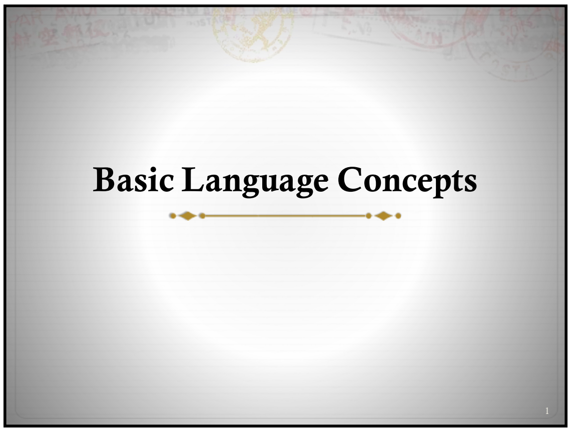
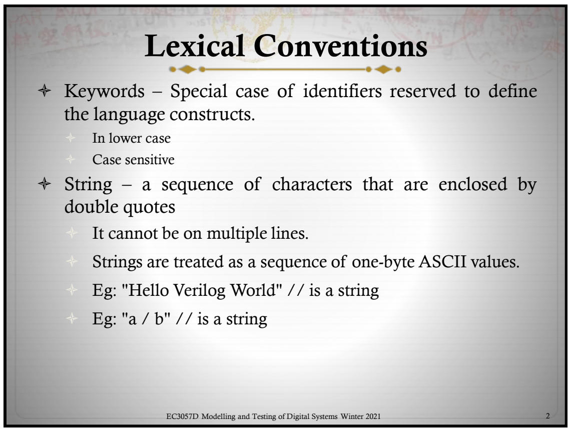
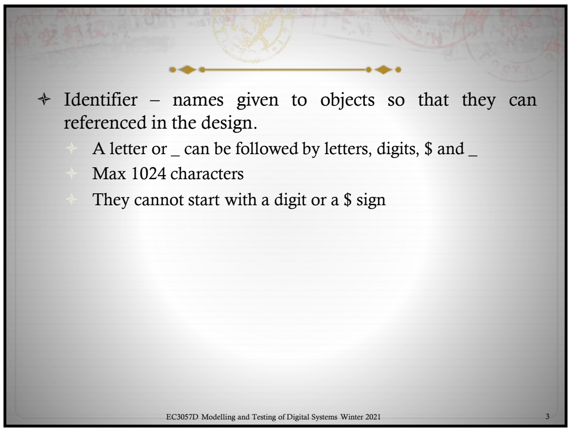


Basic Language Concepts

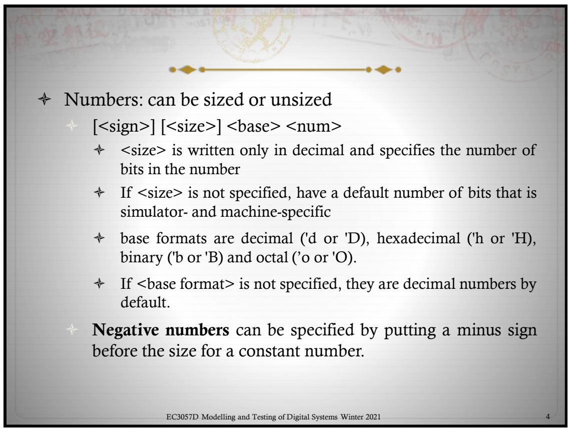


Lexical Conventions

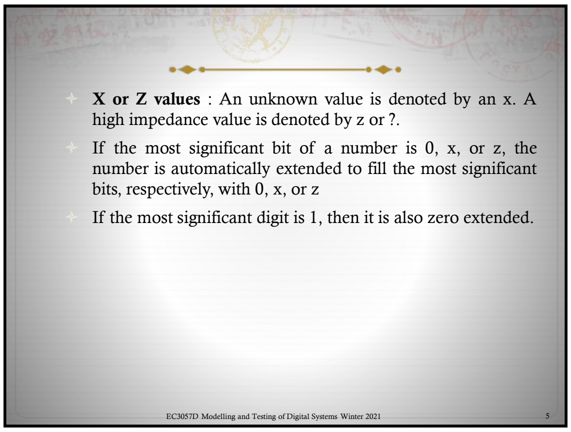
- Keywords – Special case of identifiers reserved to define the language constructs.
  - In lower case
  - Case sensitive
- String – a sequence of characters that are enclosed by double quotes
  - It cannot be on multiple lines.
  - Strings are treated as a sequence of one-byte ASCII values.
  - Eg: "Hello Verilog World" // is a string
  - Eg: "a / b" // is a string



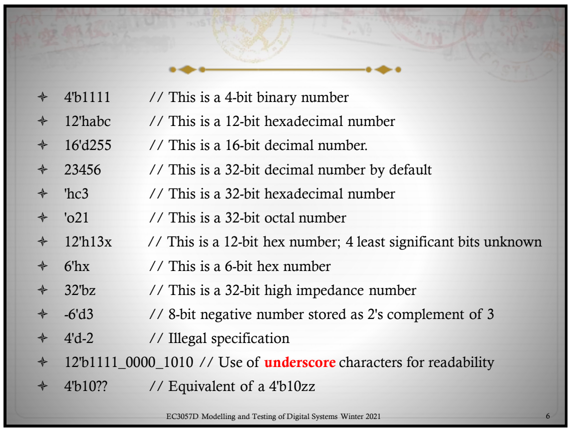
- Identifier – names given to objects so that they can referenced in the design.
  - A letter or _ can be followed by letters, digits, $ and _
  - Max 1024 characters
  - They cannot start with a digit or a $ sign



- Numbers: can be sized or unsized
  - [<sign>] [<size>] <base> <num>
    - <size> is written only in decimal and specifies the number of bits in the number
    - If <size> is not specified, have a default number of bits that is simulator- and machine-specific
    - base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O).
    - If <base format> is not specified, they are decimal numbers by default.
  - **Negative numbers** can be specified by putting a minus sign before the size for a constant number.



- **X or Z values** : An unknown value is denoted by an x. A high impedance value is denoted by z or ?.
- If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z
- If the most significant digit is 1, then it is also zero extended.



```
4'b1111          // This is a 4-bit binary number
12'habc          // This is a 12-bit hexadecimal number
16'd255          // This is a 16-bit decimal number.
23456            // This is a 32-bit decimal number by default
'hc3             // This is a 32-bit hexadecimal number
'o21             // This is a 32-bit octal number
12'h13x          // This is a 12-bit hex number; 4 least significant bits unknown
6'hx             // This is a 6-bit hex number
32'bz            // This is a 32-bit high impedance number
-6'd3            // 8-bit negative number stored as 2's complement of 3
4'd-2            // Illegal specification
12'b1111_0000_1010 // Use of underscore characters for readability
4'b10??          // Equivalent of a 4'b10zz
```

- **Comments**
  - Verilog supports 2 type of comment syntaxes
  - Single line comment start with //, and end with newline.
  - Block comment, start with /*, and end with */. Block comment cannot be nested.

Data Types

Value Set

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

Wire

- **Nets:**
  - Nets represent the connections between hardware elements.
  - Nets are one-bit values by default unless they are declared explicitly as vectors.
  - They are always driven by some source.
  - Default value for any net type variable is ‘z’ (trireg net, which defaults to x)
  - Usually, declared by the keyword ***wire***.

Registers

- Registers represent data storage elements.
- These correspond to variables in the C language.
- A **reg** type variable is the one that can hold a value.
- DO NOT confuse with hardware registers built with flip-flops.
- Register data types always retain their value until another value is placed on them.
- Unlike nets, registers do not need any drivers.

Registers (Cont..)

Example

```
reg reset; // declare a variable reset that can hold its value
initial // this construct will be discussed later
begin
    reset = 1'b1; //initialize reset to 1 to reset the digital circuit.
    #100 reset = 1'b0; // after 100 time units reset is deasserted.
End
```

- Registers can also be declared as signed variables

Example

```
reg signed [63:0] m; // 64 bit signed value
integer i; // 32 bit signed value
```

Rules for reg and wire

- The common rule in Verilog:
  - **"A variable on the Left Hand Side (LHS) of a procedural block assignment is always declared as a register data type."**
  - **"All other variables are of net type."**
- So, reg is assigned within ***always*** or ***initial*** blocks.
- A variable is declared of type wire if it appears on the left side of an continuous assignment statement.
- Continuous assignment statements start with the keyword assign.

Vectors

- Vectors have multiple bits and are often used to represent buses.
- The left most number is an MSB (Most Significant Bit).
- There are 2 representations for vectors:
  - A little-endian notation: [high# : low#]
  - A big-endian notation: [low# : high#]
  - The left number in the squared brackets is always the most significant bit of the vector

```
wire [3:0] busA;  // little-endian notation
wire [0:15] busC; // big-endian notation
reg [1:4] busB;
```

Vectors (Cont…)

- **Vector Part Select**
  - Example 1

```
reg [15:0] data;
reg [0:7] carry;
reg [0:2] inter_carry;
initial begin
    data [15:8] = 8'h12;
     inter_carry = carry [1:3];
end
```

  - Example 2

```
reg [63:0] out;
reg [3:0] dest_addr;
initial begin
   dest_addr = out [63:60];
end
```

  - Example 2

```
reg [2:0] out1;
reg [0:2] out2;
initial begin
  out1 = out2;
end
```

Vector Part Select

```
data[15:8] = 8'h_12;          // Accessing only bits 16 to 9 of data
inter_carry = carry[1:3];
```

Slice Management

```
reg [63:0] out;
reg [3:0] dest_addr;                     dest_addr[0]  = out[60];
initial begin                        =   dest_addr[1]  = out[61];
      dest_addr = out[63:60];            dest_addr[2]  = out[62];
end                                      dest_addr[3]  = out[63];
```

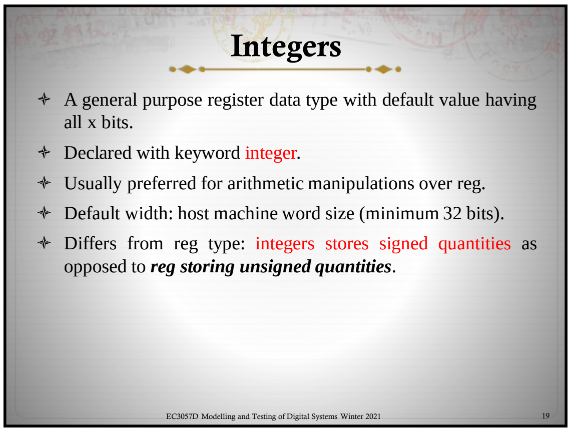
- Vector assignment (**by position!!**)

```
…
reg [2:0] bus_A;
reg [0:2] bus_B;                     bus_A[2]  = bus_B[0];
initial begin                    =   bus_A[1]  = bus_B[1];
     bus_A = bus_B;                  bus_A[0]  = bus_B[2];
end
…
```

Vectors (Cont…)

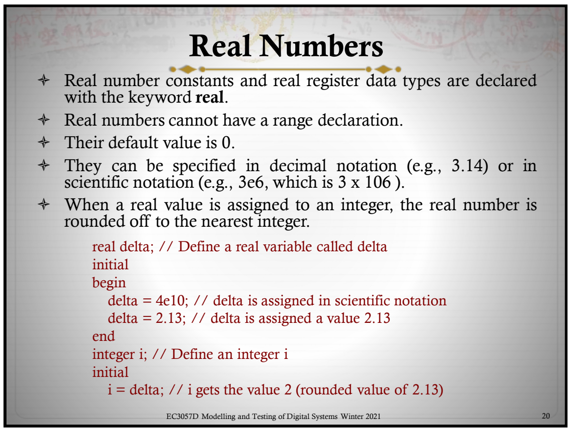
- Variable Vector Part Select
  - [<starting_bit>+ : <width>]
  - [<starting_bit>- : <width>]

```
reg [31:0] data1; reg [0:31] data2;
reg [7:0] byte1; reg [3:0] nibble1;
reg [0:7] byte2; reg [0:3] nibble2;
…
nibble1 = data1[31-:4]; //selects 4 bits from 31 to down, i.e. [31:28]
byte1 = data1[24-:8]; // selects data1[24:17]
byte2 = data2[10+:8]; // selects data2[10:17]
nibble2 = data2[28+:4]; // selects data2[28:31]
```



Integers

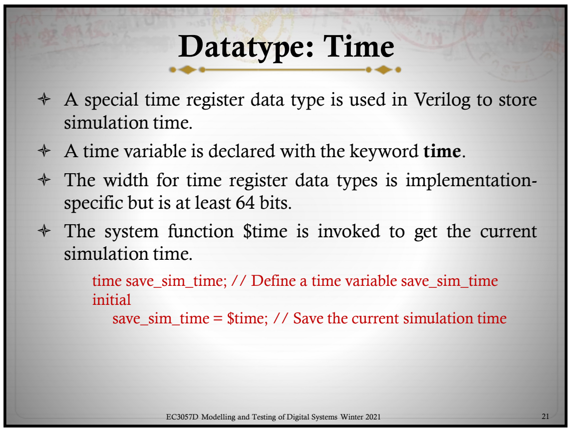
- A general purpose register data type with default value having all x bits.
- Declared with keyword integer.
- Usually preferred for arithmetic manipulations over reg.
- Default width: host machine word size (minimum 32 bits).
- Differs from reg type: integers stores signed quantities as opposed to ***reg storing unsigned quantities***.



Real Numbers

- Real number constants and real register data types are declared with the keyword **real**.
- Real numbers cannot have a range declaration.
- Their default value is 0.
- They can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3 x 106).
- When a real value is assigned to an integer, the real number is rounded off to the nearest integer.

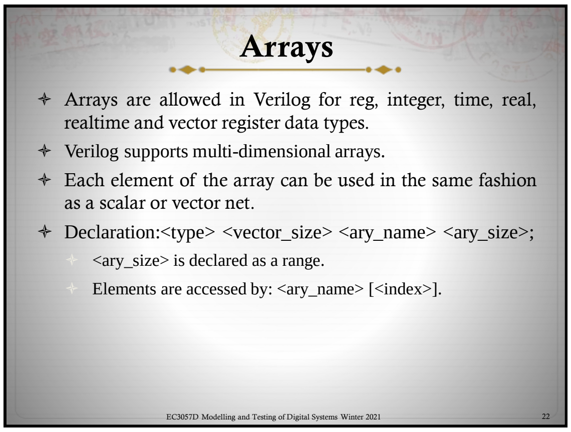
```
real delta; // Define a real variable called delta
initial
begin
   delta = 4e10; // delta is assigned in scientific notation
   delta = 2.13; // delta is assigned a value 2.13
end
integer i; // Define an integer i
initial
   i = delta; // i gets the value 2 (rounded value of 2.13)
```



Datatype: Time

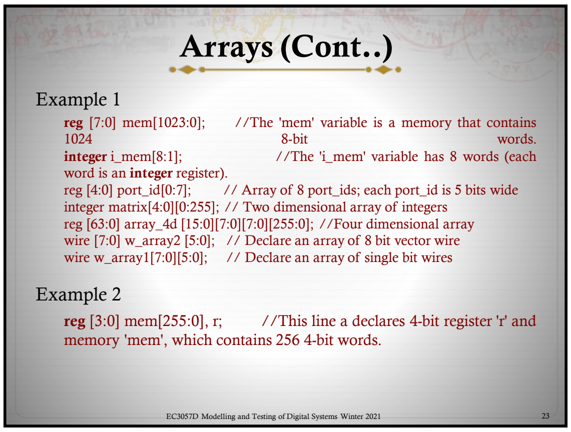
- A special time register data type is used in Verilog to store simulation time.
- A time variable is declared with the keyword **time**.
- The width for time register data types is implementation-specific but is at least 64 bits.
- The system function $time is invoked to get the current simulation time.

```
time save_sim_time; // Define a time variable save_sim_time
initial
   save_sim_time = $time; // Save the current simulation time
```



Arrays

- Arrays are allowed in Verilog for reg, integer, time, real, realtime and vector register data types.
- Verilog supports multi-dimensional arrays.
- Each element of the array can be used in the same fashion as a scalar or vector net.
- Declaration:<type> <vector_size> <ary_name> <ary_size>;
  - <ary_size> is declared as a range.
  - Elements are accessed by: <ary_name> [<index>].



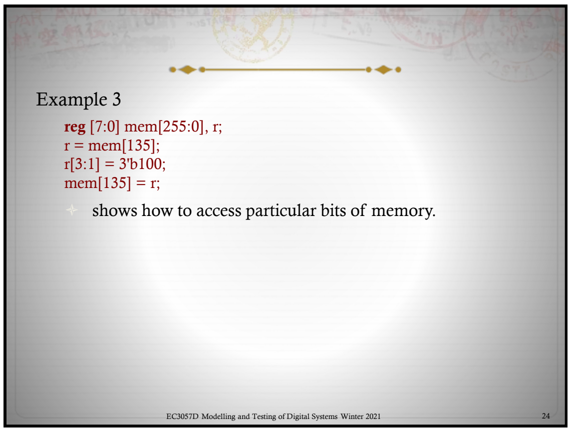
Arrays (Cont..)

Example 1

```
reg [7:0] mem[1023:0];      //The 'mem' variable is a memory that contains 1024 8-bit words.
integer i_mem[8:1];         //The 'i_mem' variable has 8 words (each word is an integer register).
reg [4:0] port_id[0:7];     // Array of 8 port_ids; each port_id is 5 bits wide
integer matrix[4:0][0:255]; // Two dimensional array of integers
reg [63:0] array_4d [15:0][7:0][7:0][255:0]; //Four dimensional array
wire [7:0] w_array2 [5:0];  // Declare an array of 8 bit vector wire
wire w_array1[7:0][5:0];    // Declare an array of single bit wires
```

Example 2

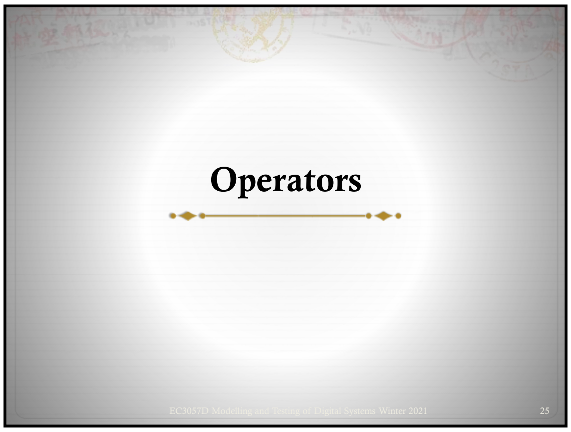
```
reg [3:0] mem[255:0], r;    //This line a declares 4-bit register 'r' and memory 'mem', which contains 256 4-bit words.
```



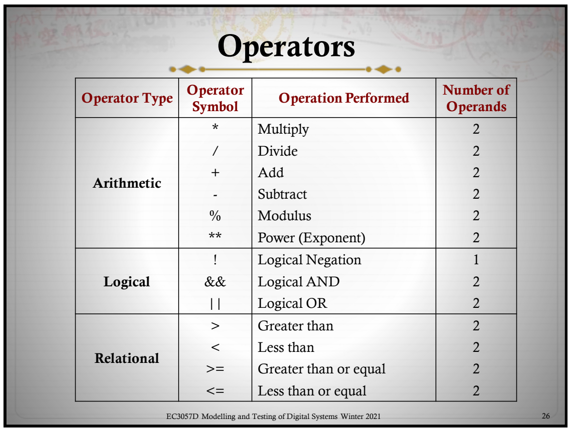
Example 3

```
reg [7:0] mem[255:0], r;
r = mem[135];
r[3:1] = 3'b100;
mem[135] = r;
```

- shows how to access particular bits of memory.

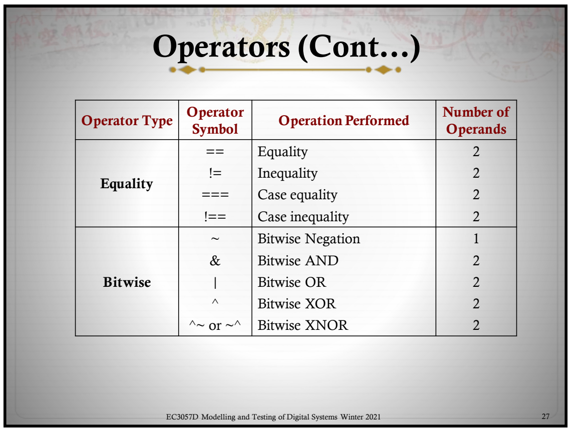


Operators



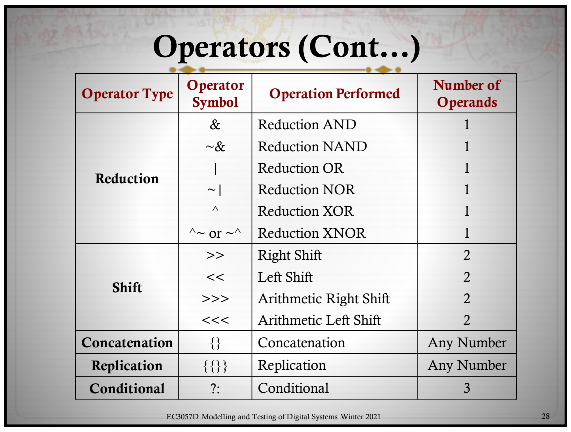
Operators

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	Multiply	2
	/	Divide	2
	+	Add	2
	-	Subtract	2
	%	Modulus	2
	**	Power (Exponent)	2
Logical	!	Logical Negation	1
	&&	Logical AND	2
	||	Logical OR	2
Relational	>	Greater than	2
	<	Less than	2
	>=	Greater than or equal	2
	<=	Less than or equal	2



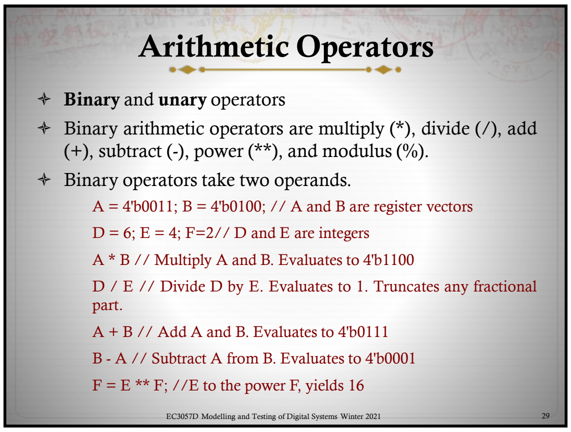
Operators (Cont…)

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Equality	==	Equality	2
	!=	Inequality	2
	===	Case equality	2
	!==	Case inequality	2
Bitwise	~	Bitwise Negation	1
	&	Bitwise AND	2
	|	Bitwise OR	2
	^	Bitwise XOR	2
	^~ or ~^	Bitwise XNOR	2



Operators (Cont…)

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Reduction	&	Reduction AND	1
	~&	Reduction NAND	1
	|	Reduction OR	1
	~|	Reduction NOR	1
	^	Reduction XOR	1
	^~ or ~^	Reduction XNOR	1
Shift	>>	Right Shift	2
	<<	Left Shift	2
	>>>	Arithmetic Right Shift	2
	<<<	Arithmetic Left Shift	2
Concatenation	{ }	Concatenation	Any Number
Replication	{ { } }	Replication	Any Number
Conditional	?:	Conditional	3



Arithmetic Operators

- **Binary** and **unary** operators
- Binary arithmetic operators are multiply (*), divide (/), add (+), subtract (-), power (**), and modulus (%).
- Binary operators take two operands.

    A = 4'b0011; B = 4'b0100; // A and B are register vectors

    D = 6; E = 4; F=2// D and E are integers

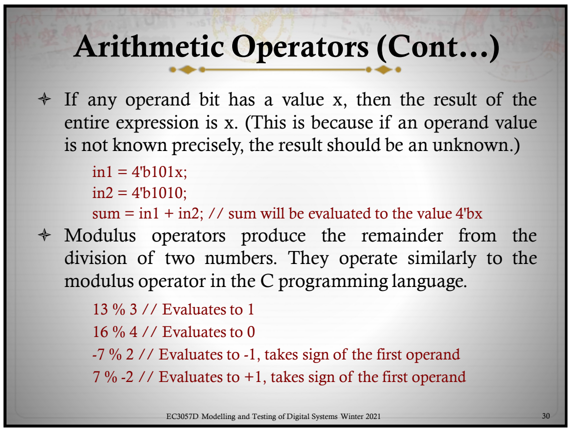
    A * B // Multiply A and B. Evaluates to 4'b1100

    D / E // Divide D by E. Evaluates to 1. Truncates any fractional part.

    A + B // Add A and B. Evaluates to 4'b0111

    B - A // Subtract A from B. Evaluates to 4'b0001

    F = E ** F; //E to the power F, yields 16



Arithmetic Operators (Cont…)

- If any operand bit has a value x, then the result of the entire expression is x. (This is because if an operand value is not known precisely, the result should be an unknown.)

    in1 = 4'b101x;

    in2 = 4'b1010;

    sum = in1 + in2; // sum will be evaluated to the value 4'bx
- Modulus operators produce the remainder from the division of two numbers. They operate similarly to the modulus operator in the C programming language.

    13 % 3 // Evaluates to 1

    16 % 4 // Evaluates to 0

    -7 % 2 // Evaluates to -1, takes sign of the first operand

    7 % -2 // Evaluates to +1, takes sign of the first operand

Arithmetic Operators (Cont…)

- The operators + and - can also work as unary operators.
- They are used to specify the positive or negative sign of the operand.
- Unary + or - operators have higher precedence than the binary + or - operators.

```
-4 // Negative 4
+5 // Positive 5
```

Logical Operators

- Logical operators are logical-and (&&), logical-or (| |) and logical-not (!). Operators && and | | are binary operators.
- **Conditions for Logical Operators**
  1. Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), or x (ambiguous).
  2. If an operand is not equal to zero, it is equivalent to a logical 1 (true condition). If it is equal to zero, it is equivalent to a logical 0 (false condition). If any operand bit is x or z, it is equivalent to x (ambiguous condition) and is normally treated by simulators as a false condition.
  3. Logical operators take variables or expressions as operands.

Logical Operators (Cont…)

```
//General Case
A = 3; B = 0;
A && B      // Evaluates to 0. Equivalent to (logical-1 && logical-0)
A || B      // Evaluates to 1. Equivalent to (logical-1 || logical-0)
!A          // Evaluates to 0. Equivalent to not (logical-1)
!B          // Evaluates to 1. Equivalent to not (logical-0)
// Unknowns
A = 2'b0x; B = 2'b10;
A && B      // Evaluates to x. Equivalent to (x && logical 1)
// Expressions
(a == 2) && (b == 3) // Evaluates to 1 if both a == 2 and b == 3 are true.
                     // Evaluates to 0 if either is false.
```

Relational operators

- Relational operators are greater-than (>), less-than (<), greater-than-or-equal-to (>=), and less-than-or-equal-to (<=).
- If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and 0 if the expression is false.
- If there are any unknown or z bits in the operands, the expression takes a value x.

```
A = 4, B = 3
X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx
    A <= B              // Evaluates to a logical 0
    A > B               // Evaluates to a logical 1
    Y >= X              // Evaluates to a logical 1
    Y < Z               // Evaluates to an x
```

Equality Operators

- Equality operators are logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==).
- When used in an expression, equality operators return logical value 1 if true, 0 if false.
- These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length.
- The logical equality operators (==, !=) will yield an x, if either operand has x or z in its bits.
- The case equality operators (===, !==) compare both operands bit by bit, including x and z and results is 1 if the operands match exactly, including x and z bits. The result is 0 if the operands do not match exactly.
- Case equality operators never result in an x.

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101
// Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx
A == B // Results in logical 0
X != Y // Results in logical 1
X == Z // Results in x
Z === M // Results in logical 1 (all bits match, including x and z)
Z === N // Results in logical 0 (least significant bit does not match)
M !== N // Results in logical 1
```

Bitwise operators

- Bitwise operators are negation (~), and(&), or (|), xor (^), xnor (^~, ~^).
- Bitwise operators perform a bit-by-bit operation on two operands.
- If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand. A z is treated as an x in a bitwise operation.
- The unary negation operator (~), which takes only one operand and operates on the bits of the single operand.

```
// X = 4'b1010, Y = 4'b1101,  Z = 4'b10x1

~X      // Negation. Result is 4'b0101
X & Y   // Bitwise and. Result is 4'b1000
X | Y   // Bitwise or. Result is 4'b1111
X ^ Y   // Bitwise xor. Result is 4'b0111
X ^~ Y  // Bitwise xnor. Result is 4'b1000
X & Z   // Result is 4'b10x0
```

Reduction Operators

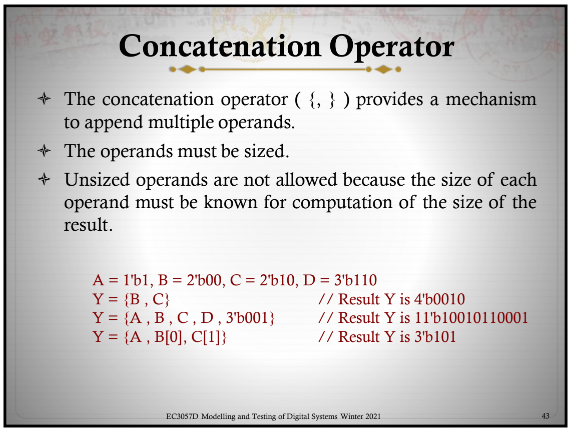
- Reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (~^, ^~).
- Reduction operators take only one operand.
- Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result.
- Reduction operators work bit by bit from right to left.
  - Reduction nand, reduction nor, and reduction xnor are computed by inverting the result of the reduction and, reduction or, and reduction xor, respectively

```
// X = 4'b1010
&X      //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|X      //Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
^X      //Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
        //A reduction xor or xnor can be used for even or odd
        //parity generation of a vector.
```

Shift Operators

- Shift operators are right shift (>>), left shift (<<), arithmetic right shift (>>>), and arithmetic left shift (<<<).
- The operands are the vector and the number of bits to shift.
- When the bits are shifted, the vacant bit positions are filled with zeros.
- Shift operations do not wrap around.
- Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.
  - **Arithmetic Shift Right (>>>)**: Shift right specified number of bits, fill the value of sign bit if the expression is signed, otherwise fill with 0.
  - **Arithmetic Shift Left (>>>)**: Shift left specified number of bits, filling with 0

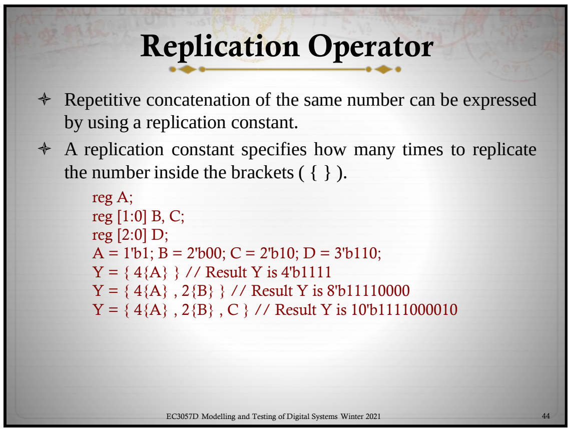
```
// X = 4'b1100
Y = X >> 1; //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB
position.
Y = X << 1; //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB
position.
Y = X << 2;     //Y is 4'b0000. Shift left 2 bits.

integer a, b, c;    //Signed data types
a = 0;
b = -10;            // 111...10110 binary
c = a + (b >>> 3); //Results in -2 decimal, due to arithmetic shift
```



Concatenation Operator

- The concatenation operator ({, }) provides a mechanism to append multiple operands.
- The operands must be sized.
- Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result.

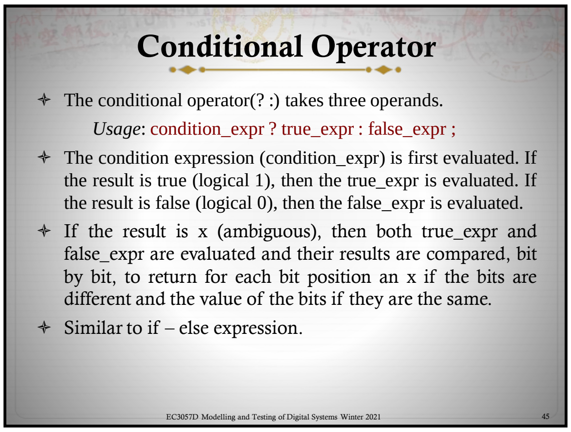
```
A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110
Y = {B , C}                        // Result Y is 4'b0010
Y = {A , B , C , D , 3'b001}       // Result Y is 11'b10010110001
Y = {A , B[0], C[1]}               // Result Y is 3'b101
```



Replication Operator

- Repetitive concatenation of the same number can be expressed by using a replication constant.
- A replication constant specifies how many times to replicate the number inside the brackets ({ }).

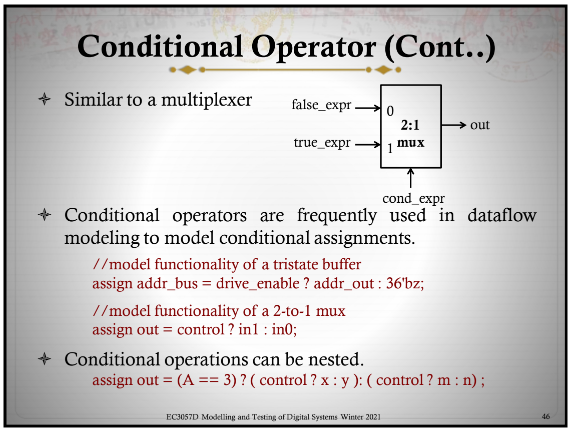
```
reg A;
reg [1:0] B, C;
reg [2:0] D;
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;
Y = { 4{A} } // Result Y is 4'b1111
Y = { 4{A} , 2{B} } // Result Y is 8'b11110000
Y = { 4{A} , 2{B} , C } // Result Y is 10'b1111000010
```



Conditional Operator

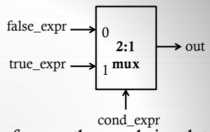
- The conditional operator(? :) takes three operands.

  Usage: condition_expr ? true_expr : false_expr ;
- The condition expression (condition_expr) is first evaluated. If the result is true (logical 1), then the true_expr is evaluated. If the result is false (logical 0), then the false_expr is evaluated.
- If the result is x (ambiguous), then both true_expr and false_expr are evaluated and their results are compared, bit by bit, to return for each bit position an x if the bits are different and the value of the bits if they are the same.
- Similar to if – else expression.



Conditional Operator (Cont..)

- Similar to a multiplexer

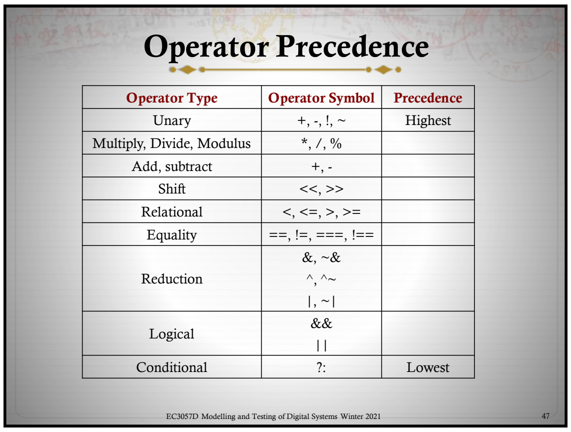


- Conditional operators are frequently used in dataflow modeling to model conditional assignments.

```
//model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 36'bz;

//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
```

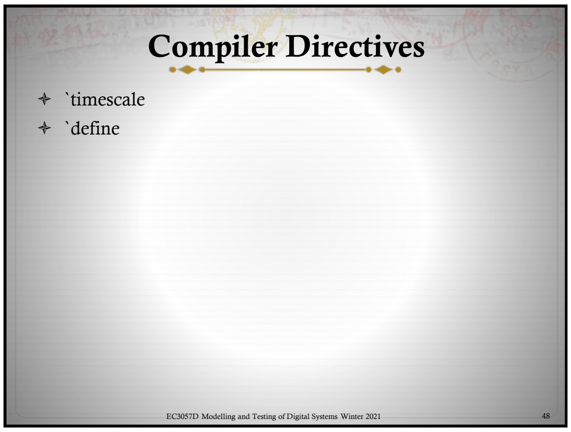
- Conditional operations can be nested.

```
assign out = (A == 3) ? ( control ? x : y ): ( control ? m : n) ;
```



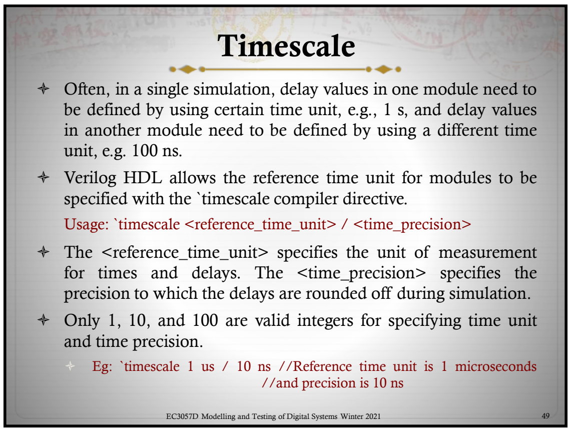
Operator Precedence

Operator Type	Operator Symbol	Precedence
Unary	+, -, !, ~	Highest
Multiply, Divide, Modulus	*, /, %	
Add, subtract	+, -	
Shift	<<, >>	
Relational	<, <=, >, >=	
Equality	==, !=, ===, !==	
Reduction	&, ~& ^, ^~ |, ~|	
Logical	&& ||	
Conditional	?:	Lowest



Compiler Directives

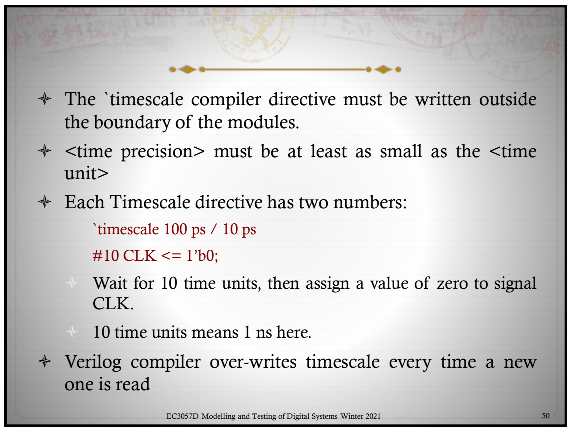
- `timescale
- `define



Timescale

- Often, in a single simulation, delay values in one module need to be defined by using certain time unit, e.g., 1 s, and delay values in another module need to be defined by using a different time unit, e.g. 100 ns.
- Verilog HDL allows the reference time unit for modules to be specified with the `timescale compiler directive.

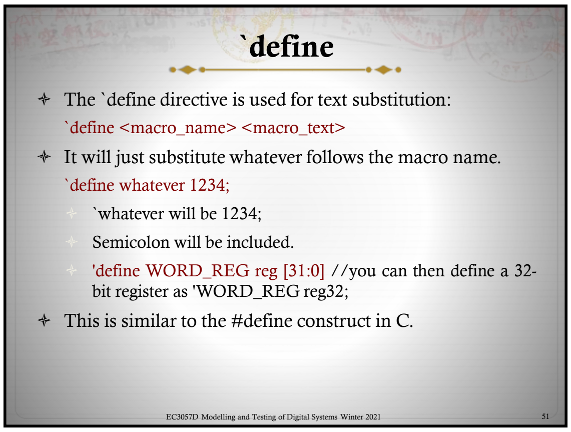
  Usage: `timescale <reference_time_unit> / <time_precision>
- The <reference_time_unit> specifies the unit of measurement for times and delays. The <time_precision> specifies the precision to which the delays are rounded off during simulation.
- Only 1, 10, and 100 are valid integers for specifying time unit and time precision.
  - Eg: `timescale 1 us / 10 ns //Reference time unit is 1 microseconds //and precision is 10 ns



- The `timescale compiler directive must be written outside the boundary of the modules.
- <time precision> must be at least as small as the <time unit>
- Each Timescale directive has two numbers:

  `timescale 100 ps / 10 ps

  #10 CLK <= 1'b0;
  - Wait for 10 time units, then assign a value of zero to signal CLK.
  - 10 time units means 1 ns here.
- Verilog compiler over-writes timescale every time a new one is read

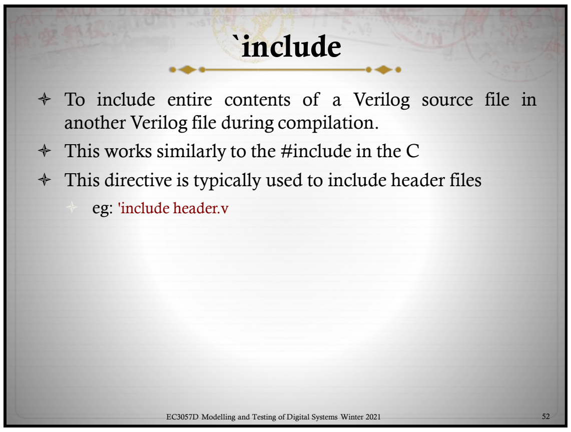


`define

- The `define directive is used for text substitution:

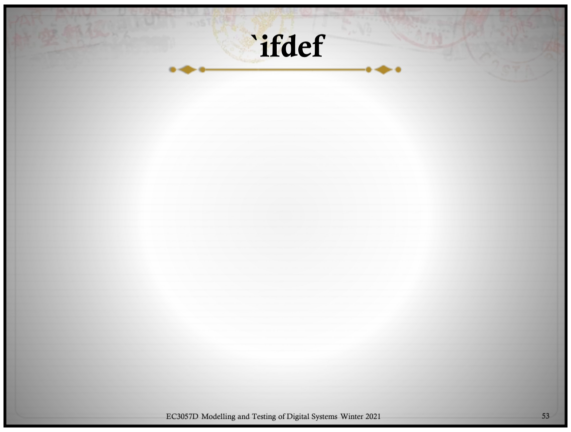
  `define <macro_name> <macro_text>
- It will just substitute whatever follows the macro name.

  `define whatever 1234;
  - `whatever will be 1234;
  - Semicolon will be included.
  - 'define WORD_REG reg [31:0] //you can then define a 32-bit register as 'WORD_REG reg32;
- This is similar to the #define construct in C.

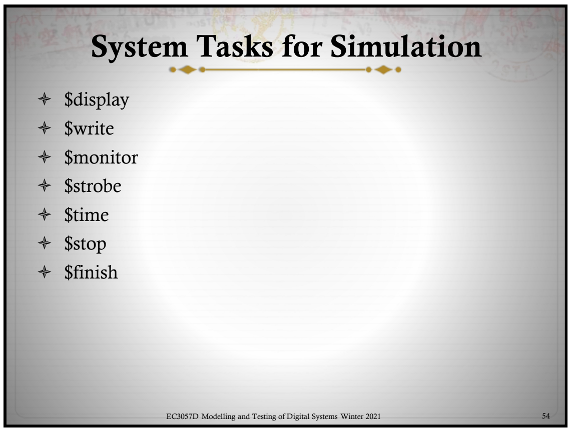


`include

- To include entire contents of a Verilog source file in another Verilog file during compilation.
- This works similarly to the #include in the C
- This directive is typically used to include header files
  - eg: 'include header.v

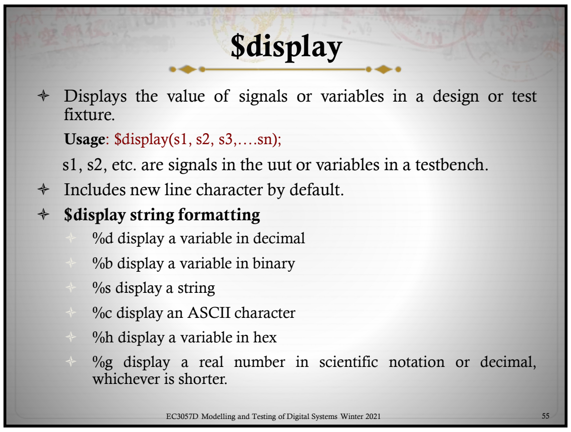


`ifdef



System Tasks for Simulation

- $display
- $write
- $monitor
- $strobe
- $time
- $stop
- $finish

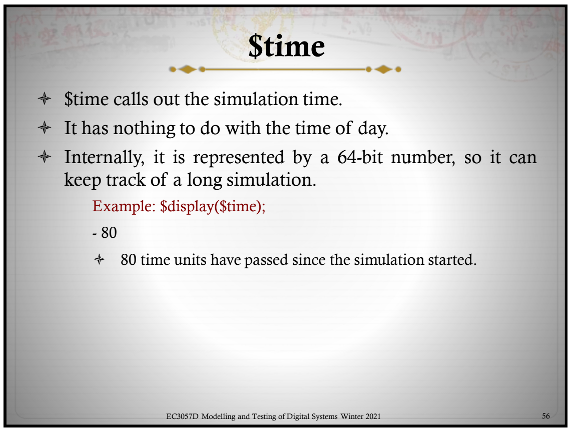


$display

- Displays the value of signals or variables in a design or test fixture.

  Usage: $display(s1, s2, s3,….sn);

  s1, s2, etc. are signals in the uut or variables in a testbench.
- Includes new line character by default.
- **$display string formatting**
  - %d display a variable in decimal
  - %b display a variable in binary
  - %s display a string
  - %c display an ASCII character
  - %h display a variable in hex
  - %g display a real number in scientific notation or decimal, whichever is shorter.

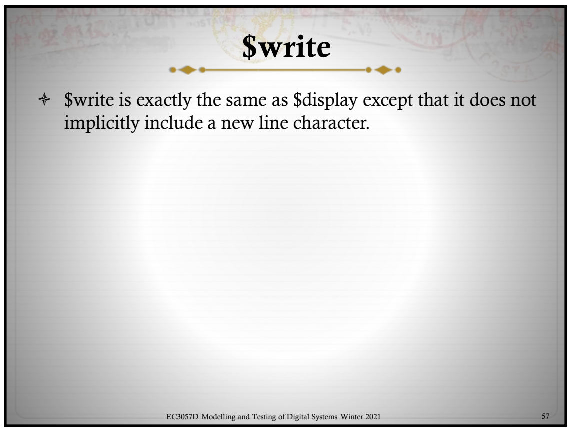


$time

- $time calls out the simulation time.
- It has nothing to do with the time of day.
- Internally, it is represented by a 64-bit number, so it can keep track of a long simulation.

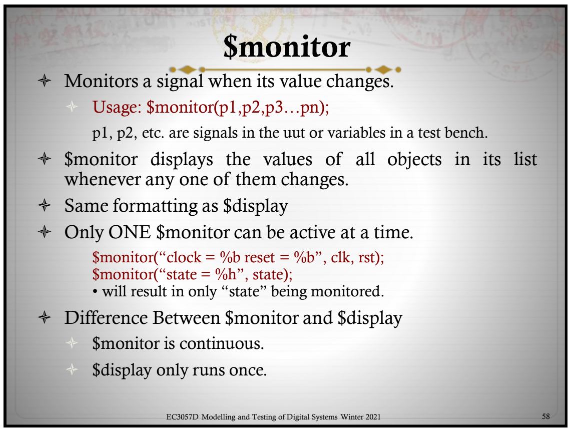
  Example: $display($time);

  - 80
  - 80 time units have passed since the simulation started.



$write

- $write is exactly the same as $display except that it does not implicitly include a new line character.

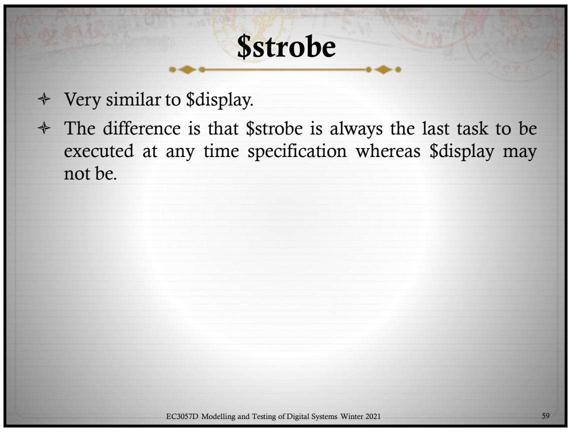


$monitor

- Monitors a signal when its value changes.
  - Usage: $monitor(p1,p2,p3…pn);

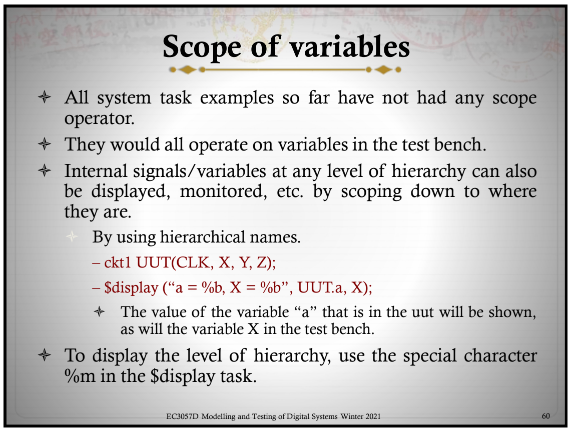
  p1, p2, etc. are signals in the uut or variables in a test bench.
- $monitor displays the values of all objects in its list whenever any one of them changes.
- Same formatting as $display
- Only ONE $monitor can be active at a time.

  $monitor(“clock = %b reset = %b”, clk, rst);
  $monitor(“state = %h”, state);
  • will result in only “state” being monitored.
- Difference Between $monitor and $display
  - $monitor is continuous.
  - $display only runs once.



$strobe

- Very similar to $display.
- The difference is that $strobe is always the last task to be executed at any time specification whereas $display may not be.

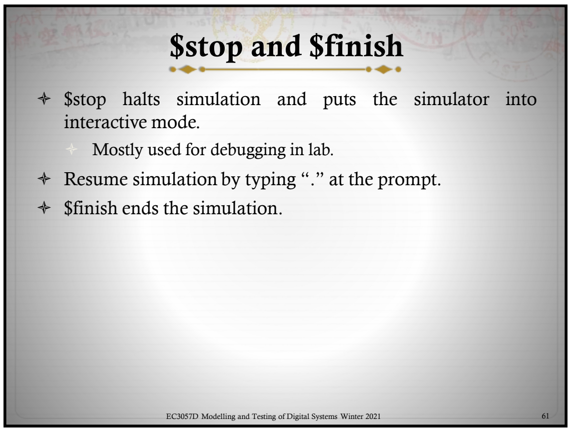


Scope of variables

- All system task examples so far have not had any scope operator.
- They would all operate on variables in the test bench.
- Internal signals/variables at any level of hierarchy can also be displayed, monitored, etc. by scoping down to where they are.
  - By using hierarchical names.

    – ckt1 UUT(CLK, X, Y, Z);

    – $display (“a = %b, X = %b”, UUT.a, X);
    - The value of the variable “a” that is in the uut will be shown, as will the variable X in the test bench.
- To display the level of hierarchy, use the special character %m in the $display task.



$stop and $finish

- $stop halts simulation and puts the simulator into interactive mode.
  - Mostly used for debugging in lab.
- Resume simulation by typing "." at the prompt.
- $finish ends the simulation.