Chapter 3: Dataflow Modeling

Prof. Soo-Ik Chae

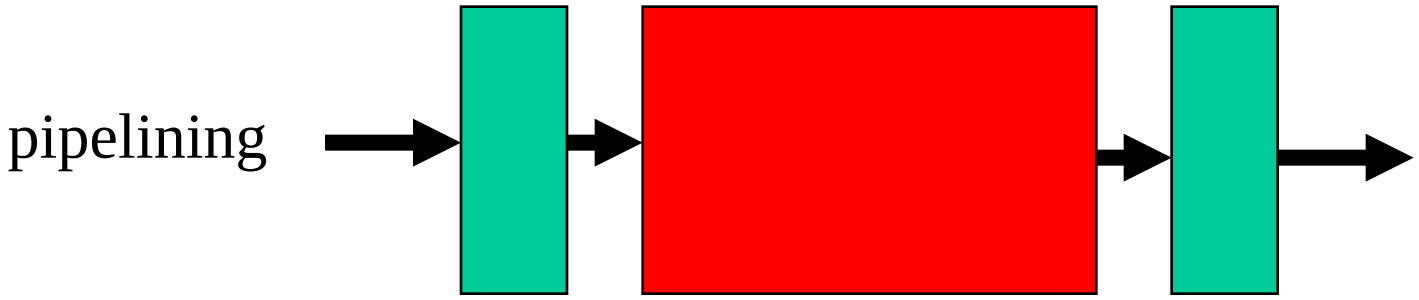
Objectives

After completing this chapter, you will be able to:

- Describe what is the dataflow modeling
- Describe how to use continuous assignments
- Describe how to specify delays in continuous assignments
- Describe the data types allowed in Verilog HDL
- Describe the operation of the operators used in Verilog HDL
- Describe the operands may be used associated with a specified operator

Why Dataflow ?

- Rationale of dataflow: any digital system can be constructed by interconnecting registers and a combinational logic put between them for performing the necessary functions.
  - Dataflow provides a powerful way to implement a design.
  - Logic synthesis tools can be used to create a gate-level circuit from a dataflow design description.
  - RTL (register transfer level) is a combination of dataflow and behavioral modeling.



Assignments

- Two basic forms of assignments
  - Continuous assignment: assign values to nets
  - Procedural assignment: assign values to variables
- Two additional forms of assignments: procedural continuous assignments
  - assign/deassign
  - force/release
- An assignment consists of two parts: a LHS and a RHS separated by = or <=
  - RHS: any expression that evaluates to a value to which the LHS value is to be assigned.
  - LHS: can take one of the forms given in Table 30.

Assignments

Table 6-1—Legal left-hand forms in assignment statements

Statement type	Left-hand side
Continuous assignment	Net (vector or scalar) Constant bit-select of a vector net Constant part-select of a vector net Constant indexed part-select of a vector net Concatenation or nested concatenation of any of the above left-hand side
Procedural assignment	Variables (vector or scalar) Bit-select of a vector reg, integer, or time variable Constant part-select of a vector reg, integer, or time variable Indexed part-select of a vector reg, integer, or time variable Memory word Concatenation or nested concatenation of any of the above left-hand side

Continuous Assignments

- Continuous assignment: the most basic statement of dataflow modeling.
  - It is used to drive a value onto a net.
  - It is always active.
  - Provides a way to model combinational logic without specifying an interconnection of gates. Instead, it specifies the logical expression that drives the net.
  - It can only update values of net data types such as wire, triand, etc.
  - This assignment shall occur whenever the value of the right-hand side changes.

Continuous Assignments

- A continuous assignment begins with the keyword assign.

```
assign net_lvalue = expression;

assign net1 = expr1,
       net2 = expr2,
       ...,
       netn = exprn;
```

  - net_lvalue is a scalar or vector net, or their concatenation.
  - RHS operands can be variables or nets or function calls.
  - Registers or nets can be scalar or vectors.
  - Delay values can be specified.

Continuous Assignments

- Assignments on nets shall be continuous and automatic
- This means that
  - whenever an operand in the RHS expression changes value, the whole RHS shall be evaluated and if the new value is different from the previous value, then the new value shall be assigned to the LHS.

Continuous Assignments

- An implicit continuous assignment
  - is the shortcut of declaring a net first and then writing a continuous assignment on the net.
  - is always active.
  - can only have one implicit declaration assignment per net.

```
wire out;                    // net declaration
assign out = in1 & in2; // regular continuous assignment

wire  out = in1 & in2;    // implicit continuous assignment
```

Continuous Assignments

- An implicit net declaration
  - is a feature of Verilog HDL.
  - will be inferred for a signal name when it is used to the left of a continuous assignment.

```
wire in1, in2;
assign out = in1 & in2;
```

Note that: out is not declared as a wire, but an implicit wire declaration for out is done by the simulator.

Delays

- Three ways of specifying delays
  - Regular assignment delay
  - Implicit continuous assignment delay
  - Net declaration delay
- Regular assignment delays
  - The delay value is specified after the keyword assign.
  - The inertial delay model is used (default model).

```
wire in1, in2, out;
assign #10 out = in1 & in2;
```

Delays

❖ Implicit continuous assignment delays

- An implicit continuous assignment is used to specify both the delay and assignment on the net.
- The inertial delay model is used (default model).

```
// implicit continuous assignment delay
wire #10 out = in1 & in2;    // the delay is part of the continuous assignment
                             // and is not a net delay.
// regular assignment delay
wire out;
assign #10 out = in1 & in2;
```

Continuous assignment and delays

- In situations where a RHS operand changes before a previous change has had to propagate to the LHS, then the following steps are taken.

1. The value of the RHS expression is evaluated
2. If the RHS value differs from the value currently scheduled to propagate the LHS, then the currently scheduled propagation event is descheduled.
3. If the new RHS value equals the current LHS value, no event is scheduled.
4. If the new RHS value differs from the current LHS value, a delay is calculated in the standard way using the current value of the LHS, the newly calculated value of the RHS, and the delays indicated on the statement; a new propagate event is then scheduled to occur delay time units in the future.

Delays

❖ Net declaration delays

- A net can be declared associated with a delay value.
- Net declaration delays can also be used in gate-level modeling.

```
// net  delays
wire #10 out;                    // transport delay
assign out = in1 & in2;

// regular assignment delay
wire out;
assign #10 out = in1 & in2;     // inertial delay
```

The Basis of Dataflow Modeling

- The essence of dataflow modeling is

  expression = operators + operands

  - Operands can be any one of allowed data types.
  - Operators act on the operands to product desired results.

  Operands:

  - constants
  - integers
  - real numbers
  - nets
  - registers
  - time
  - bit-select
  - part-select
  - memories
  - function calls

Operators

Arithmetic	Bitwise	Reduction	Relational
+: add - : subtract * : multiply / : divide % : modulus **: exponent	~ : NOT & : AND | : OR ^ : XOR ~^, ^~ : XNOR	& : AND | : OR ~& : NAND ~| : NOR ^ : XOR ~^, ^~ : XNOR	>: greater than <: less than >= : greater than or equal <=: less than or equal
Shift	**case equality**	**Logical**	**Miscellaneous**
<< : left shift >> : right shift <<< : arithmetic left shift >>>: arithmetic right shift	===: equality !==: inequality	&&: AND || : OR ! : NOT	{ , }: concatenation {c{ }}: replication ? : conditional
	Equality		
	==: equality !=: inequality		

Precedence of Operators

Operators	Symbols	Precedence
Unary	+ - ! ~	Highest
Exponent	**	
Multiply, divide, modulus	* / %	
Add, subtract	+ -	
Shift	<< >> <<< >>>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	& ~& ^ ^~ | ~|	
Logical	&& ||	
Conditional	?:	Lowest

Operands

- The operands in an expression can be any of:
  - constants,
  - parameters,
  - nets,
  - variables (reg, integer, time, real, realtime),
  - bit-select,
  - part-select,
  - array element, and
  - function calls

Constants

- Three types of constant in Verilog HDL are
  - integer: a general-purpose variable used of manipulating quantities that are not regarded as hardware registers.
  - real, and
  - string

Constants

- Integer constant
  - simple decimal form

    -123 // is decimal -123

    12345 // is decimal 12345
  - base format notation

    16'habcd // a 16-bit hexadecimal number

    2006 // unsized number--a 32-bit decimal number

    4'sb1001 // a 4-bit signed number, it represents -7.

Constants

- Real constant
  - decimal notation
    1.5 //
    .3 // illegal ---
    1294.872 //
  - scientific notation
    15E12
    32E-6
    26.176_45_e-12
- String constant
  - A string is a sequence of characters enclosed by double quotes ("").
  - It may not be split into multiple lines.
  - One character is represented as an 8-bit ASCII code.

Data Types

- Two classes of data types:
  - nets: Nets mean any hardware connection points.
  - variables: Variables represent any data storage elements.
- Variable data types
  - reg
  - integer
  - time
  - real
  - realtime

Variable Data Types

- A reg variable
  - holds a value between assignments.
  - may be used to model hardware registers.
  - need not actually represent a hardware storage element.

```
reg a, b;                 // reg a, and b are 1-bit reg
reg [7:0] data_a;    // an 8-bit reg, the msb is bit 7
reg [0:7] data_b;    // an 8-bit reg, the msb is bit 0
reg signed [7:0] d; // d is an 8-bit signed reg
```

The integer Variable

- The integer variable
  - contains integer values.
  - has at least 32 bits.
  - is treated as a signed reg variable with the lsb being bit 0.

```
integer i,j;          // declare two integer variables
integer data[7:0]; // array of integer
```

The time Variable

- The time variable
  - is used for storing and manipulating simulation time quantities.
  - is typically used in conjunction with the $time system task.
  - holds only unsigned value and is at least 64 bits, with the lsb being bit 0.

```
time events;          // hold one time value
time current_time; // hold one time value
```

The real and realtime Variables

- The real and realtime variables
  - cannot use range declaration and
  - their initial values are defaulted to zero (0.0).

```
real events;                // declare a real variable
realtime current_time; // hold current time as real
```

Vectors

- A vector (multiple bit width) describes a bundle of signals as a basic unit.
  - [high:low] or [low:high]
  - The leftmost bit is the MSB.
  - Both nets and reg data types can be declared as vectors.
- The default is 1-bit vector or called scalar.

Bit-Select and Part-Select

❖ Bit-Select and Part-Select

- integer and time can also be accessed by bit-select or part-select.
- real and realtime are not allowed to be accessed by bit-select or part-select.
- Constant part select: data_bus[3:0], bus[3]
- Variable part select:

  [<starting_bit>+:width]: data_bus[8+:8]

  [<starting_bit>-:width]: data_bus[15-:8]

Array and Memory Elements

- Array and Memory Elements
  - all net and variable data types are allowed to be declared as multi-dimensional arrays.
  - an array element can be a scalar or a vector if the element is a net or reg data type.

```
wire a[3:0];              // a scalar wire array of 4 elements
reg  d[7:0];              // a scalar reg array of 8 elements
wire [7:0]  x[3:0];       // an 8-bit wire array of 4 elements
reg  [31:0] y[15:0];      // a 32-bit reg array of 16 elements
integer states [3:0];     // an integer array of 4 elements
time    current[5:0];     // a time array of 6 elements
```

The Memory

- Memory
  - Memory is used to model a read-only memory (ROM), a random access memory (RAM), and a register file.
  - Reference to a memory may be made to a whole word or a portion of a word of memory.

```
reg [3:0] mema [7:0];        // 1-d array of 4-bit vector
reg [7:0] memb [3:0][3:0]; // 2-d array of 8-bit vector
wire sum [7:0][3:0];         // 2-d array of scalar wire

mema[4][3]        // the 3rd bit of 4th element
mema[5][7:4]      // the higher four bits of 5th element
memb[3][1][1:0] // the lower two bits of [3][1]th element
sum[5][0]         // [5][0]th element
```

Bitwise Operators

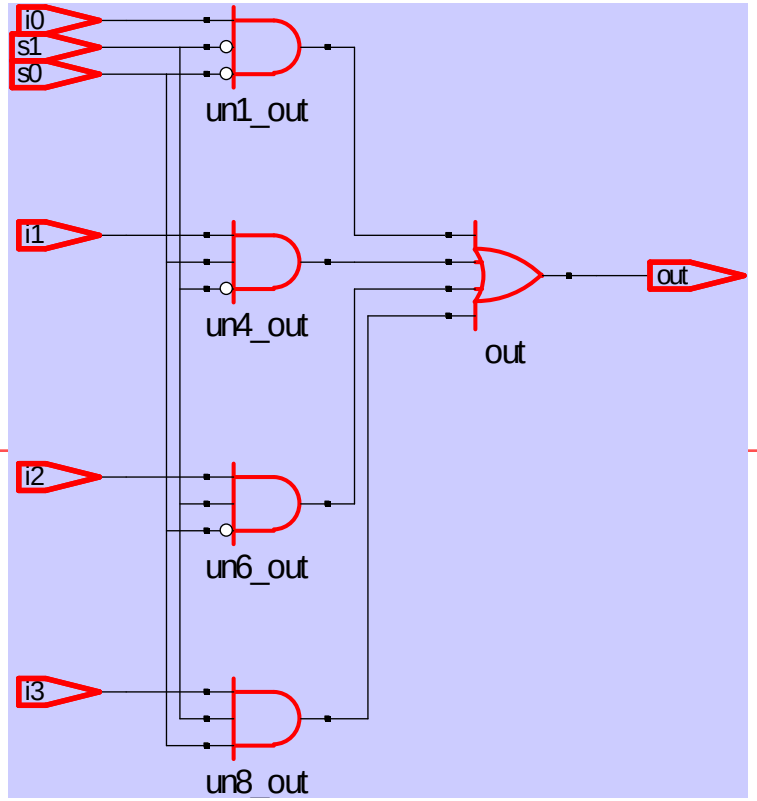
❖ Bitwise operators

- They perform a bit-by-bit operation on two operands.
- A z is treated as x in bit-wise operation.
- The shorter operand is zero-extended to match the length of the longer operand.

Symbol	Operation
~	Bitwise negation
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
~^, ^~	Bitwise exclusive nor

A 4-to-1 MUX

```
module mux41_dataflow(i0, i1, i2, i3, s1, s0, out);
// Port declarations
input i0, i1, i2, i3;
input s1, s0;
output out;
// Using basic and, or , not logic operators.
assign out =    (~s1 & ~s0 & i0) |
                (~s1 & s0 & i1)   |
                (s1 & ~s0 & i2)   |
                (s1 & s0 & i3) ;
endmodule
```



Arithmetic Operators

❖ Arithmetic operators

- If any operand bit has a value x, then the result is x.
- The operators + and – can also used as unary operators to represent signed numbers.
- Modulus operators produce the remainder from the division of two numbers.
- In Verilog HDL, 2's complement is used to represent negative numbers.

Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponent (power)
%	Modulus

Concatenation and Replication Operators

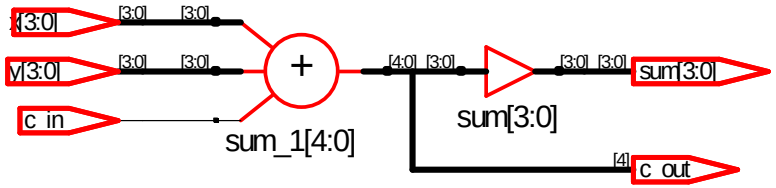
- Concatenation operators
  - The operands must be sized.
  - Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants.
  - Example: y = {a, b[0], c[1]};
- Replication operators
  - They specify how many times to replicate the number inside the braces.
  - Example: y = {a, 4{b[0]}, c[1]};

Symbol	Operation
{ , }	Concatenation
{const_expr{}}	Replication

A 4-bit Full Adder

```
module four_bit_adder(x, y, c_in, sum, c_out);
// I/O port declarations
input  [3:0] x, y;    // declare as a 4-bit array
input  c_in;
output [3:0] sum;   // declare as a 4-bit array
output c_out;

// Specify the function of a 4-bit adder.
   assign {c_out, sum} = x + y + c_in;
endmodule
```



A multiply and accumulate (MAC) unit

```
// an example to illustrate arithmetic operators
module multiplier_accumulator (x, y, z, result);
input  [7:0] x, y, z;
output [15:0] result;

   assign result = x * y + z;
endmodule
```

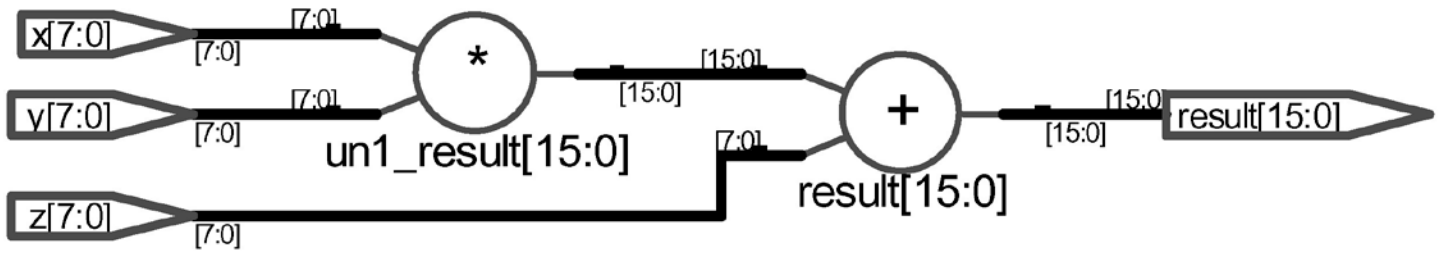


Figure 3.2: A multiply and accumulate unit.

Result size of an expression

```
// an example to illustrate arithmetic operators
module arithmetic_operators (a, b, e, c, d);
input  [3:0] a, b;
input  [6:0] e;
output [3:0] c;
output [7:0] d;
   assign c = a + b;
   assign d = a + b + e;
endmodule
```

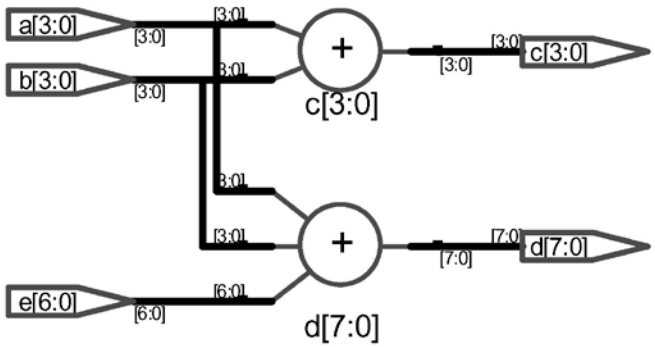


Figure 3.3: The synthesized result of the `arithmetic_operator` module.

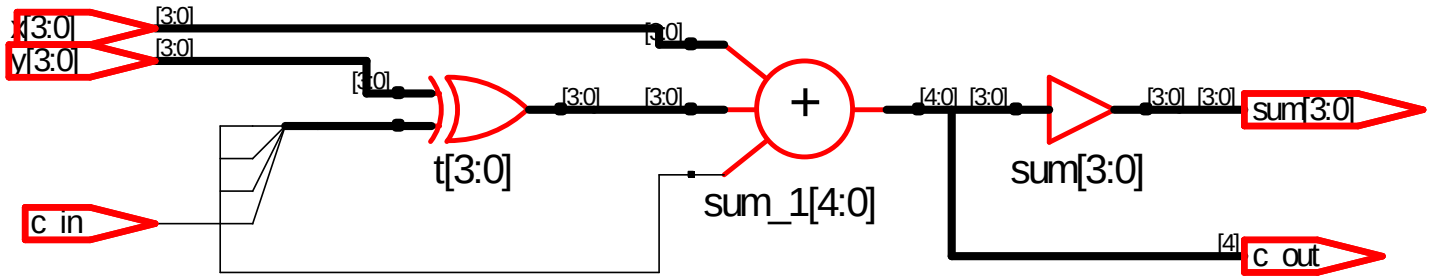
Mixed signed and unsigned operands

```
// an example to illustrate arithmetic operators
module arithmetic_operators (a, b, e, c, d);
Input signed  [3:0] a, b;
input  [6:0] e;
output signed [3:0] c;
output [7:0] d;
   assign c = a + b;
   assign d = a + b + e;
endmodule
```

A 4-bit two's complement adder

```
module twos_adder(x, y, c_in, sum, c_out);
// I/O port declarations
input    [3:0] x, y;    // declare as a 4-bit array
input    c_in;
output   [3:0] sum;   // declare as a 4-bit array
output   c_out;
wire     [3:0] t;       // outputs of xor gates

// Specify the function of a two's complement adder
   assign t = y ^ {4{c_in}};
   assign {c_out, sum} = x + t + c_in;
endmodule
```



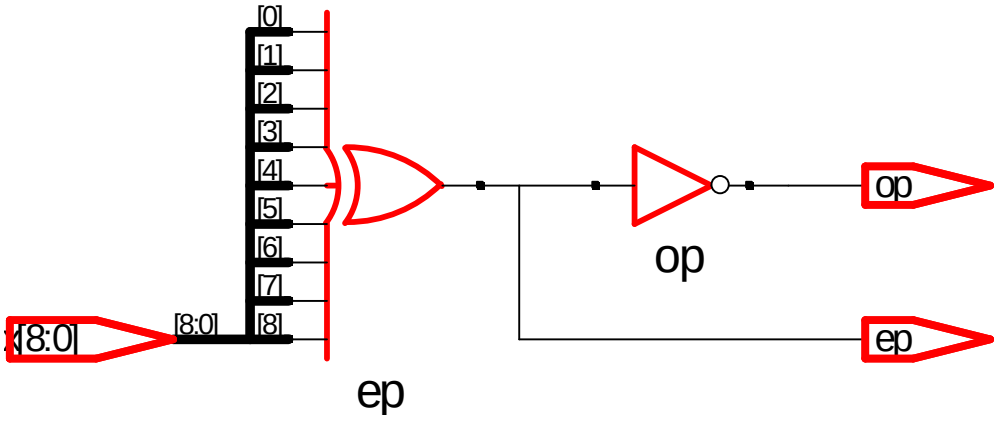
Reduction Operators

- Reduction operators
  - perform only on one vector operand.
  - carry out a bit-wise operation on a single vector operand and yield a 1-bit result.
  - work bit by bit from right to left.

Symbol	Operation
&	Reduction and
~&	Reduction nand
|	Reduction or
~|	Reduction nor
^	Reduction exclusive or
~^, ^~	Reduction exclusive nor

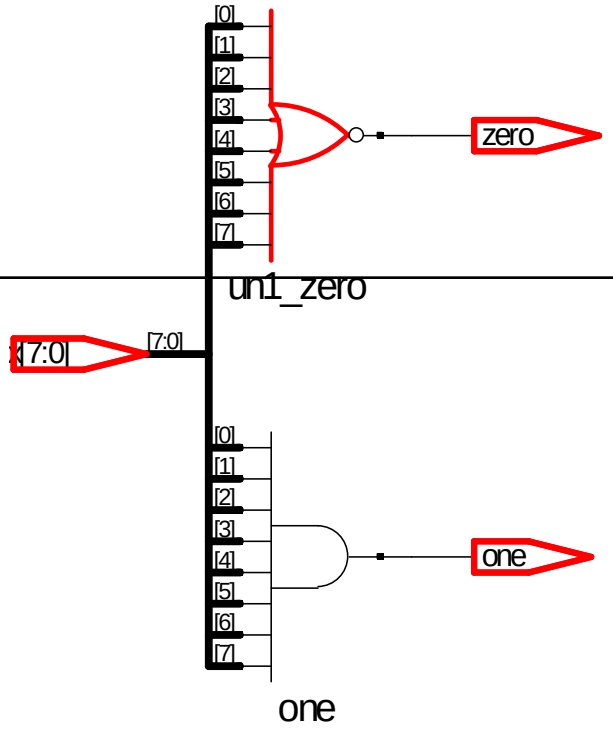
Reduction Operators --- A 9-Bit Parity Generator

```
module parity_gen_9b_reduction(x, ep,op);
// I/O port declarations
input    [8:0] x;
output   ep, op;
// dataflow modeling using reduction operator
assign ep = ^x;    // even parity generator
assign op = ~ep;  // odd parity generator
endmodule
```



Reduction Operators --- An All-Bit-Zero/One detector

```
module all_bit_01_detector_reduction(x, zero,one);
// I/O port declarations
input    [7:0] x;
output   zero, one;
// dataflow modeling
   assign zero = ~(|x);  // all-bit zero detector
   assign one = &x;      // all-bit one detector
endmodule
```



Logical Operators

- Logical operators
  - They always evaluate to a 1-bit value, 0, 1, or x.
  - If any operand bit is x or z, it is equivalent to x and treated as a false condition by simulators.

Symbol	Operation
!	Logical negation
&&	Logical and
||	Logical or

Relational Operators

❖ Relational operators

- They return logical value 1 if the expression is true and 0 if the expression is false.
- The expression takes a value x if there are any unknown (x) or z bits in the operands.

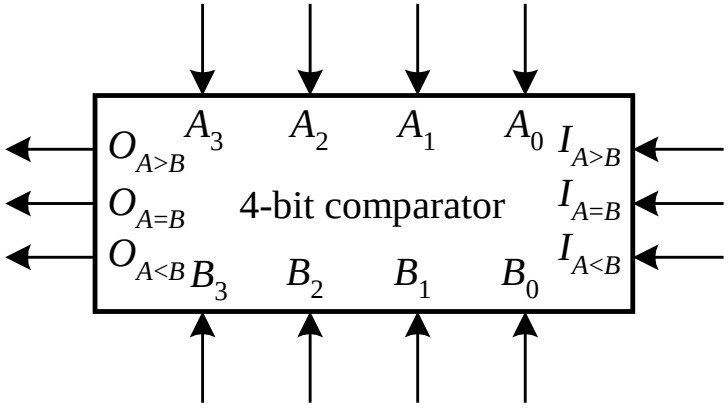
Symbol	Operation
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

Equality Operators

- Equality operators
  - compare the two operands bit by bit, with zero filling if the operands are of unequal length.
  - return logical value 1 if the expression is true and 0 if the expression is false.
- The operators (==, !=) yield an x if either operand has x or z in its bits.
- The operators (===, !==) yield a 1 if the two operands match exactly and 0 if the two operands not match exactly.

Symbol	Operation
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality

Relational Operators --- A 4-b Magnitude Comparator




```
module four_bit_comparator(Iagtb, Iaeqb, Ialtb, a, b, Oagtb, Oaeqb, Oaltb);
// I/O port declarations
input    [3:0] a, b;
input    Iagtb, Iaeqb, Ialtb;
output   Oagtb, Oaeqb, Oaltb;
// dataflow modeling using relation operators
   assign Oaeqb = (a == b) && (Iaeqb == 1);  // equality
   assign Oagtb = (a > b) || ((a == b)&& (Iagtb == 1));  // greater than
   assign Oaltb = (a < b) || ((a == b)&& (Ialtb == 1));   // less than
endmodule
```

Shift Operators

Symbol	Operation
>>	Logical right shift
<<	Logical left shift
>>>	Arithmetic right shift
<<<	Arithmetic left shift

- Logical shift operators
  - >> operator: logical right shift
  - << operator: logical left shift
  - The vacant bit positions are filled with zeros.
- Arithmetic shift operators
  - >>> operator: arithmetic right shift
    - The vacant bit positions are filled with the MSBs (sign bits).
  - <<< operator: arithmetic left shift
    - The vacant bit positions are filled with zeros.

Shift Operators

```
// example to illustrate logic and arithmetic shifts
module arithmetic_shift(x,y,z);
input  signed [3:0] x;
output [3:0] y;
output signed [3:0] z;
assign y = x >> 1;   // logical right shift
assign z = x >>> 1; // arithmetic right shift
endmodule
```

Note that: net variables x and z must be declared with the keyword signed. Replaced net variable with unsigned net (i.e., remove the keyword signed) and see what happens.

The Conditional Operator

❖ Conditional Operator

Usage: condition_expr ? true_expr: false_expr;

- The condition_expr is evaluated first.
- If the result is true then the true_expr is executed; otherwise the false_expr is evaluated.

```
if  (condition_expr) true_expr;
else false_expr;
```

- a 2-to-1 multiplexer

```
assign out = selection ? in_1: in_0;
```

The Conditional Operator --- A 4-to-1 MUX

```
module mux4_to_1_cond (i0, i1, i2, i3, s1, s0, out);
// Port declarations from the I/O diagram
input   i0, i1, i2, i3;
input   s1, s0;
output out;
// Using conditional operator (?:)
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
endmodule
```