Digital IC Design

Session 6: Data Flow Modeling and Operators

2024 | Nineplus Infotech X Anseong Polytechnic University
Sr Research Engineer, William Woo

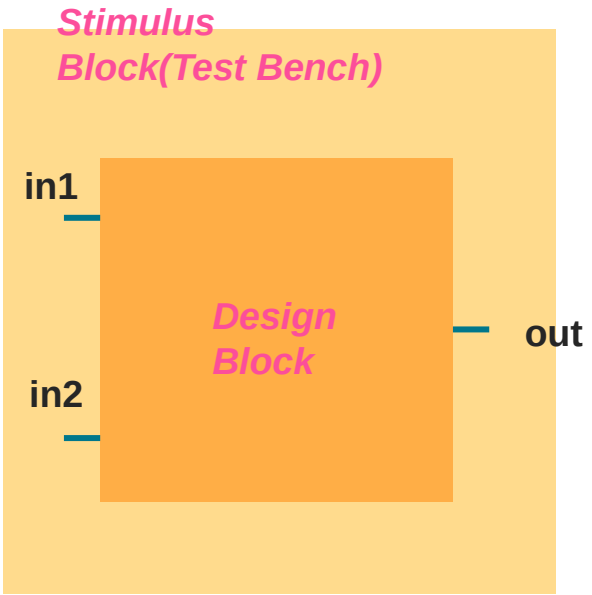
cadence®

1. Port

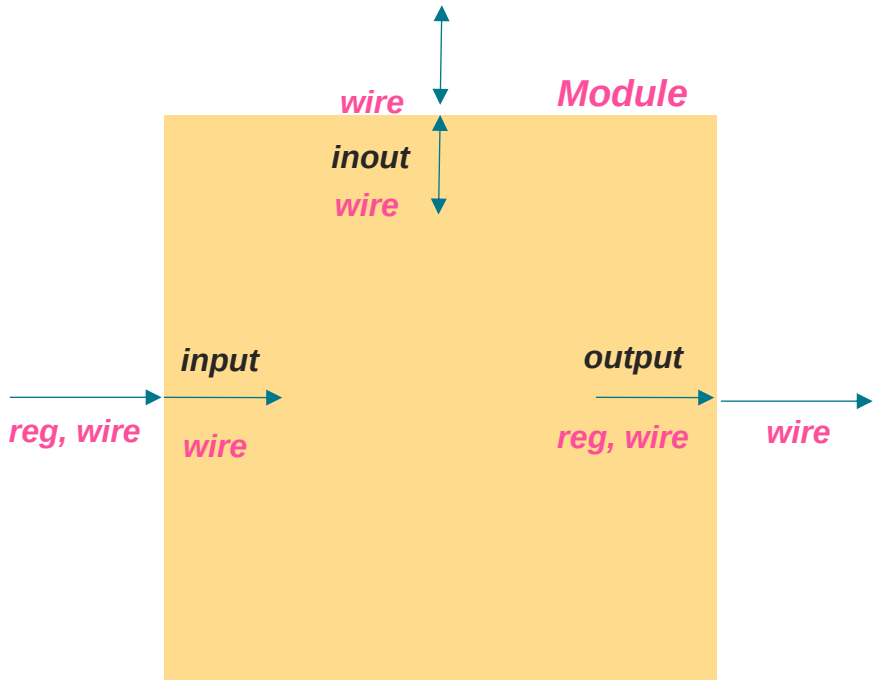
Ports

- Ports
  - 모듈이 외부의 환경과 소통할 수 있는 인터페이스를 제공함
  - Stimulus Block은 Port를 가지지 않음(필요 없음)

- input: input port
- output: output port
- inout: bi-directional port



Port Declaration Principle



모듈을 기준으로,

1. input port는 wire
2. output port에 연결되는 신호는 wire
3. inout port는 wire, inout에 연결된 신호도 wire

- wire
  - Continuously driven by other signal
  - Connections between HW elements or module

- reg
  - Retain value until another value is stored

Connecting Ports to External

```
module ripple_carry_counter(q, clk, reset);
output [3:0] q;
input clk, reset;
T_FF tff0(q[0], clk, reset);
T_FF tff1(q[1], q[0], reset);
T_FF tff2(q[2], q[1], reset);
T_FF tff3(q[3], q[2], reset);
endmodule
```

```
module T_FF(q, clk, reset);
output q;
input clk, reset;
wire d;
D_FF dff0(q, d, clk, reset);
not n1(d, q);
endmodule
```

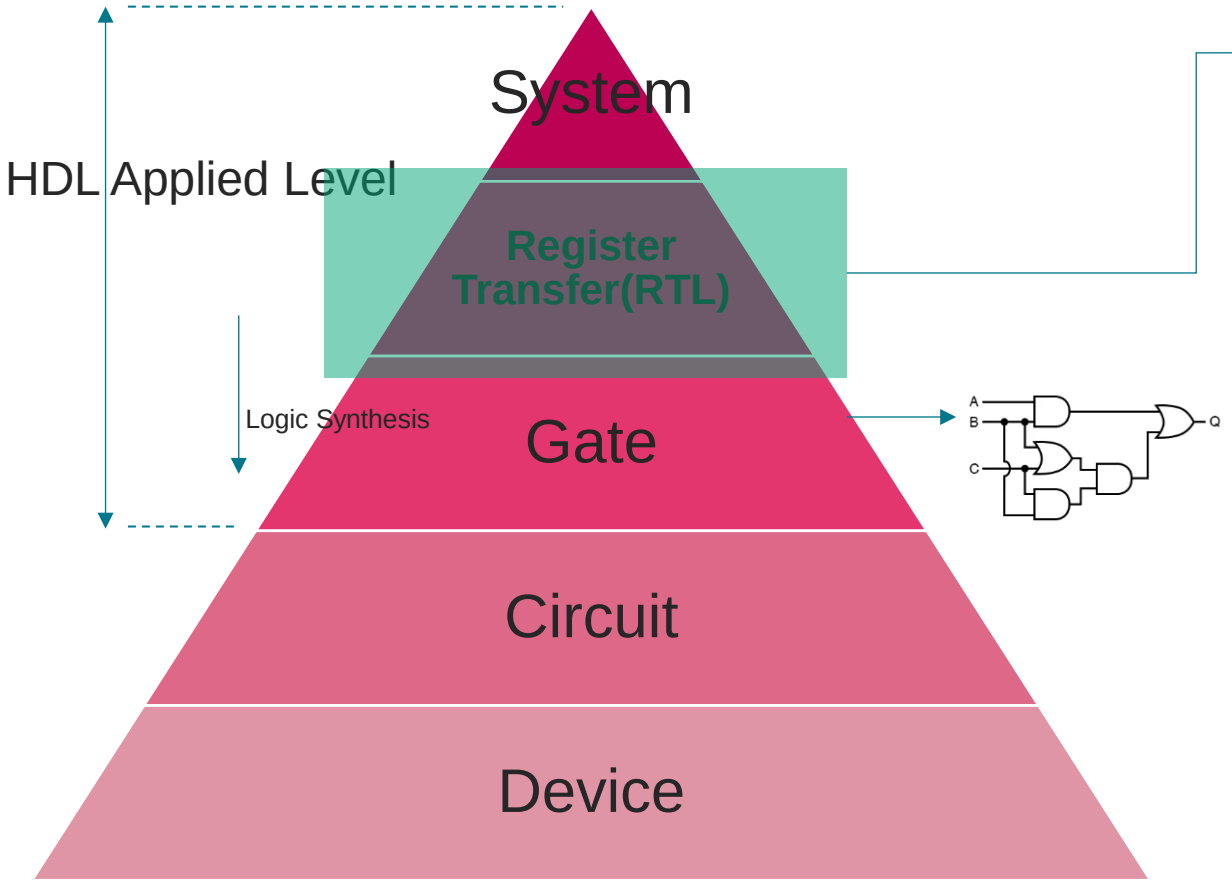
Connecting by Ordered List

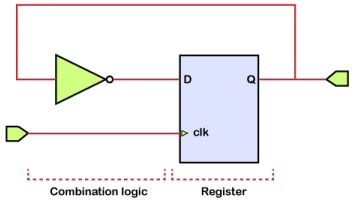
```
module stimulus;
reg clk; // Input
reg reset; // Input
wire [3:0] q; // Output

ripple_carry_counter r1 (.q(q), .clk(clk), .reset(reset));
initial
clk = 1'b0; // Set clk to 0
always
#5 clk = ~clk; // Toggle clk every 5 time units
initial
begin
reset = 1'b1;
#25 reset = 1'b0;
#180 reset = 1'b1;
#10 reset = 1'b0;
#20 $finish;
end

initial
$monitor($time, "Output q = %d", q);
endmodule
```

Ordered connection by Name

2. Dataflow Modeling

Review and Elaborate!





- **Dataflow Modeling**
  - Register간의 데이터 흐름에 집중
  - 개별 Gate의 instantiation 보다는 모듈이 데이터를 어떻게 처리하는가에 집중
  - 전달함수의 표현에 집중 → 효율성
- Logic Synthesis
  - EDA Tool은 Dataflow model을 Gate Level로 변환
- RTL(Register Transfer Level) Design
  - Dataflow Modeling + Behavioral Modeling

[What is Logic Synthesis?]

Assign: driving value[s] onto a net

…Dataflow modeling describes the design in terms of expressions instead of primitive gates… → 데이터의 흐름 = 수식

Continuously driving → Always active:
as soon as RHS changes, LHS changes.

Standard or
Canonical Form

assign signal = **expression(operand + operator)**

wire scalar or vector

wire or reg

Basic examples

Translate the below code into circuits!

```
...

// Example1
assign out = in1 & in2;


// Example2
assign addr[15:0] = addr1[15:0] ^ addr2[15:0];


// Example3
assign {carry_out, sum[3:0]} = in1[3:0] + in2[3:0] + carry_in;

...
```

Expression: operand and operator combined

Expression: 데이터의 흐름을 기술. Operand와 Operator의 결합

Operand: RHS에서 변수. reg | wire 둘 다 가능. **Type Casting**

Operator: expression의 결과를 얻기 위한 operand의 연산

- **Drive desired logic result**
- **Produce logic circuit**

3. Operators

Overview (1)

Operator Type	Symbol	Function	Num of Operand
Arithmetic	*	Multiply	2
	/	Divide	2
	+	Add	2
	-	Subtract	2
	%	Modulus	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

Overview (2)

Operator Type	Symbol	Function	Num of Operand
Equality	==	Equality	2
	!=	Inequality	2
	===	Case equality	2
	!==	Case inequality	2
Bitwise	~	Bitwise negation	1
	&	Bitwise and	2
	|	Bitwise or	2
	^	Bitwise nor	2
	^~ OR ~^	Bitwise xnor	2

Overview (3)

Operator Type	Symbol	Function	Num of Operand
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
Concatenation	{ }	Concatenation	All
Replication	{ { } }	Replication	All
Conditional	? :	Conditional	3

Operator Precedence

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

연산자 우선순위의 모호함을 피하기 위해서는 괄호(parenthesis)를 사용하는 습관을 가지자

Operator(1): Arithmetic Operator

Operator	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

```
module arithmetic;

reg [7:0] foo;
reg [7:0] bar;

initial
begin

foo = 45;
bar = 9;

$display("foo + bar = %d", foo + bar);
$display("foo - bar = %d", foo - bar);
$display("foo x bar = %d", foo * bar);
$display("foo / bar = %d", foo / bar);
$display("foo %% bar = %d", foo % bar);
$display("foo^2 = %d", bar ** 2);

end


endmodule
```

Operator(2): Conditional Operator

Operator	Description
A < B	A smaller than B
A > B	A greater than B
A <= B	A smaller than or equal to B
A >= B	A greater than or equal to B

- True: return 1
- False: return 0

```
module conditional;

reg [7:0] foo;
reg [7:0] bar;

initial begin
    /* case 1 */
    foo = 45;
    bar = 9;
    if (foo >= bar) begin
        $display("foo >= bar");
    end else begin
        $display("foo < bar");
    end

    /* case 2 */
    foo = 45;
    bar = 45;
    if (foo <= bar) begin
        $display("foo <= bar");
    end else begin
        $display("foo <bar");
    end

    /* case 3 */
    foo = 9;
    bar = 8;
    if (foo > bar) begin
        $display("foo > bar");
    end else begin
        $display("foo <= bar");
    end

    /* case 4 */
    foo = 22;
    bar = 22;
    if (foo < bar) begin
        $display("foo < bar");
    end else begin
        $display("foo >= bar");
    end
end
endmodule
```

Operator(3): Equality Operator

Operator	Description	Results
A == B	A equal to B | result unknown if **x** or **z** in operand[s]	**0, 1, x**
A != B	A not equal to B | result unknown if **x** or **z** in operand[s]	**0, 1, x**
A === B	A equal to B, including **x** and **z**	**0, 1**
A !== = B	A no equal to B, including **x** and **z**	**0, 1**

```
module equality;

reg [7:0] foo;
reg [7:0] bar;

initial begin
    foo = 45; bar = 9;
    $display("Logical result for foo(%0d) === bar(%0d) : %0d", foo, bar, foo === bar);
    foo = 'b101x; bar = 'b1011;
    $display("Logical result for foo(%0b) === bar(%0b) : %0d", foo, bar, foo === bar);
    foo = 'b101x; bar = 'b101x;
    $display("Logical result for foo(%0b) === bar(%0b) : %0d", foo, bar, foo === bar);
    foo = 'b101z; bar = 'b1z00;
    $display("Logical result for foo(%0b) !== bar(%0b) : %0d", foo, bar, foo !== bar);
    foo = 39; bar = 39;
    $display("Logical result for foo(%0d) == bar(%0d) : %0d", foo, bar, foo == bar);
    foo = 14; bar = 14;
    $display("Logical result for foo(%0d) != bar(%0d) : %0d", foo, bar, foo != bar);

end
endmodule
```

Operator(4): Logical Operator

Operator	Description
A && B	True if a and b are true
A || B	True if a or b is true
!A	Converts non-zero value to zero, and vice versa

```
module logical;

reg [7:0] foo;
reg [7:0] bar;

initial begin
    foo = 45; bar = 9;
    $display("Logical result for foo(%0d) && bar(%0d) : %0d", foo, bar, foo == bar);
    foo = 0; bar = 4;
    $display("Logical result for foo(%0d) && bar(%0d) : %0d", foo, bar, foo == bar);
    foo = 'dx; bar = 3;
    $display("Logical result for foo(%0d) && bar(%0d) : %0d", foo, bar, foo == bar);
    foo = 'b101z; bar = 5;
    $display("Logical result for foo(%0d) && bar(%0d) : %0d", foo, bar, foo == bar);
    foo = 45; bar = 9;
    $display("Logical result for foo(%0d) || bar(%0d) : %0d", foo, bar, foo || bar);
    foo = 0; bar = 4;
    $display("Logical result for foo(%0d) || bar(%0d) : %0d", foo, bar, foo || bar);
    foo = 'dx; bar = 3;
    $display("Logical result for foo(%0d) || bar(%0d) : %0d", foo, bar, foo || bar);
    foo = 'b101z; bar = 5;
    $display("Logical result for foo(%0d) || bar(%0d) : %0d", foo, bar, foo || bar);
    foo = 4;
    $display("Logical result for !foo(%0d) : %0d", foo, !foo);
    foo = 0;
    $display("Logical result for !foo(%0d) : %0d", foo, !foo);
end
```

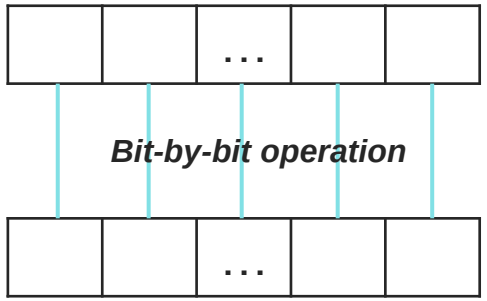
Operator(5): Bitwise Operator

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

Bitwise AND

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

Bitwise OR



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

Bitwise XOR

^~ ~^	0	1	X
0	0	1	X
1	1	0	X
X	X	X	X

Bitwise XNOR

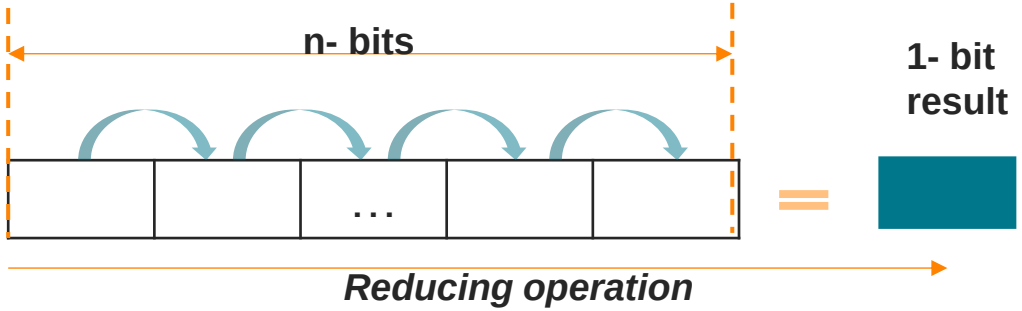
~	
0	1
1	0
X	X

Bitwise NEGATION

Operator(5): Bitwise Operator

```
1   module bitwise;
2
3   initial begin
4     // Bit Wise Negation
5     $display (" ~4'b0001           = %b", (~4'b0001));
6     $display (" ~4'bx001           = %b", (~4'bx001));
7     $display (" ~4'bz001           = %b", (~4'bz001));
8     // Bit Wise AND
9     $display (" 4'b0001 &  4'b1001 = %b", (4'b0001 &  4'b1001));
10    $display (" 4'b1001 &  4'bx001 = %b", (4'b1001 &  4'bx001));
11    $display (" 4'b1001 &  4'bz001 = %b", (4'b1001 &  4'bz001));
12    // Bit Wise OR
13    $display (" 4'b0001 |  4'b1001 = %b", (4'b0001 |  4'b1001));
14    $display (" 4'b0001 |  4'bx001 = %b", (4'b0001 |  4'bx001));
15    $display (" 4'b0001 |  4'bz001 = %b", (4'b0001 |  4'bz001));
16    // Bit Wise XOR
17    $display (" 4'b0001 ^  4'b1001 = %b", (4'b0001 ^  4'b1001));
18    $display (" 4'b0001 ^  4'bx001 = %b", (4'b0001 ^  4'bx001));
19    $display (" 4'b0001 ^  4'bz001 = %b", (4'b0001 ^  4'bz001));
20    // Bit Wise XNOR
21    $display (" 4'b0001 ~^ 4'b1001 = %b", (4'b0001 ~^ 4'b1001));
22    $display (" 4'b0001 ~^ 4'bx001 = %b", (4'b0001 ~^ 4'bx001));
23    $display (" 4'b0001 ~^ 4'bz001 = %b", (4'b0001 ~^ 4'bz001));
24    #10 $finish;
25  end
26  endmodule
```

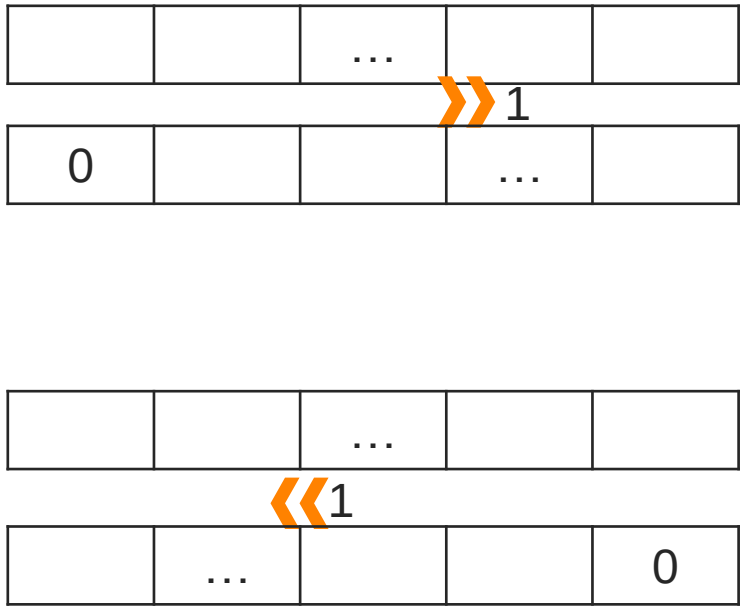
Operator(6): Reduction Operator



Operator	Description
&	Reduction AND
~&	Reduction NAND
|	Reduction OR
~|	Reduction NOR
^	Reduction XOR
~^, ^~	Reduction XNOR

```
1   module reduction;
2
3   initial begin
4     // Bit Wise AND reduction
5     $display (" &  4'b1001 = %b", (&  4'b1001));
6     $display (" &  4'bx111 = %b", (&  4'bx111));
7     $display (" &  4'bz111 = %b", (&  4'bz111));
8     // Bit Wise NAND reduction
9     $display (" ~& 4'b1001 = %b", (~& 4'b1001));
10    $display (" ~& 4'bx001 = %b", (~& 4'bx001));
11    $display (" ~& 4'bz001 = %b", (~& 4'bz001));
12    // Bit Wise OR reduction
13    $display (" |  4'b1001 = %b", (|  4'b1001));
14    $display (" |  4'bx000 = %b", (|  4'bx000));
15    $display (" |  4'bz000 = %b", (|  4'bz000));
16    // Bit Wise OR reduction
17    $display (" ~| 4'b1001 = %b", (~| 4'b1001));
18    $display (" ~| 4'bx001 = %b", (~| 4'bx001));
19    $display (" ~| 4'bz001 = %b", (~| 4'bz001));
20    // Bit Wise XOR reduction
21    $display (" ^  4'b1001 = %b", (^  4'b1001));
22    $display (" ^  4'bx001 = %b", (^  4'bx001));
23    $display (" ^  4'bz001 = %b", (^  4'bz001));
24    // Bit Wise XNOR
25    $display (" ~^ 4'b1001 = %b", (~^ 4'b1001));
26    $display (" ~^ 4'bx001 = %b", (~^ 4'bx001));
27    $display (" ~^ 4'bz001 = %b", (~^ 4'bz001));
28    #10 $finish;
29  end
30
31  endmodule
```

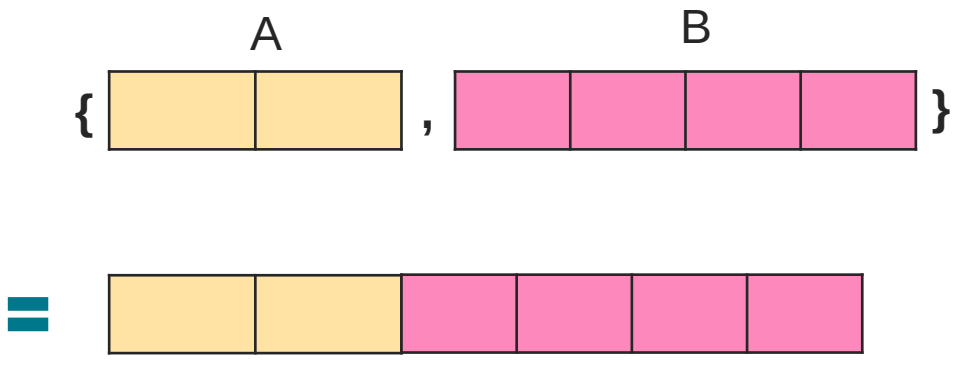
Operator(7): Shift Operator




```
module shift;

initial begin
  // Left Shift
  $display (" 4'b1001 << 1 = %b", (4'b1001 << 1));
  $display (" 4'b10x1 << 1 = %b", (4'b10x1 << 1));
  $display (" 4'b10z1 << 1 = %b", (4'b10z1 << 1));
  // Right Shift
  $display (" 4'b1001 >> 1 = %b", (4'b1001 >> 1));
  $display (" 4'b10x1 >> 1 = %b", (4'b10x1 >> 1));
  $display (" 4'b10z1 >> 1 = %b", (4'b10z1 >> 1));
  #10 $finish;
end

endmodule
```

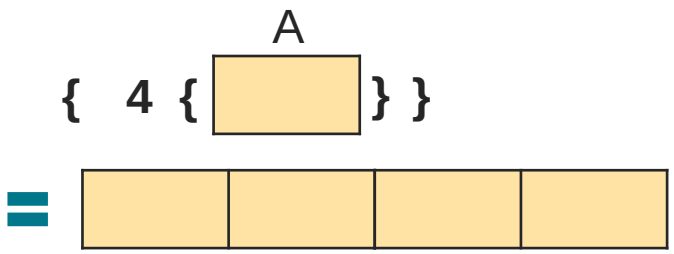
Shift Operator를 사용하면 어떤 arithmetic operation이 가능할까?

Operator(8~9): Concatenation / Replication Operator



Concatenation

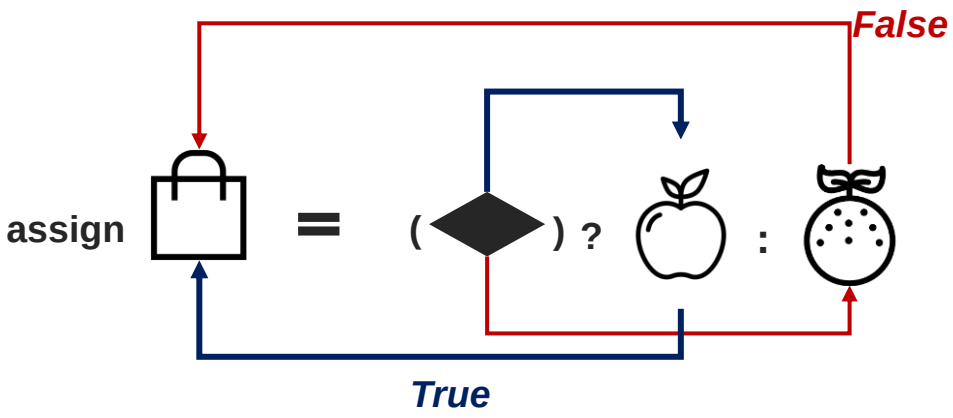
```
module concatenation;

initial begin
  // concatenation
  $display (" {4'b1001,4'b10x1}  = %b", {4'b1001,4'b10x1});
  #10 $finish;
end

endmodule
```



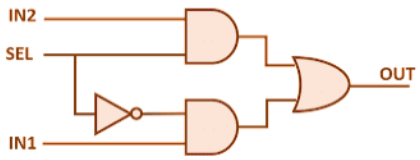
Replication

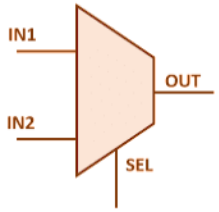
```
module replication;

initial begin
  // replication
  $display (" {4{4'b1001}}      = %b", {4{4'b1001}});
  // replication and concatenation
  $display (" {4{4'b1001,1'bz}} = %b", {4{4'b1001,1'bz}});
  #10 $finish;
end

endmodule
```

Operator(10): Tenary Operator



assign out = (sel ==1'b1) ? in1 : in0;






```
1   module tri_state_buf;
2
3
4   wire out;
5   reg enable,data;
6   // Tri state buffer
7   assign out = (enable) ? data : 1'bz;
8
9   initial begin
10    $display ("time\t enable data out");
11    $monitor ("%g\t %b      %b    %b",$time,enable,data,out);
12    enable = 0;
13    data = 0;
14    #1 data = 1;
15    #1 data = 0;
16    #1 enable = 1;
17    #1 data = 1;
18    #1 data = 0;
19    #1 enable = 0;
20    #10 $finish;
21  end
22
23  endmodule
```

Tri-state buffer modeling

LAB(1)

- Practice diverse operators in ~/Digital_Design/src/session_6/operator
- Guess what the result will be before simulation!

LAB(2)

- Clock Gating Modeling

LAB(3)

- Implement MUX modeling using "*tenary operator*"