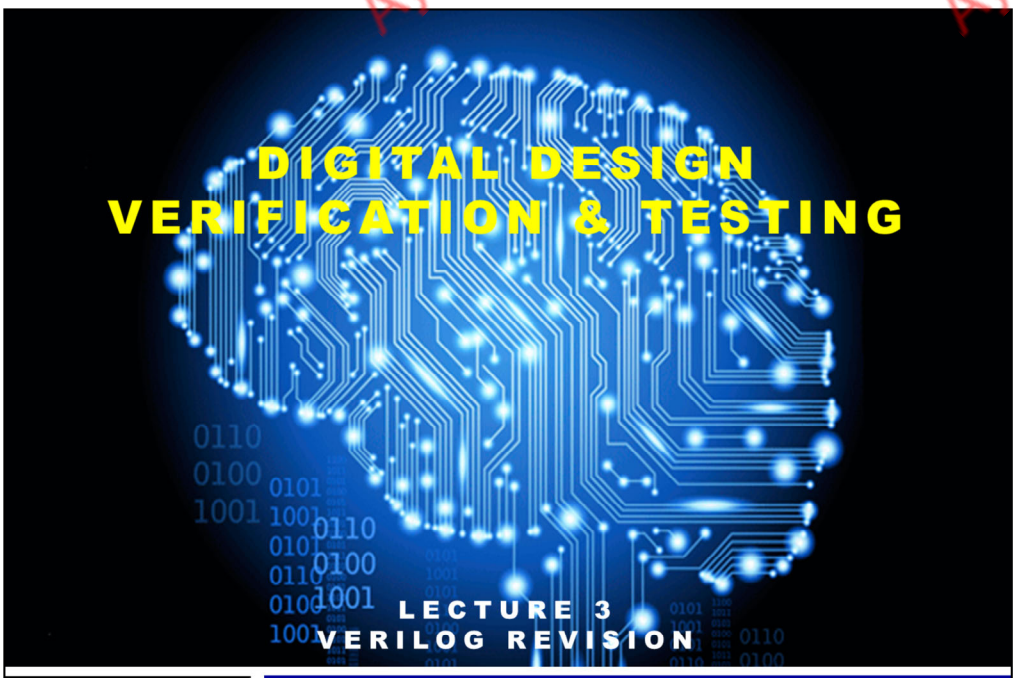
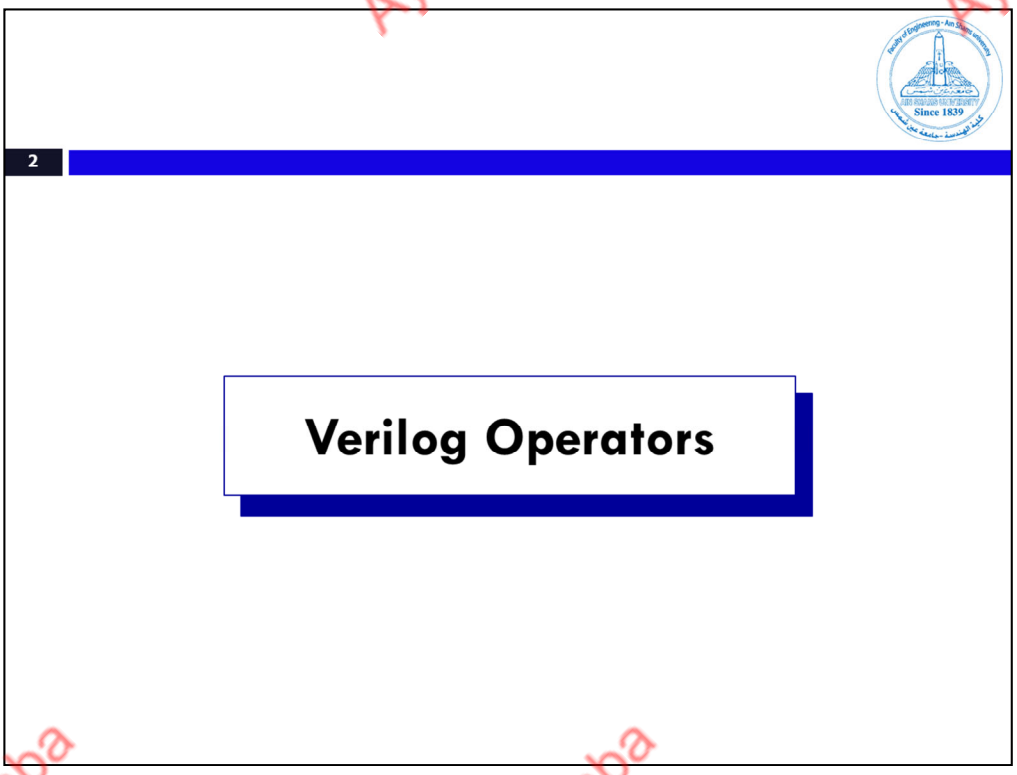


CSE 313s

Ayman wahba



Relational Operators

- 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
- The result is:
  - 0 if the relation is false
  - 1 if the relation is true
  - x if either of the operands is x or z
- Note: If a value is x or z, then the result of that test is false

Relational Operators (example)


```
module des;
  reg [7:0] a;
  reg [7:0] b;

  initial begin
    a = 45;
    b = 9;
    $display ("Result for %0d >= %0d: %b", a, b, a >= b);

    a = 8'b00001z01;
    b = 45;
    $display ("Result for %0d <= %0d: %0d", a,b, a <= b);

    a = 9;
    b = 8;
    $display ("Result for %0d > %0d: %0d", a, b, a > b);

    a = 22;
    b = 22;
    $display ("Result for %0d < %0d: %0d", a, b, a < b);
  end
endmodule
```

```
Result for 45 >= 9: 1
Result for Z <= 45: x
Result for 9 > 8: 1
Result for 22 < 22: 0
```

Arithmetic Operators

- **Binary: +, -, *, /, % (the modulus operator)**
- **Unary: +, -**
- **Integer division truncates any fractional part**
- **The result of a modulus operation is always positive.**
- **If any operand bit value is the unknown value x, or z then the entire result value is x**
- **Register data types are used as unsigned values**
- **Negative numbers are stored in two's complement format**

- Binary operator, means that it has two operands

Arithmetic Operators (example)


```
module des;
  reg [7:0] a,b; int c,d;
  initial begin
    a = 45; b = 6;
    $display("Result for %0d + %0d: %0d", a, b, a + b);
    $display("Result for %0d - %0d: %0d", a, b, a - b);
    $display("Result for %0d * %0d: %0d", a, b, a * b);
    $display("Result for %0d / %0d: %0d", a, b, a / b);
    $display("Result for %0d %% %0d: %0d", a, b, a % b);
    $display("Result for %0d ** 3: %0d", b, b ** 3);
    a = -31; b = 4; $display("Result for %0d %% %0d: %0d", a, b, a % b);
    a = 4; b = -31; $display("Result for %0d %% %0d: %0d", a, b, a % b);
    a = 8'b00001111; b = 8'b01z00000; $display("Result for %b + %b: %b", a, b, a + b);
    a = 5; b = 14; $display("Result for %0d - %0d: %b", a, b, a - b);
    c = -31; d = 4; $display("Result for %0d %% %0d: %0d", c, d, c % d);
    c = 31;  d = -4;$display("Result for %0d %% %0d: %0d", c, d, c % d);
    c = -31; d = -4;$display("Result for %0d %% %0d: %0d", c, d, c % d);
  end
endmodule
```

```
Result for 45 + 6: 51
Result for 45 - 6: 39
Result for 45 * 6: 14
Result for 45 / 6: 7
Result for 45 % 6: 3
Result for 6 ** 3: 216
```

```
Result for 225 % 4: 1
Result for 4 % 225: 4
Result for 00001111 + 01z00000: xxxxxxxx
Result for 5 - 14: 11110111
Result for -31 % 4: -3
Result for 31 % -4: 3
Result for -31 % -4: -3
```

- Note that -31 is dealt with as 225 because reg data type is always considered as unsigned numbers. -31 in two's complement is: 11100001 which is seen as (128+64+32+1 = 225), so -31 % 4 is dealt with as 225 % 4 = 1.
- When any of the operands has an, x or z, bit, the whole result is x. So

  00001111 + 01z00000 = xxxxxxxx
- When we try the mod operator (%) with integer numbers, the sign of the result is the same as the first operand.

Equality Operators

- a = = = b exact comparison, including x and z (result 1 or 0)
- a ! = = b exact comparison, including x and z (result 1 or 0)
- a = = b a equal to b, result may be x (result 1, 0, or x)
- a ! = b a not equal to b, (result 1, 0, or x)

- Operands are compared bit by bit, with zero filling if the two operands do not have the same length
- For the = = and ! = operators: the result is x, if either operand contains an x or a z
- For the = = = and ! = = operators: bits with x and z are included in the comparison and must match for the result to be true. The result is always 0 or 1.

Equality Operators (example)


```
module equality;
  reg [7:0] a;
  reg [7:0] b;

  initial begin
    a='b11x0;   b='b1100; $display ("Result for %0b === %0b: %0d", a,b, a === b);
    a='b11x0;   b='b1100; $display ("Result for %0b == %0b: %0d", a,b, a == b);
    a='b111100; b='b1100; $display ("Result for %0b == %0b: %0d", a,b, a == b);
    a='b001100; b='b1100; $display ("Result for %0b == %0b: %0d", a,b, a == b);
    a='b11x0;   b='b11x0; $display ("Result for %0b === %0b: %0d", a,b, a === b);
    a='b11z0;   b='b11x0; $display ("Result for %0b === %0b: %0d", a,b, a === b);
    a='b11x0;   b='b11x0; $display ("Result for %0b == %0b: %0d", a,b, a == b);

  end
endmodule
```

```
Result for 11x0 === 1100: 0
Result for 11x0 == 1100: x
Result for 111100 == 1100: 0
Result for 1100 == 1100: 1
Result for 11x0 === 11x0: 1
Result for 11z0 === 11x0: 0
Result for 11x0 == 11x0: x
```

Logical Operators

- ! logic negation (converts non zero value to 0 and vice versa)
- && logical and
- || logical or

- Expressions connected by && and || are evaluated from left to right.
- Evaluation stops as soon as the result is known.
- The result is a scalar value:
  - 0 if the relation is false.
  - 1 if the relation is true
  - x if the result can't be determined

Logical Operators (example)


```
module equality;
  reg [7:0] a;
  reg [7:0] b;

  initial begin
    a='b1110;   b='b1100; $display ("Result for %0b && %0b: %0d", a,b, a && b);
    a='b11x0;   b='b1100; $display ("Result for %0b && %0b: %0d", a,b, a && b);
    a='b00x0;   b='b1110; $display ("Result for %4b && %0b: %0d", a,b, a && b);
    a='b00z0;   b='b1110; $display ("Result for %4b && %0b: %0d", a,b, a && b);
    a='b11z0;   b='b1100; $display ("Result for %4b && %0b: %0d", a,b, a && b);
    a='b0011;   b='b1100; $display ("Result for %4b || %0b: %0d", a,b, a || b);
    a='b1100;             $display ("Result for ! %4b: %0d", a, !a);
    a='b0000;             $display ("Result for ! %4b: %0d", a, !a);
    a='b1x00;             $display ("Result for ! %4b: %0d", a, !a);
    a='bxxxx;             $display ("Result for ! %4b: %0d", a, !a);
  end
endmodule
```

```
Result for 1110 && 1100: 1
Result for 11x0 && 1100: 1
Result for 00x0 && 1110: x
Result for 00z0 && 1110: x
Result for 11z0 && 1100: 1
Result for 0011 || 1100: 1
```

```
Result for ! 1100: 0
Result for ! 0000: 1
Result for ! 1x00: 0
Result for ! xxxx: x
```

Bit-Wise Operators

- ~ negation
- & and
- | or
- ^ exclusive or
- ^~ or ~^ exclusive nor (equivalence)

&	0	1	X	Z
1	0	1	X	X
0	0	0	0	0
X	0	X	X	X
Z	0	X	X	X

|	0	1	X	Z
1	1	1	1	1
0	0	1	X	X
X	X	1	X	X
Z	X	1	X	X

^	0	1	X	Z
1	1	0	X	X
0	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

~1	0
~0	1
~X	X
~Z	X

- When operands are of unequal bit length, the shorter operand is zero filled in the most significant bit positions

Bit-Wise Operators (example)


```
module equality;
  reg [3:0] a;
  reg [3:0] b;

  initial begin
    a='b1110;   b='b1100; $display ("Result for %0b & %0b: %4b", a,b, a & b);
    a='b11x0;   b='b1100; $display ("Result for %0b & %0b: %4b", a,b, a & b);
    a='b00x0;   b='b1110; $display ("Result for %4b & %0b: %4b", a,b, a & b);
    a='b00z0;   b='b1110; $display ("Result for %4b & %0b: %4b", a,b, a & b);
    a='b11z0;   b='b1100; $display ("Result for %4b | %0b: %4b", a,b, a | b);
    a='b0011;   b='b1100; $display ("Result for %4b | %0b: %4b", a,b, a | b);
  end
endmodule
```

```
Result for 1110 & 1100: 1100
Result for 11x0 & 1100: 1100
Result for 00x0 & 1110: 00x0
Result for 00z0 & 1110: 00x0
Result for 11z0 | 1100: 11x0
Result for 0011 | 1100: 1111
```

Reduction Operators

- & and
- ~& nand
- | or
- ~| nor
- ^ xor
- ^~ or ~^ xnor

- Reduction operators are unary.
- They perform a bit-wise operation on a single operand to produce a single bit result.
- Reduction unary NAND and NOR operators operate as AND and OR respectively, but with their outputs negated.
- Unknown bits are treated as described before.

Reduction Operators (example)


```
module equality;
  reg [7:0] a;

  initial begin
    a='b11101111; $display ("Result for & %8b: %0b", a, &a);
    a='b11101x11; $display ("Result for & %8b: %0b", a, &a);
    a='b11111x11; $display ("Result for & %8b: %0b", a, &a);
    a='b11111011; $display ("Result for ^ %8b: %0b", a, ^a);
    a='b11001010; $display ("Result for ^ %8b: %0b", a, ^a);
    a='b11001010; $display ("Result for | %8b: %0b", a, |a);
  end
endmodule
```

```
Result for & 11101111: 0
Result for & 11101x11: 0
Result for & 11111x11: x
Result for ^ 11111011: 1
Result for ^ 11001010: 0
Result for | 11001010: 1
```

Shift Operators

- << left shift
- >> right shift

- The left operand is shifted by the number of bit positions given by the right operand.

- The vacated bit positions are filled with zeroes.

Shift Operators (example)


```
module shift;
  reg [7:0] data;
  int i;
  initial begin
    data = 8'b00000101;
    $display ("Original data = %8b", data);
    for (i = 0; i < 9; i +=1 ) begin
      $display ("data << %0d = %8b", i, data << i);
    end

    data = 8'b11000000;
    $display ("Original data = %8b", data);
    for (i = 0; i < 9; i +=1 ) begin
      $display ("data >> %0d = %8b", i, data >> i);
    end
  end
endmodule
```

```
Original data = 00000101
data << 0 = 00000101
data << 1 = 00001010
data << 2 = 00010100
data << 3 = 00101000
data << 4 = 01010000
data << 5 = 10100000
data << 6 = 01000000
data << 7 = 10000000
data << 8 = 00000000
```

```
Original data = 11000000
data >> 0 = 11000000
data >> 1 = 01100000
data >> 2 = 00110000
data >> 3 = 00011000
data >> 4 = 00001100
data >> 5 = 00000110
data >> 6 = 00000011
data >> 7 = 00000001
data >> 8 = 00000000
```

Concatenation Operators

- Concatenations are expressed using the brace characters { and }, with commas separating the expressions within.
- Examples
  {a, b[3:0], c, 4'b1001} // if a and c are 8-bit numbers, the results has 24 bits.
  Unsized constant numbers are not allowed in concatenations.
- Repetition multipliers that must be constants can be used:
  {3{a}} // this is equivalent to {a, a, a}
- Nested concatenations are possible:
  {b, {3{c, d}}} // this is equivalent to {b, c, d, c, d, c, d}

```
1  module test;
2    logic [7:0] a, b, c;
3    logic [23:0] d;
4    logic [55:0] e;
5
6    initial begin
7      a = 8'b11001110;
8      b = 8'b01011111;
9      c = 8'b11110000;
10     # 10;
11     d = {a, b[3:0], c, 4'b1001};
12     # 10;
13     d = {3{a}};
14     # 10;
15     e = {b, {3{c, a}}};
16     #30;
17     a = 8'b00000000;
18   end
19
20   initial begin
21     $dumpfile("test.vcd");
22     $dumpvars;
23   end
24 endmodule
```

Concatenation Operators (example)


```
module concat;
  reg a;
  reg b;
  reg [2:0] c;
  reg [1:0] short_result;
  reg [7:0] result;

  initial begin
    a = 1; b = 0; c = 3'b110;
    short_result = {a,b};
    result = {b, a, c[1:0], 3'b001, c[2]};
    $display("a = %b, b = %b, c = %b", a, b, c);
    $display("{a,b} = %b", short_result);
    $display("{b, a, c[1:0], 3'b001, c[2]} = %b", result);
    $display("{3{c}} = %b", {3{c}});
    $display("{b,{3{c,a}}} = %b", {b,{3{c,a}}});
  end
endmodule
```

```
a = 1, b = 0, c = 110
{a,b} = 10
{b, a, c[1:0], 3'b001, c[2]} = 01100011
{3{c}} = 110110110
{b,{3{c,a}}} = 0110111011101
```

Conditional Operators

- The conditional operator has the following C-like format:

  cond_expr ? *true_expr* : *false_expr*

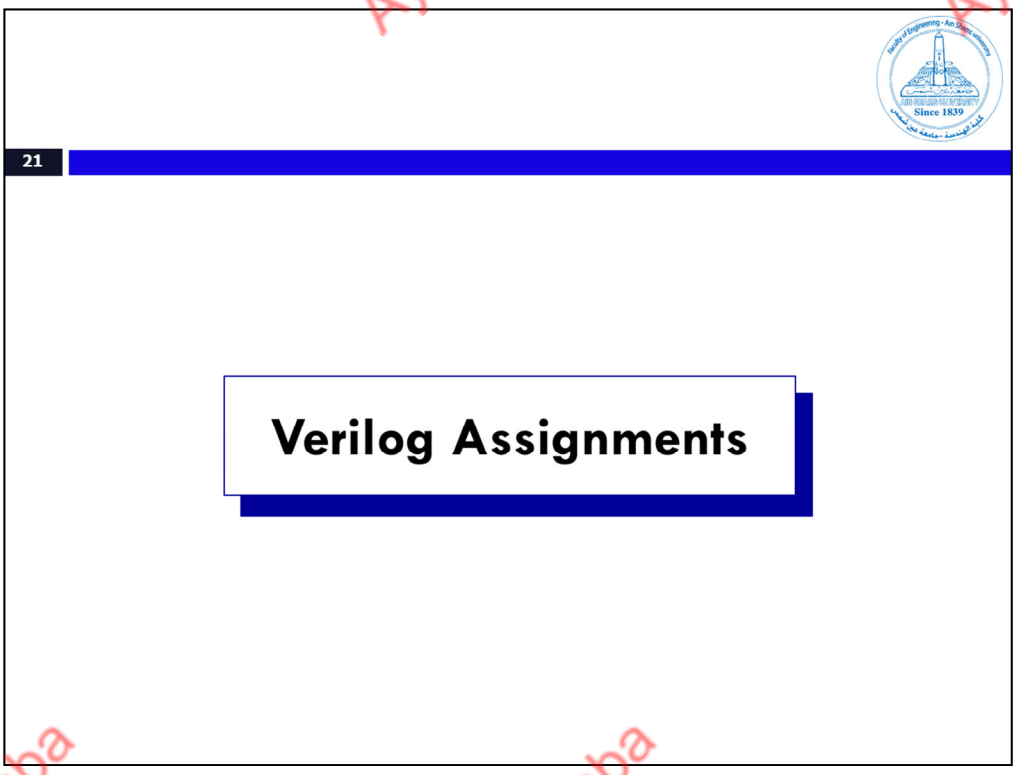
- The *true_expr* or the *false_expr* is evaluated and used as a result depending on whether *cond_expr* evaluates to true or false

Example:

```
out = (enable) ? data : 8'bz; // Tri state buffer
```

Operator Precedence

- Unary, Multiply, Divide, Modulus + - ! ~ * / %
- Add, Subtract, Shift. +, - , <<, >>
- Relation, Equality <, >, <=, >=, ==, !=, ===, !===
- Reduction &, !&,^,^~,|,~|
- Logic &&, ||
- Conditional ? :



Non-Blocking Assignment

c <= a&b;	<= non-blocking
	
	always @(posedge clk)
	begin
	test_1 <= 1'b1;
	test_2 <= test_1;
	test_3 <= test_2;
	end
		• Used only inside an always or initial block.
		• All these statements are executed in parallel. So every LHS will take the value of the RHS at the moment of entering the block.
		• Here 1 will be entered into test_3 at the 3rd edge of the clock.
		• LHS should always be a reg or a vector of reg.
		• RHS can be of type reg or wire.

```
module test;
  int a, b, c, d, e;

  initial begin
    a <= 5;
    b <= 23;
    c <= 10;
    d <= 0;
    # 10;

    d <= a + b; // d = 28 (1C hex)
    e <= d + c; // e = 10 (A hex)

    # 10;
    d = a + b;  // d = 28 (1C hex)
    e = d + c;  // d = 38 (26 hex)

    #30;
    d = 0;
  end

  initial begin
    $dumpfile("test.vcd");
    $dumpvars;
  end
endmodule
```

Blocking Assignment

c = 1'b0;	= blocking assignment
	
	always @(posedge clk)
	begin
	test_1 = 1'b1;
	test_2 = test_1;
	tet_3 = test_2;
	end
		• Used only inside an always or initial block.
		• The statements are executed in sequence, one after the other, exactly like software.
		• Here 1 will be entered into test_3 when at the 1st edge of the clock.
		• LHS should always be a reg or a vsector of reg.
		• RHS can be of type reg or wire.

Continuous Assignment

assign c = ~a;		• Signals of type wire require continuous assignment of a value. • Used when you connect gates together. • LHS should always be a wire or a vector of wires. It can never be reg • RHS can be of type reg or wire. • The assignment is always active. Whenever any operand on the RHS changes in value, LHS will be updated with the new value.

Data Types

Verilog Types

Verilog Language has two primary data types:

- **Nets**: represents structural connections between components.
- **Registers**: represent variables used to store data.

Every signal has a data type associated with it:

- Explicitly declared with a declaration in your Verilog code.
- Implicitly declared with no declaration but used to connect structural building blocks in your code.
- Implicit declaration is always a net of type wire and is one bit wide.

Wire:

Wire data type is used in the continuous assignments or ports list. It is treated as a wire So it cannot hold a value. It can be driven and read. Wires are used for connecting different modules.

- Wires are used for connecting different elements.
- They can be treated as *physical wires*.
- They can be read or assigned.
- No values get stored in them.
- They need to be driven by either *continuous assign* statement or from a *port of a module*.

Reg:

Reg is a date storage element in Verilog. It's not an actual hardware register but it can store values. Registers retain their value until next assignment statement.

- Contrary to their name, regs do not necessarily correspond to physical registers.
- They represent data storage elements in Verilog.
- They retain their value till next value is assigned to them.

Port Connection Rules

- **Inputs** : internally must always be type *net*, externally the inputs can be connected to variable *reg* or *net* type.
- **Outputs** : internally can be type *net* or *reg*, externally the outputs must be connected to a variable net type.
- **Inouts** : internally or externally must always be type *net*, can only be connected to a variable net type.