

---

## Lecture 4: HDL III

ECE 228

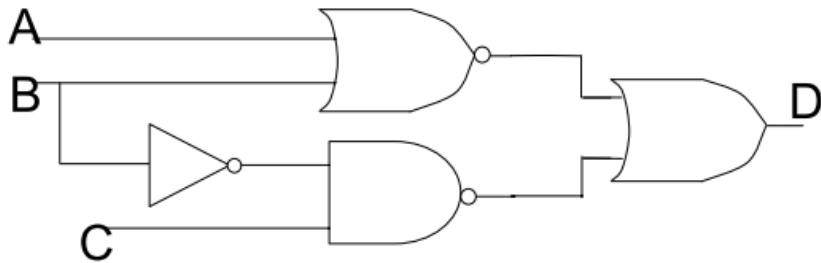
Mostafizur Rahman  
rahmanmo@umkc.edu

# Critical Thinking Exercise

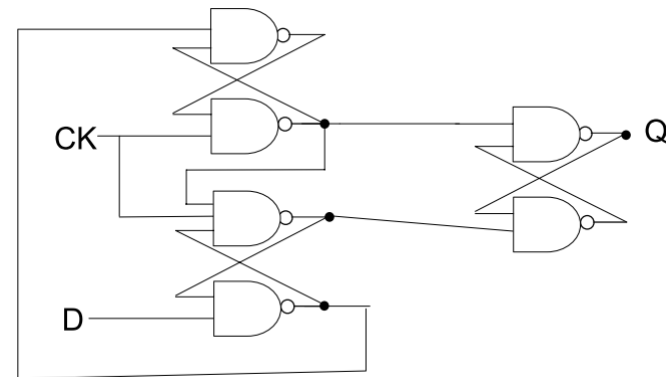
Price	\$799.99	\$478.29	\$430.99	\$579.99
Shipping	\$6.99			
Sold By	<a href="#">Free city (We Record S/N)</a>	Amazon.com	<a href="#">52 WEEKS DEAL</a>	Amazon.com
computer memory size	8 GB	8 GB	8 GB	8 GB
Processor (CPU) Manufacturer	Intel	AMD	AMD	Intel
Processor (CPU) Speed	1.6 GHz	3.6 GHz	2 GHz	4.2 GHz
Display Resolution Max	1920 x 1080 pixels	1920 x 1080 pixels	1920 x 1080 pixels	2040 x 1380
Display Size	15.6 in	15.6 in	15.6 in	15.6 in
Display Technology	—	—	HD	LED
Hard Disk Size	1,000 GB	256 GB	1,000 GB	256 GB (SSD)
Item Dimensions	14.2 x 9.6 x 0.8 in	14.1 x 9.1 x 0.75 in	10.15 x 14.96 x 0.78 in	14.31 x 9.86 x 0.71 in
Item Weight	3.7 lbs	3.5 lbs	4.8 lbs	—
Operating System	Windows 10 Home	Windows 10 Home	Windows 10 Home	Windows 10 Home
Processor Count	8	4	8	4
System RAM Type	DDR4 SDRAM	DDR4 SDRAM	DDR4 SDRAM	DDR4 SDRAM
Wireless Compatibility	802.11a/b/g/n/ac	802.11b/n/ac, Bluetooth	—	802.11ax

Alex, 23year old college student, wants to buy a computer primarily to complete his school assignments (document work, pspice simulations, virtual lab connections, etc.) Budget \$500ish

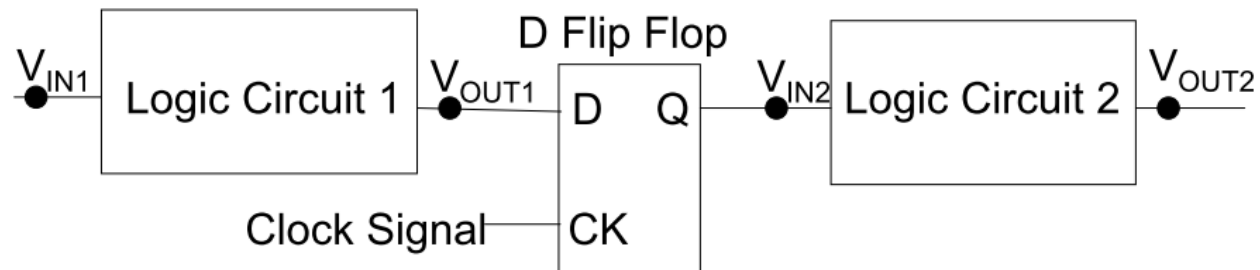
# Critical Thinking Exercise



Which one is the worst path for propagation delay?



How about this?



# Critical Thinking Exercise

---

```
module test(a, b, c, d, out );
```

```
    input  a, b, c, d;
```

```
    output out;
```

```
    wire [1:0] t0, t1, t2;
```

```
    assign t0  = (a & c);
```

```
    assign t1  = (a & b);
```

```
    assign t2  = (b & c);
```

```
    out = (t0 & t1 & t2);
```

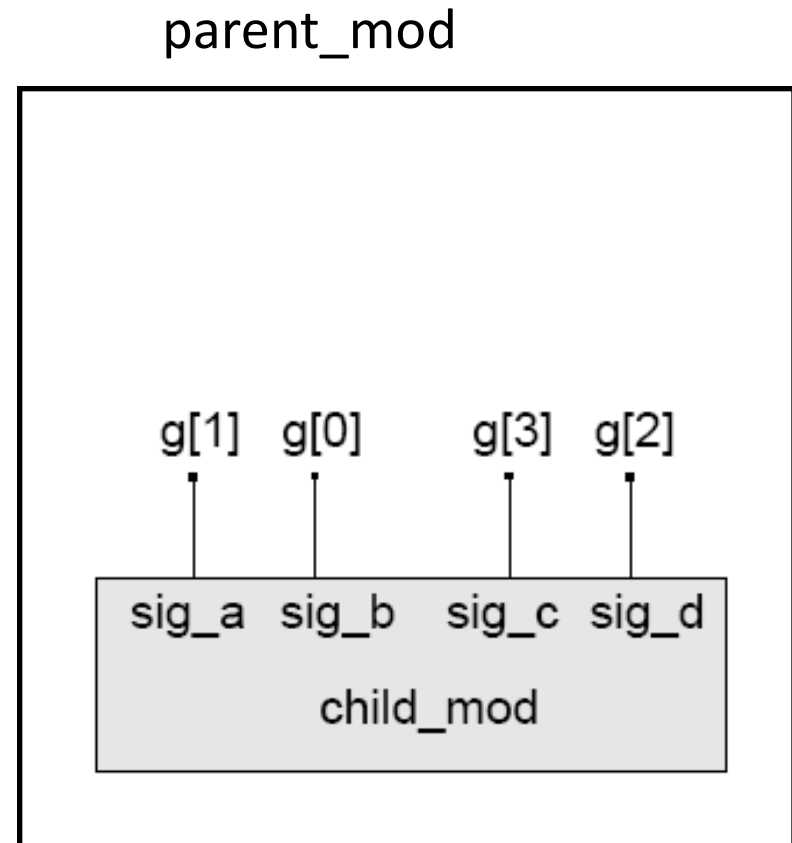
```
endmodule
```

## Type of Port Connections (cont.)

- Connection by Name

```
module child_mod (sig_a, sig_b,  
sig_c, sig_d);  
  input    sig_a, sig_b;  
  output   sig_c, sig_d;  
  
  // module description goes here.  
  
endmodule
```

```
module parent_mod;  
  wire [3:0] g;  
  // Listed order not significant  
  
  child_mod G1 ( .sig_c(g[3]),  
                 .sig_d(g[2]),  
                 .sig_b(g[0]),  
                 .sig_a(g[1]));  
  
endmodule
```



# Dataflow style

---

assign A = X | (Y & ~Z);

use of Boolean operators  
(~ for bit-wise, ! for logical negation)

assign B[3:0] = 4'b01XX;

bits can take on four values  
(0, 1, X, Z)

assign C[15:0] = 4'h00ff;

variables can be n-bits wide  
(MSB:LSB)

assign #3 {Cout, S[3:0]} = A[3:0] + B[3:0] + Cin;

use of arithmetic operator

multiple assignment (concatenation)

delay of performing computation, only used by simulator, not synthesis

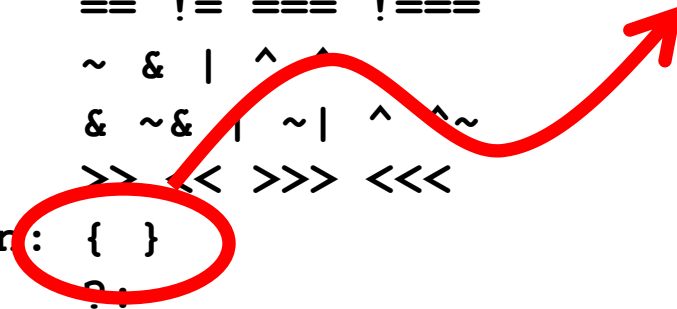
# Dataflow : Key Points

---

- Dataflow modeling enables the designer to focus on where the state is in the design and how the data flows between these state elements without becoming bogged down in gate-level details
  - Continuous assignments are used to connect combinational logic to nets and ports
  - A wide variety of operators are available including:

Arithmetic:	+ - * / % **
Logical:	! &&
Relational:	> < >= <=
Equality:	== != === !==
Bitwise:	~ &   ^
Reduction:	& ~&   ~  ^ ^~
Shift:	>> << >>> <<<
Concatenation:	{ }
Conditional:	?:

assign signal[3:0]  
= { a, b, 2'b00 }



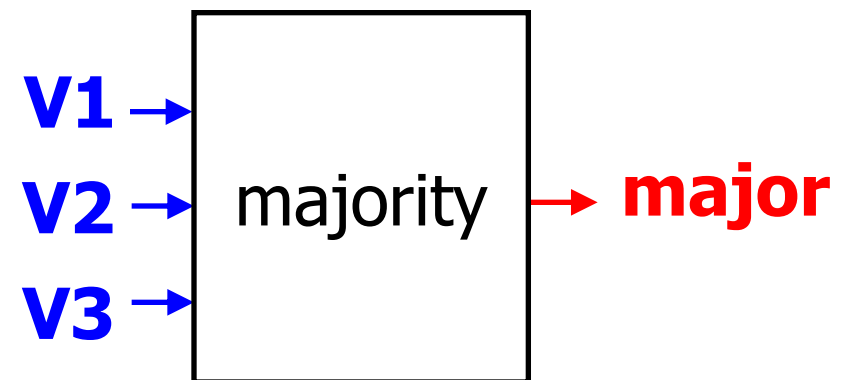
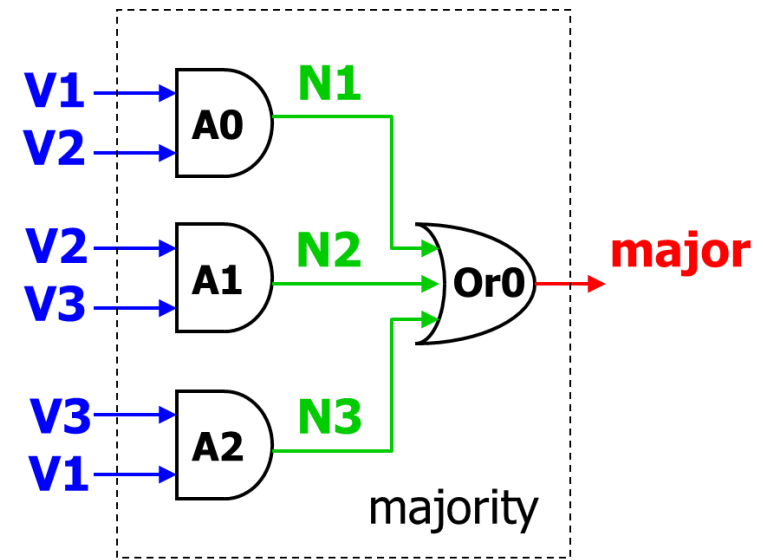
# Dataflow Example: Majority Detector

- Use continuous assignment statements to assign Boolean expressions to signals.
- If an input value changes, the value of the assignment is immediately updated. This is combinational *hardware*, not *software*.

```
module majority (major, V1, V2, V3) ;
```

```
output major ;  
input V1, V2, V3 ;
```

```
assign major = (V1 & V2)  
               | (V2 & V3)  
               | (V1 & V3);  
endmodule
```





# Lexical Conventions

---

- **Modules** can be on separate files
  - Example: module MOD1;
  - File Names: You can save them as anything but recommend to save it as its module name Example mod1.v
- **Logic Values**
  - The Verilog HDL has 4 logic values system:

Logic Value	Description
<b>0</b>	zero, low, or false
<b>1</b>	one, high, or true
<b>z</b> or <b>Z</b>	high impedance (tri-stated or floating)
<b>x</b> or <b>X</b>	unknown or uninitialized

# Lexical Conventions

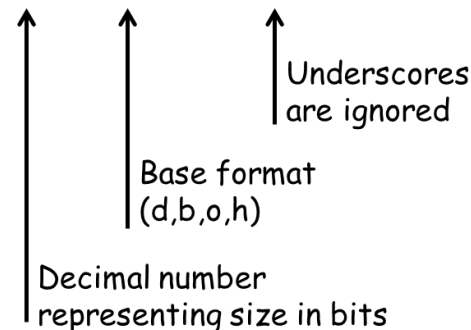
## ■ Numeric Literals

### – Literal Integer Numbers

- Syntax: **size**'**base** **value** Sized integer in a specific radix (base)
  - **size** (*optional*) is the number of bits in the number. Unsized integers default to at least 32-bits.
  - '**base** (*optional*) represents the radix. The default base is decimal.

Base	Symbol	Legal Values
binary	<b>b</b> or <b>B</b>	0, 1, x, X, z, Z, ?, _
octal	<b>o</b> or <b>O</b>	0-7, x, X, z, Z, ?, _
decimal	<b>d</b> or <b>D</b>	0-9, _
hexadecimal	<b>h</b> or <b>H</b>	0-9, a-f, A-F, x, X, z, Z, ?, _

4' b10\_11



# Lexical Conventions

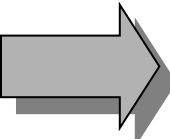
- The **?** is another way of representing the **Z** logic value.
- An **\_** (underscore) is ignored (used to enhance readability).
- Values are expanded from right to left (lsb to msb).
- When size is less than value, the upper bits are truncated.
- When size is larger than value, and the left-most bit of value is 0 or 1, zeros are left-extended to fill the size.
- When size is larger than value, and the left-most bit of value is Z or X, the Z or X is left-extended to fill the size.

Examples			
<b>10</b>	unsized	decimal	0...01010 (32-bits)
<b>'o7</b>	unsized	octal	0...00111 (32-bits)
<b>1'b1</b>	1 bit	binary	1
<b>8'Hc5</b>	8 bits	hex	11000101
<b>6'hF0</b>	6 bits	hex	110000 (truncated)
<b>6'hF</b>	6 bits	hex	001111 (zero filled)
<b>6'hZ</b>	6 bits	hex	ZZZZZZ (Z filled)

# Logical Operators

---

- Operate on Boolean operands. Operands may be a net, register or expression are treated as unsigned
- `& &` → logical AND
- `| |` → logical OR
- `!` → logical NOT
- Operands evaluated to ONE bit value: *0*, *1* or *x*
- Result is ONE bit value: *0*, *1* or *x*

<code>A = 6;</code>		<code>A &amp;&amp; B → 1 &amp;&amp; 0 → 0</code>
<code>B = 0;</code>		<code>A     !B → 1     1 → 1</code>
<code>C = x;</code>		<code>C     B → x     0 → x</code>

but `C&&B=0`

# Bitwise Operators (i)

---

- Operation on bit by bit basis

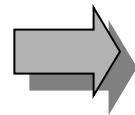
- Standard operations include:

- $\&$  → bitwise AND
- $|$  → bitwise OR
- $\sim$  → bitwise NOT
- $\wedge$  → bitwise XOR
- $\sim \wedge$  or  $\wedge \sim$  → bitwise XNOR
- $\sim \langle \text{op} \rangle$  Bitwise not
- $\langle \text{op} \rangle \ \& \ \langle \text{op} \rangle$  Bitwise and
- $\langle \text{op} \rangle \ | \ \langle \text{op} \rangle$  Bitwise or
- $\langle \text{op} \rangle \ \wedge \ \langle \text{op} \rangle$  Bitwise xor
- $\langle \text{op} \rangle \ \sim \wedge \ \langle \text{op} \rangle$  Bitwise xnor
- $\langle \text{op} \rangle \ \wedge \sim \ \langle \text{op} \rangle$  Bitwise xnor

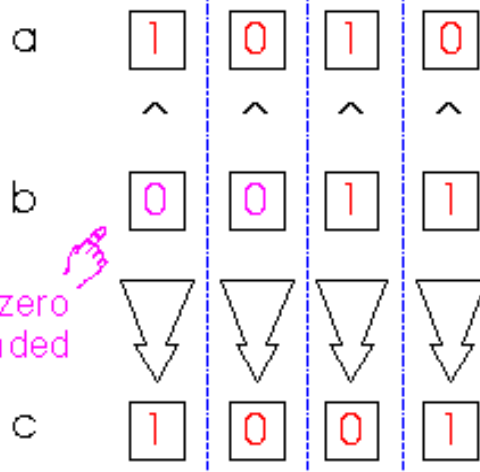
- If the operands do not have the same size, the shorter word is extended with 0 padding

# Bitwise Operators (ii)

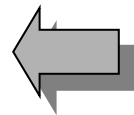
- $a = 4'b1010;$   
 $b = 4'b1100;$



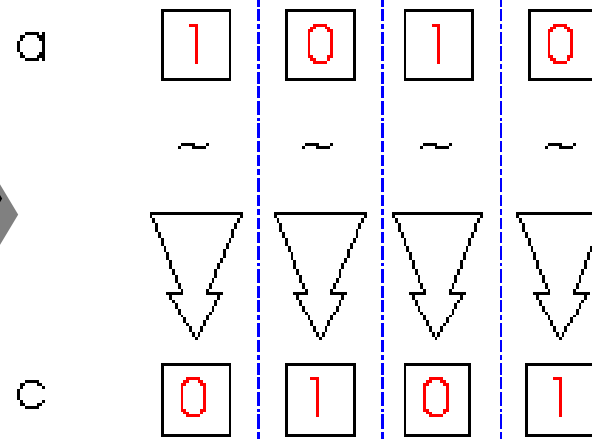
$c = a \wedge b;$



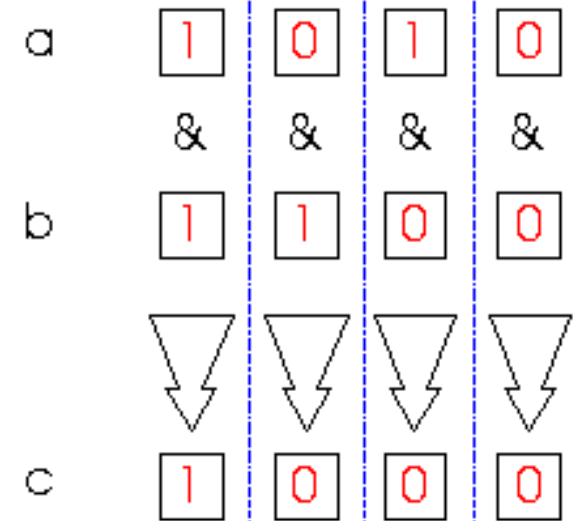
zero  
extended



$c = \sim a;$



$c = a \& b;$



- $a = 4'b1010;$   
 $b = 2'b11;$

# Reduction Operators

---

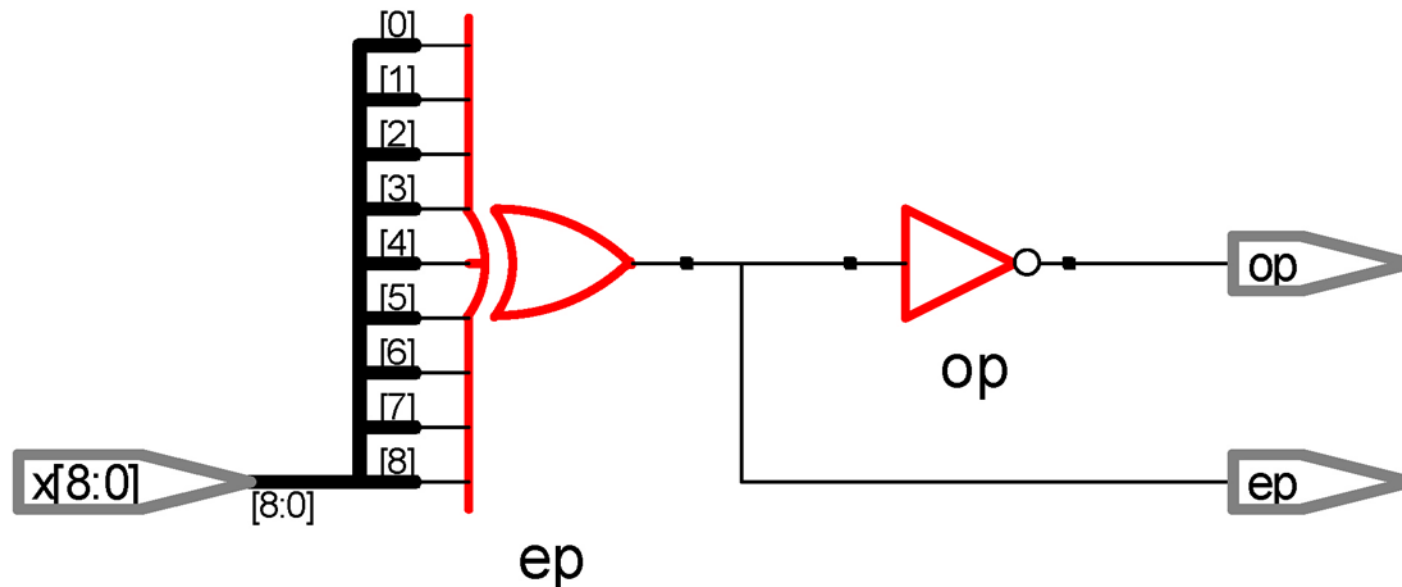
- These are unary operators which create a single bit value for a data word of multiple bits
  - One multi-bit operand → One single-bit result
  - $\&$  → AND
  - $|$  → OR
  - $\wedge$  → XOR
  - $\sim \&$  → NAND
  - $\sim |$  → NOR
  - $\sim \wedge$  or  $\wedge \sim$  → XNOR
- Example
  - $\& \text{ <op>}$  and-reduction
  - $| \text{ <op>}$  or-reduction
  - $\wedge \text{ <op>}$  xor-reduction

`a = 4'b1001;`

`c = |a; // c = 1|0|0|1 = 1`

# A 9-Bit Parity Generator

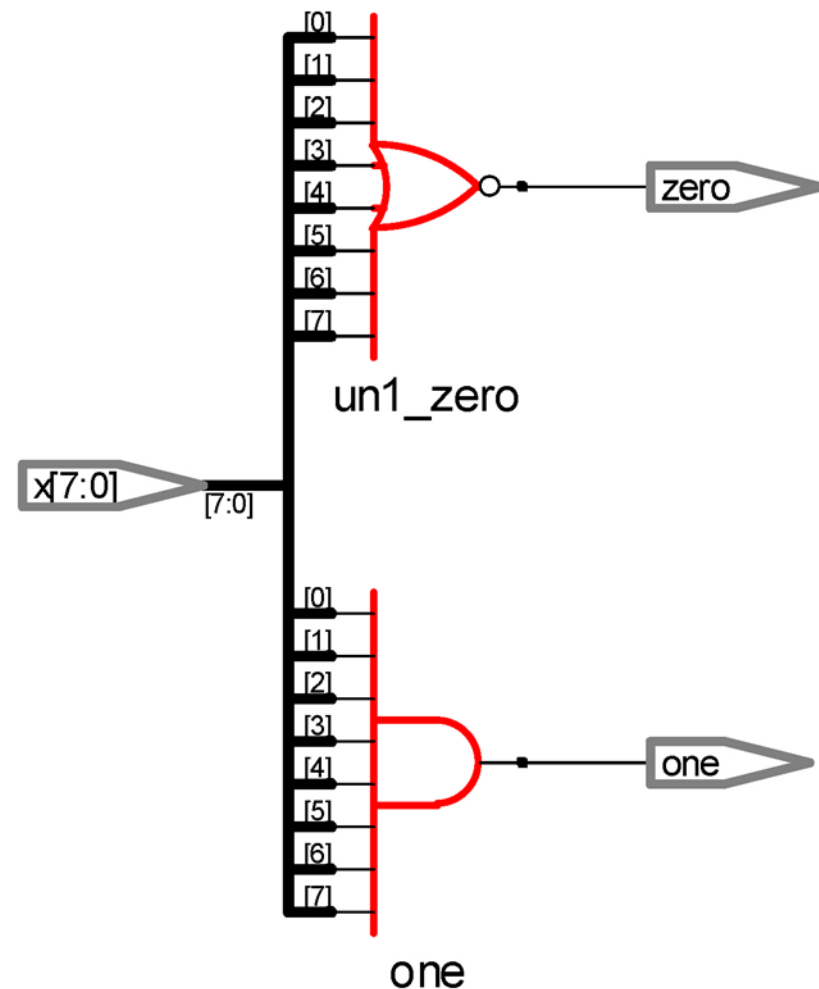
```
// dataflow modeling using reduction operator  
assign ep = ^x; // even parity generator  
assign op = ~ep; // odd parity generator
```





# An All-Bit-Zero/One Detector

```
// dataflow modeling  
assign zero = ~(|x); // all-bit zero  
assign one = &x;    // all-bit one
```



# Shift Operators

---

- `>>` → shift right
- `<<` → shift left
- Result is same size as first operand, always zero filled

```
a = 4'b1010;
```

```
...
```

```
d = a >> 2;    // d = 0010
```

```
c = a << 1;    // c = 0100
```

# Concatenation Operator

---

- Forms a single operand from two or more operands, and is useful for forming logical buses.
- {op1, op2, ..} → concatenates op1, op2, .. to single number

```
reg a;  
reg [2:0] b, c;  
..  
a = 1'b 1;  
b = 3'b 010;  
c = 3'b 101;  
catx = {a, b, c};           // catx = 1_010_101  
caty = {b, 2'b11, a};       // caty = 010_11_1
```

## Replication { <const> {op} }

```
catr = {4{a}, b, 2{c}};     // catr =  
1111_010_101101           19
```

# Relational Operators

---

- The operators compare operands and produce a Boolean results (true or false)
  - $>$  → greater than
  - $<$  → less than
  - $>=$  → greater or equal than
  - $<=$  → less or equal than
- Result is one bit value: *0*, *1* or *x*
- If the operands are net or registers, their values are treated as unsigned
  - $1 > 0 \rightarrow 1'b1$
  - $3'b111 < 3'b001 \rightarrow 1'b0$

# Equality Operators

---

- `==` → logical equality
  - `!=` → logical inequality
  - `===` → case equality (identical)
  - `!==` → case inequality (not identical)
- Return *0*, *1* or *x*
- Return *0* or *1*

— `4'b 1z0x == 4'b 1z0x` → **x**

— `4'b 1z0x != 4'b 1z0x` → **x**

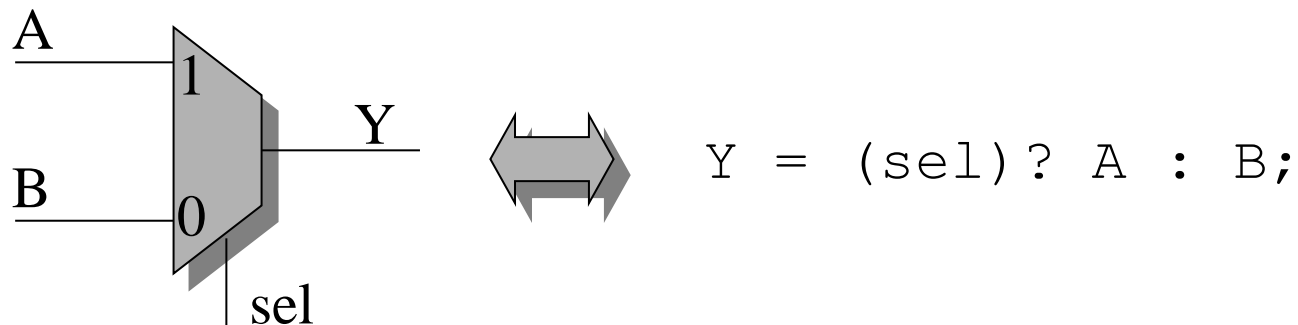
— `4'b 1z0x === 4'b 1z0x` → **1**

— `4'b 1z0x !== 4'b 1z0x` → **0**

# Conditional Operator

---

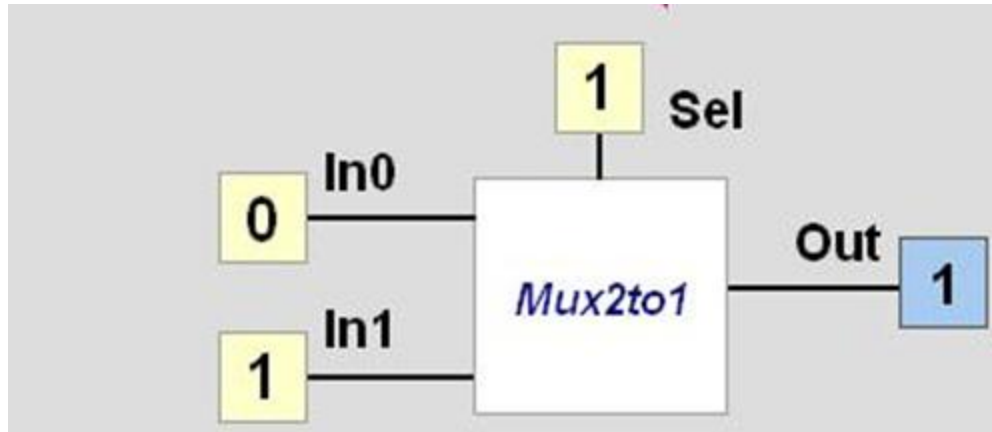
- `cond_expr ? true_expr : false_expr`
  - If `cond_expr` is TRUE, then the result is `true_expr`
  - If the `cond_expr` is unknown, then result is X's
  - Otherwise, result is `false_expr`
- Like a 2-to-1 mux ..



## 2-to-1 MUX - Continuous Assignment

---

- ❑ A conditional assignment has three signals
  - The first signal is the control signal
  - If the control signal is true, the second signal is assigned to the left hand side (LHS) signal ; otherwise, the third signal is assigned to LHS signal.

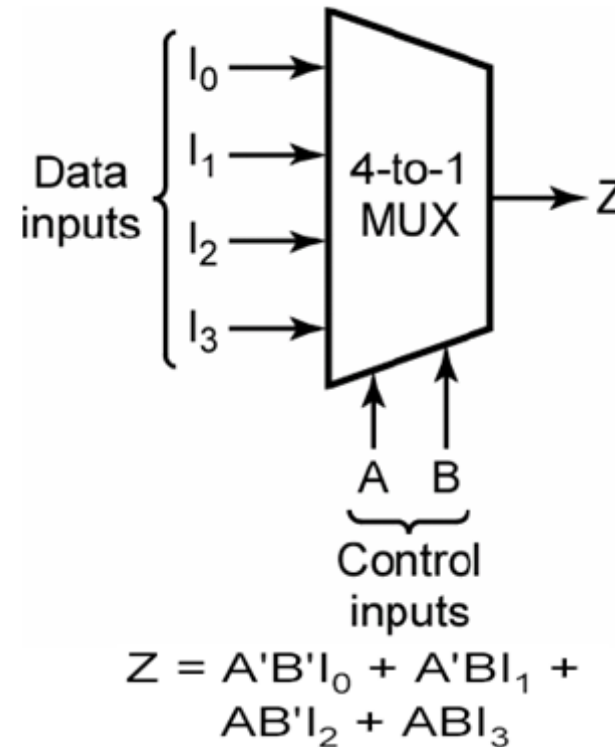


```
module Mux2to1(Out, In0, In1, Sel);  
output Out;  
input In0, In1, Sel;  
  
assign Out = Sel ? In1 : In0;  
  
endmodule
```

# An Example - A 4-to-1 MUX

// using conditional operator (?:)


assign Z = A ? ( B ? I<sub>3</sub> : I<sub>2</sub>) : (B ? I<sub>1</sub> : I<sub>0</sub>) ;





# Operator Precedence

---

<code>+ - ! ~ unary</code>	highest precedence
<code>* / %</code>	
<code>+ - (binary)</code>	
<code>&lt; &lt; &gt; &gt;</code>	
<code>&lt; &lt;= &gt; &gt; &gt;</code>	
<code>== != === !==</code>	
<code>&amp; ~ &amp;</code>	
<code>^ ^~ ~^</code>	
<code>  ~  </code>	
<code>&amp; &amp;</code>	
<code>   </code>	
<code>?: conditional</code>	lowest precedence

Use parentheses to  
enforce your  
priority

# Verilog Operators

Verilog Operator	Name	Functional Group
()	bit-select or part-select	
()	parenthesis	
! ~ &   ~& ~  ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	Logical Bit-wise Reduction Reduction Reduction Reduction Reduction Reduction
+ -	unary (sign) plus unary (sign) minus	Arithmetic Arithmetic
{}	concatenation	Concatenation
{{}}	replication	Replication
* / %	multiply divide modulus	Arithmetic Arithmetic Arithmetic
+ -	binary plus binary minus	Arithmetic Arithmetic
<< >>	shift left shift right	Shift Shift

Verilog Operator	Name	Functional Group
> >= < <=	greater than greater than or equal to less than less than or equal to	Relational Relational Relational Relational
== !=	logical equality logical inequality	Equality Equality
=== !==	case equality case inequality	Equality Equality
&	bit-wise AND	Bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	Bit-wise Bit-wise
	bit-wise OR	Bit-wise
&&	logical AND	Logical
	logical OR	Logical
?:	conditional	Conditional

# Reserved Keywords

---

always	endmodule	large	reg	tranif0
and	endprimitive	macromodule	release	tranif1
assign	endspecify	nand	repeat	tri
attribute	endtable	negedge	rnmos	tri0
begin	endtask	nmos	rpmos	tri1
buf	event	nor	rtran	triand
bufif0	for	not	rtranif0	trior
bufif1	force	notif0	rtranif1	trireg
case	forever	notif1	scalared	unsigned
casex	fork	or	signed	vectored
casez	function	output	small	wait
cmos	highz0	parameter	specify	wand
deassign	highz1	pmos	specparam	weak0
default	if	posedge	strength	weak1
defparam	ifnone	primitive	strong0	while
disable	initial	pull0	strong1	wire
edge	inout	pull1	supply0	wor
else	input	pulldown	supply1	xnor
end	integer	pullup	table	xor
endattribute	join	rcmos	task	
endcase	medium	real	time	
endfunction	module	realtime	tran	

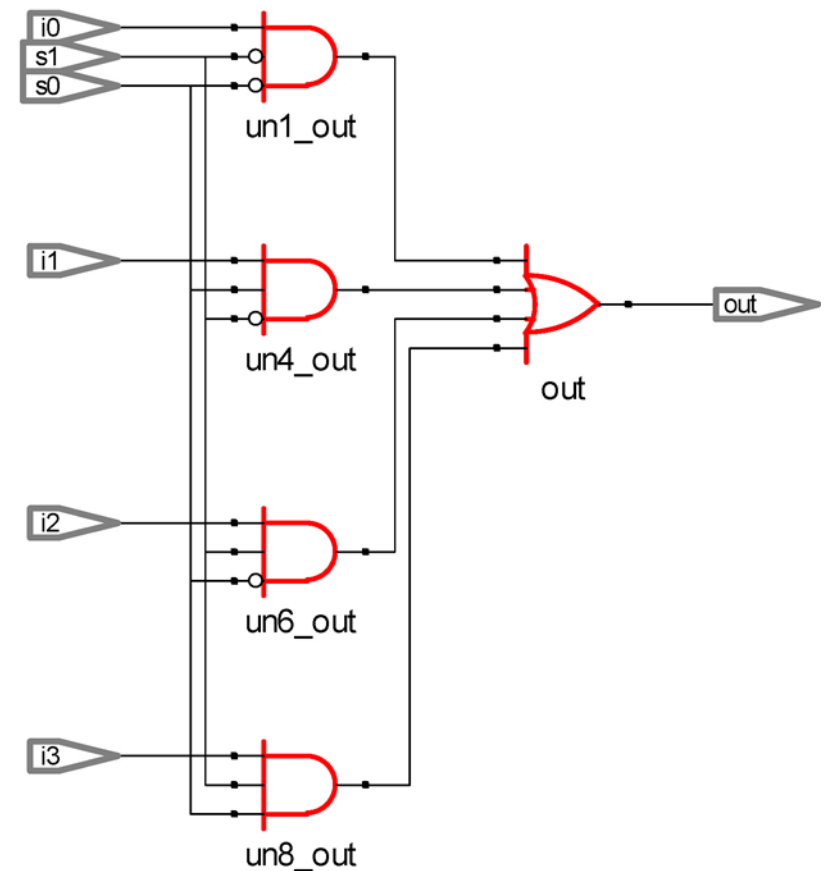
---

# Examples

# An Example - A 4-to-1 MUX

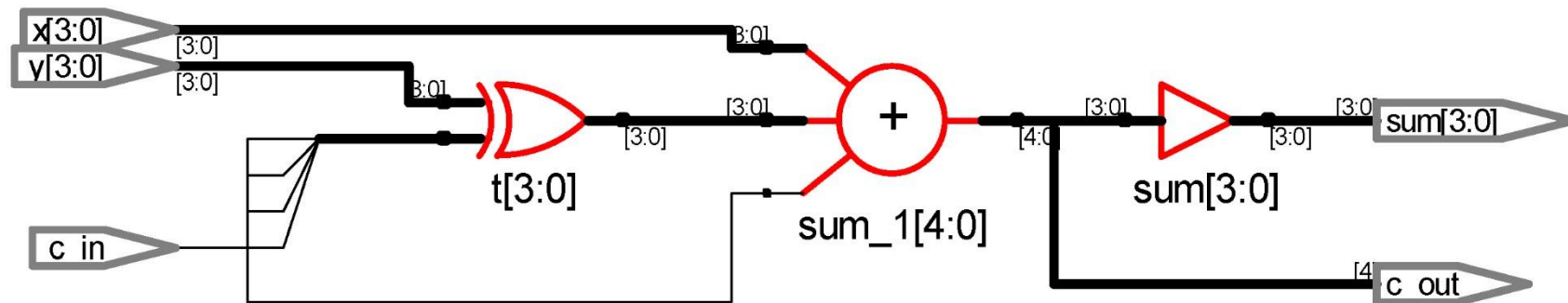
// using basic and, or , not logic operators

```
assign out = (~s1 & ~s0 & i0) |  
             (~s1 & s0 & i1) |  
             (s1 & ~s0 & i2) |  
             (s1 & s0 & i3);
```

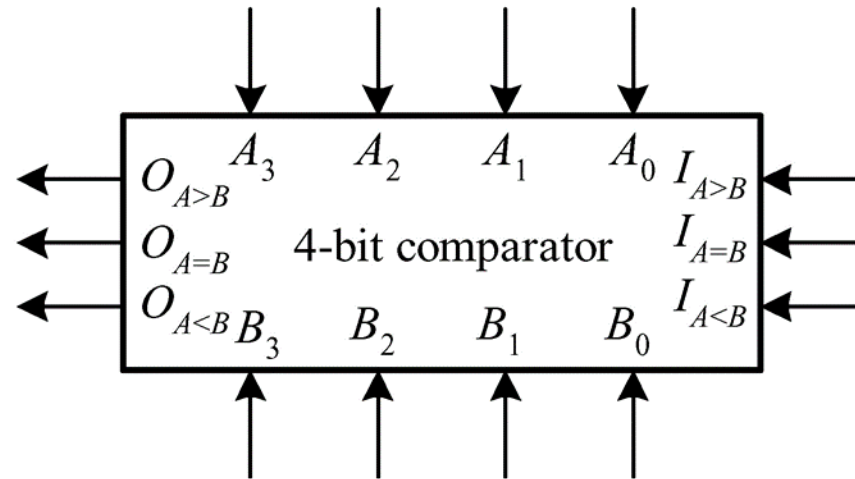


# A 4-Bit Two's Complement Adder

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



# A 4-B Magnitude Comparator



// dataflow modeling using relation operators

assign Oaeqb = (A == B) && (Iaeqb == 1); // =

assign Oagtb = (A > B) || ((A == B) && (Iagtb == 1)); // >

assign Oaltb = (A < B) || ((A == B) && (Ialtb == 1)); // <