```

```
for ( initialization; condition; modifier ) begin
  // statements
end

// example
for ( int j=0,i=4; j<8; j++ ) begin
   if( j==i ) $display("j=%0d i=%0d",j,i);
end

string array[4] = '{"apple", "orange", "pear", "grape"};
for ( int i=0; i<$size(array); i++ ) begin
    $display("array[%0d] = %s", i, array[i]);
end
```

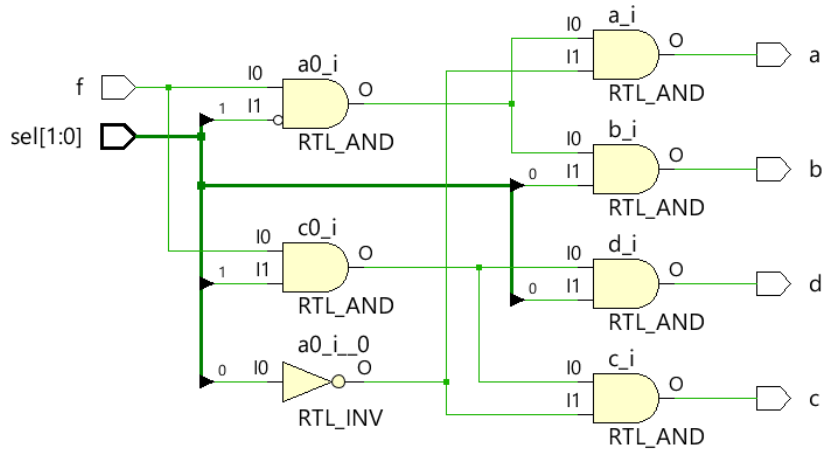
ALWAYS – COMBINATIONAL LOGIC

- Always blocks are combinational logic
- Include input signals in the sensitivity list
- You can also use the wildcard (*) to include all combinational signals

```
// combinational full-adder
always @ (a or b or cin) begin
   {cout, sum} = a + b + cin;
end
```




```
// combination 1x4 demux
always @ ( f or sel) begin
    a = f & ~sel[1] & ~sel[0];
    b = f &  sel[1] & ~sel[0];
    c = f & ~sel[1] &  sel[0];
    d = f &  sel[1] &  sel[0];
end
```



ALWAYS_COMB – COMBINATIONAL PROCESSES

- SystemVerilog introduces always_comb, which does not require a sensitivity list
- Use always_comb blocks with blocking assignments (=)
- Every variable should have a default value to avoid inadvertent introduction of latches
- Don't assign to same variable from more than one always_comb block.
  - Race conditions in behavioral sim,
  - synthesizes incorrectly

```
always_comb begin
   out = 2'd0;  // default value
   if (in1 == 1)
      out = 2'd1;
   else if (in2 == 1)
      out = 2'd2;
   end
```

ALWAYS_FF – CLOCKED PROCESSES

- An always block may be used to implement sequential logic which has memory elements like flip flops that can hold values.
- SystemVerilog introduces always_ff for clocked signals
- Use always_ff @(posedge clk) only with non-blocking assignment operator (<=)
- Use only positive-edge triggered flip-flops for state
- Do not assign the same variable from more than one always_ff block.
  - Race condition in behavioral simulation; synthesizes incorrectly.
- Do not mix blocking and non-blocking assignments
  - only use non-blocking assignments (<=) for sequential logic
  - only use block assignments (=) for combinational logic

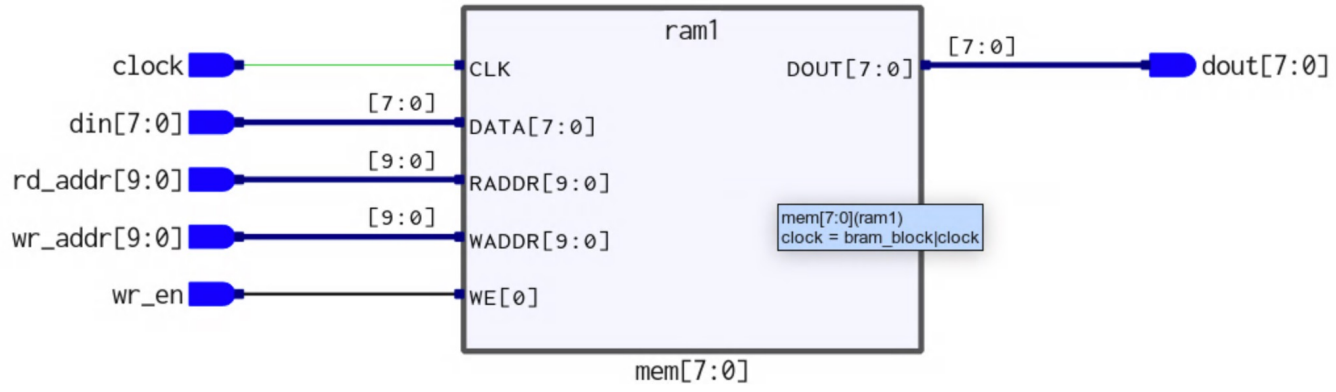
```
module dff (
   input logic clk, rst,
   input logic d,
   output logic q);

   // Verilog clocked process
   always @(posedge clk or posedge rst)
     if (rst == 1'b1) q <= '0;
     else q <= d1;
endmodule
```

```
module dff (
   input logic clk, rst,
   input logic d,
   output logic q);

   // SystemVerilog clocked process
   always_ff @(posedge clk or posedge rst)
     if (rst == 1'b1) q <= '0;
     else q <= d1;
endmodule
```

BLOCK RAM

```
module bram
#(parameter BRAM_ADDR_WIDTH = 10,
  parameter BRAM_DATA_WIDTH = 8)
 (input  logic clock,
  input  logic [BRAM_ADDR_WIDTH-1:0] rd_addr,
  input  logic [BRAM_ADDR_WIDTH-1:0] wr_addr,
  input  logic wr_en,
  input  logic [BRAM_DATA_WIDTH-1:0] din,
  output logic [BRAM_DATA_WIDTH-1:0] dout);

  logic [BRAM_DATA_WIDTH-1:0] mem [0:2**BRAM_ADDR_WIDTH-1];
  logic [BRAM_ADDR_WIDTH-1:0] read_addr;

  always_comb begin
     dout = mem[read_addr];
  end

  always_ff @(posedge clock) begin
    read_addr <= rd_addr;
    if (wr_en) mem[wr_addr] <= din;
  end
endmodule
```

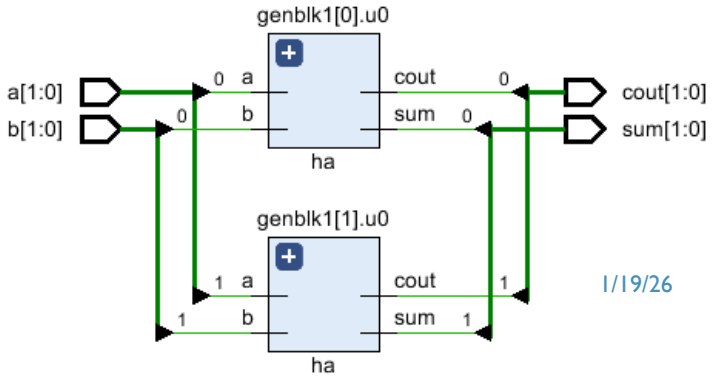
Synthesis tools require that arrays be implemented as **unpacked arrays** to be synthesized as memory.



GENERATE

- A generate block allows to multiply module instances or perform conditional instantiation of any module.
- It provides the ability for the design to be built based on Verilog parameters.
- These statements are particularly convenient when the same operation or module instance needs to be repeated multiple times.
- There are 3 kinds of generate statements:
  - generate-for
  - generate-if
  - generate-case

```
// Design that contains N instances of half adder
module my_ha_design
  #(parameter N=2)
   (input [N-1:0] a, b,
    output [N-1:0] sum, cout);

  // Generate for loop to instantiate N times
  genvar i;
  generate
     // instantiate N instances of half-adder (ha)
     for (i = 0; i < N; i = i + 1) begin
       ha u0(a[i], b[i], sum[i], cout[i]);
     end
  endgenerate
endmodule
```



DISTRIBUTED BLOCK RAM

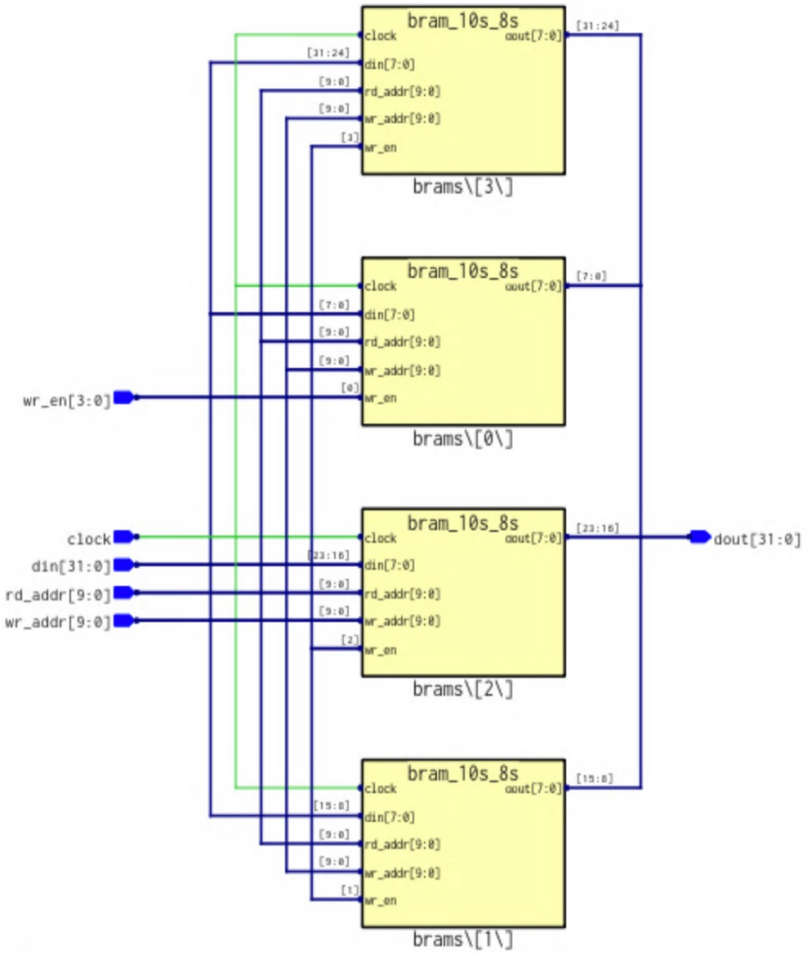
Version 1: Unpacked Array

```
module bram_block
#(parameter BRAM_ADDR_WIDTH = 10,
  parameter BRAM_DATA_WIDTH = 32)
 (input  logic clock,
  input  logic [BRAM_ADDR_WIDTH-1:0] rd_addr,
  input  logic [BRAM_ADDR_WIDTH-1:0] wr_addr,
  input  logic [BRAM_DATA_WIDTH/8-1:0] wr_en,
  input  logic [BRAM_DATA_WIDTH-1:0] din,
  output logic [BRAM_DATA_WIDTH-1:0] dout);

bram #(
  .BRAM_ADDR_WIDTH(BRAM_ADDR_WIDTH),
  .BRAM_DATA_WIDTH(8))
brams [BRAM_DATA_WIDTH/8-1:0] (
  .clock(clock),
  .rd_addr(rd_addr),
  .wr_addr(wr_addr),
  .wr_en(wr_en),
  .dout(dout),
  .din(din)
);

endmodule
```

SystemVerilog allows for implicit distribution of the din and dout signals across the BRAM array.

Version 2: Generate-For

```
module bram_block
#(parameter BRAM_ADDR_WIDTH = 10,
  parameter BRAM_DATA_WIDTH = 32)
 (input  logic clock,
  input  logic [BRAM_ADDR_WIDTH-1:0] rd_addr,
  input  logic [BRAM_ADDR_WIDTH-1:0] wr_addr,
  input  logic [BRAM_DATA_WIDTH/8-1:0] wr_en,
  input  logic [BRAM_DATA_WIDTH-1:0] din,
  output logic [BRAM_DATA_WIDTH-1:0] dout);

  genvar i;
  generate
    for (i=0; i < BRAM_DATA_WIDTH/8; i++)
    begin
      bram #(
       .BRAM_ADDR_WIDTH(BRAM_ADDR_WIDTH),
       .BRAM_DATA_WIDTH(8)
      ) bram_inst (
        .clock(clock),
        .rd_addr(rd_addr),
        .wr_addr(wr_addr),
        .wr_en(wr_en[i]),
        .dout(dout[(i*8)+7 -: 8]),
        .din(din[(i*8)+7 -: 8])
      );
    end
  endgenerate
endmodule
```



NEXT…

- Assignment 1: Fibonacci
- Behavioral Designs