Digital Design with the Verilog HDL

Chapter 4 RTL Model

Binh Tran-Thanh

May 26, 2023

RTL Verilog

- Higher-level of description than structural
  - Don't always need to specify each individual gate
  - Can take advantage of **operators**
- More hardware-explicit than behavioral
  - Doesn't look as much like software
  - Frequently easier to understand what's happening
- Very easy to synthesize
  - Supported by even primitive synthesizers

Continuous Assignment

- Implies structural hardware

  ```
  assign <LHS> = <RHS expression>;
  ```

- Example

  ```
  wire out, a, b;
  assign out = a & b;
  ```

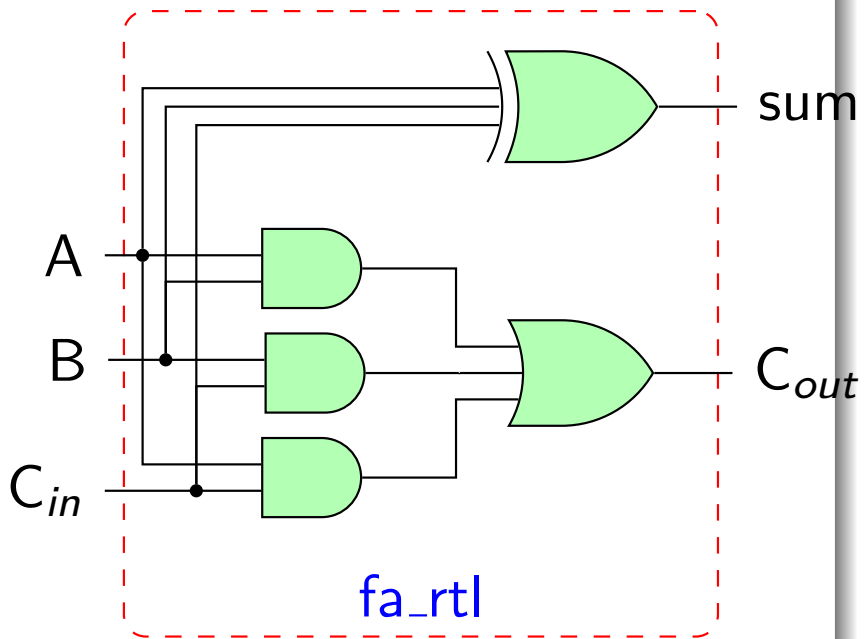

- If RHS result changes, LHS is updated with new value
  - Constantly operating ("continuous"!)
  - It's **hardware!**
- Used to model combinational logic and latches

Full Adder: RTL/Dataflow

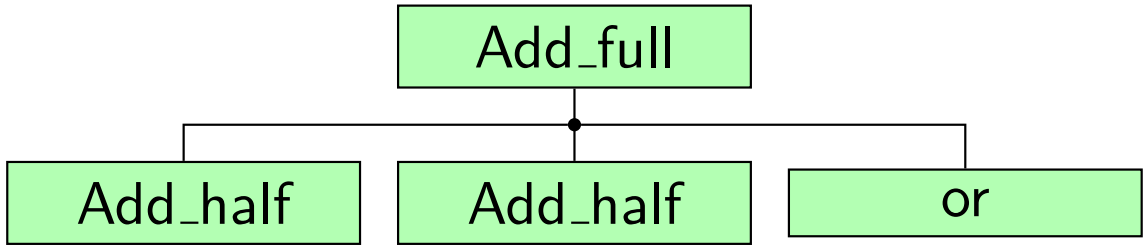
Example from Lecture 02

```
module fa_rtl (A, B, CI, S, CO);
  input A, B, CI;
  output S, CO;

  // use continuous assignments
  assign S = A ^ B ^ CI;
  assign CO = (A & B) | (A & CI)
     | (B & CI);
endmodule
```



RTL And Structural Combined




```
module Add_half(sum,
    cout, a, b);
  output sum, cout;
  input a, b;

  assign sum = a ^ b;
  assign cout = a & b;
endmodule
```

```
module Add_full(c_out, sum, a, b,
    c_in);
 output sum, c_out;
 input a, b, c_in;
 wire psum, c1, c2;

 Add_half AH1(partsum, c1, a, b);
 Add_half AH2(sum, c2, psum, c_in);
 assign c_out = c1 | c2;
endmodule
```

Continuous Assignment LHS

Can assign values to

- Scalar nets
- Vector nets
- Single bits of vector nets
- Part-selects of vector nets
- Concatenation of any of the above

Examples

```
assign out[7:4] = a[3:0] | b[7:4];
assign val[3] = c & d;
assign {a, b} = stimulus[15:0];
```

Continuous Assignment RHS

Use operators

- Arithmetic, Logical, Relational, Equality, Bitwise, Reduction, Shift, Concatenation, Replication, Conditional
- Same set as used in Behavioral Verilog

Can also be a pass-through!

```
assign a = stimulus[16:9];
assign b = stimulus[8:1];
assign cin = stimulus[0];
```

- Note: "aliasing" is only in one direction
  - Cannot give 'a' a new value elsewhere to set stimulus[16:9]!

Implicit Continuous Assignments

- Can create an implicit continuous assign
- Goes in the wire declaration

```
wire[3:0] sum = a + b;
```

- Can be a useful shortcut to make code succinct, but doesn't allow fancy LHS combos

```
assign {cout, sum} = a + b + cin;
```

- Personal choice
  - You are welcome to use it when appropriate

Implicit Wire Declaration

- Can create an implicit wire
- When wire is used but not declared, it is implied

```
module majority(output out, input a, b, c);
  assign part1 = a & b;
  assign part2 = a & c;
  assign part3 = b & c;
  assign out = part1 | part2 | part3;
endmodule
```

- Lazy! Don't do it!
  - Use explicit declarations
  - To design well, need to be "in tune" with design!

Verilog Operators

Operator	Name	Group	Example	Precedence
[ ]	Select		b[2:1] b[2]	
{}	Concatenation	Concat.	{a,b}	
{{}}	Replication	Repl.	{3{a}}	
!	Negation (inverse)	Logical	!a	1 (unary)
~	Negation (not)	Bit-wise	~ a	
&	Reduction AND	Reduction	& a	
|	Reduction OR		| a	
~ &	Reduction NAND		~ & a	
~ |	Reduction NOR		~ | a	
∧	Reduction XOR		∧ a	
~ ∧ or ∧ ~	Reduction XNOR		~ ∧ a	
+	Positive (unary)	Arithmetic	+ a	
−	Negative (unary)		− a	

Verilog Operators

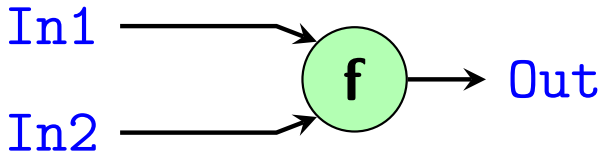
Operator	Name	Group	Example	Precedence
*	Multiplication	Arithmetic	a* b	2 (binary)
/	Division		a/ b	
%	Modulus		a% b	
+	Addition		a+ b	3 (binary)
−	Subtraction		a− b	
<<	Shift left	Shift	a<< 4	4 (binary)
>>	Shift right		a>> 4	
>	Greater	Relational	a> b	5 (binary)
>=	Greater or equal		a>= b	
<	Less		a< b	
>=	Less or equal		a>= b	

Verilog Operators

Operator	Name	Group	Example	Precedence
==	Equal (logic)	Equal	a== b	6 (binary)
! =	Not equal (logic)		a! = b	
===	Equal (case)		a=== b	
! ==	Not equal (case)		a! == b	
&	bit-wise AND	Bit-wise	a&b	7 (binary)
∧	bit-wise XOR		a∧ b	
∼ ∧ or ∧ ∼	bit-wise XNOR		a∼ ∧ b	
|	bit-wise OR		a| b	
&&	logical AND	Logic	a&&b	8 (binary)
||	logical OR		a|| b	
?:	Conditional	Conditional	a? b: c	9 (binary)

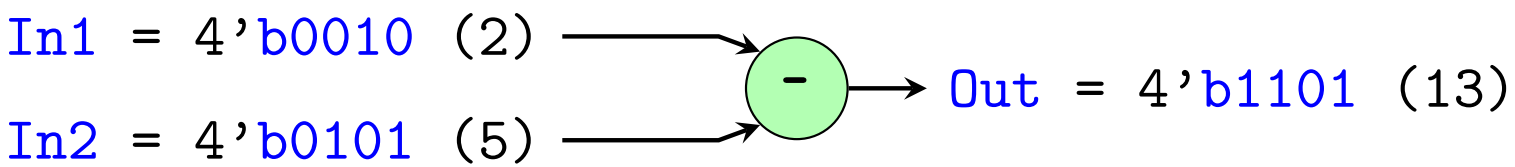
Arithmetic Operators ($+, -, *, /, \%$)

If any bit in the operands is **x** or **z**, the result will be **x**

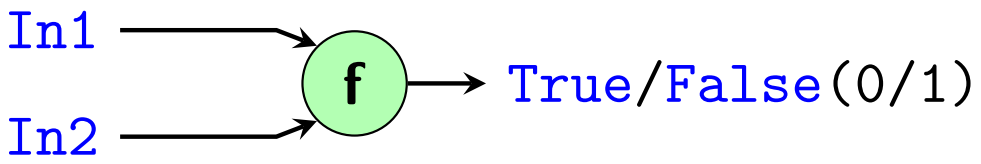


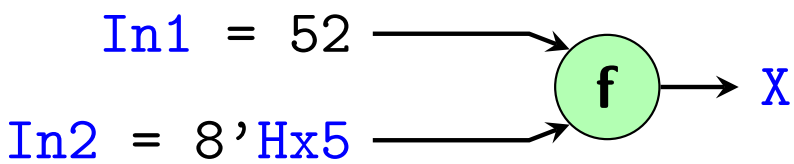
The result's size

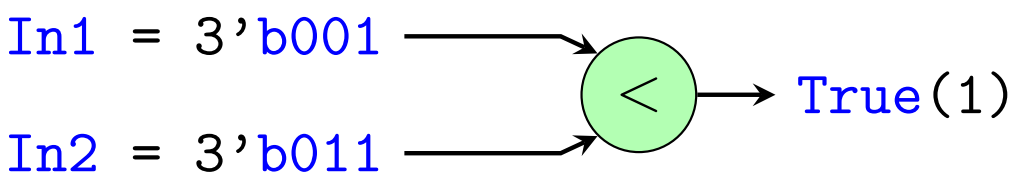
- **Mul: sum of both operands**
- **Others: size of the bigger operand**

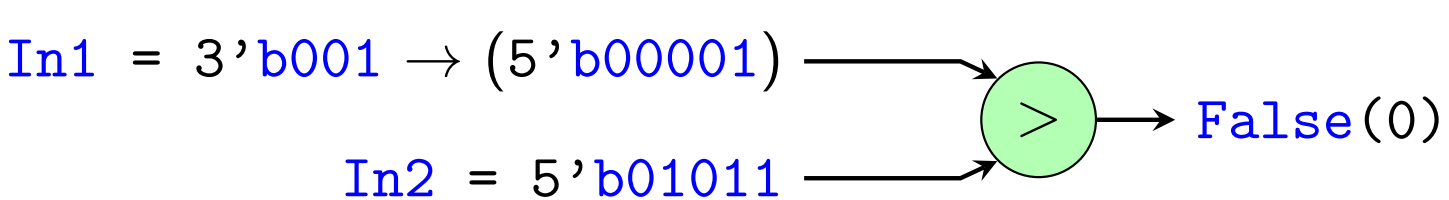


Relational Operators ($<, >, <=, >=$)









Equality Operators (==, ===, ! =, ! ===)

- Logical comparison (== and !=)
  - The **x** and **z** values are processed as in Relative operators
  - The result may be x
- Case comparison (=== and !==)
  - Bitwise compare
  - x === x, z === z, x !== z
  - The result is always 0 or 1
- If two operands are not the same size, **0(s) will be inserted into MSB** bits of the smaller operand

```
Data = 4'b11x0;
Addr = 4'b11x0;
Data == Addr  //x
Data === Addr //1
```

Logical Operators (||, &&, !)

- **Vector with at least one bit 1 is 1**
- **If any bit in the operands is x or z, the result will be x**

```
ABus = 4'b0110;
BBus = 4'b0100;
ABus || BBus// 1
ABus && BBus// 1
!ABus        // Similar to !BBus
             // 0
```

Bit-wise Operators (&, |, ∼, ∧, ∧ ∼)

& (and)	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

| (or)	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

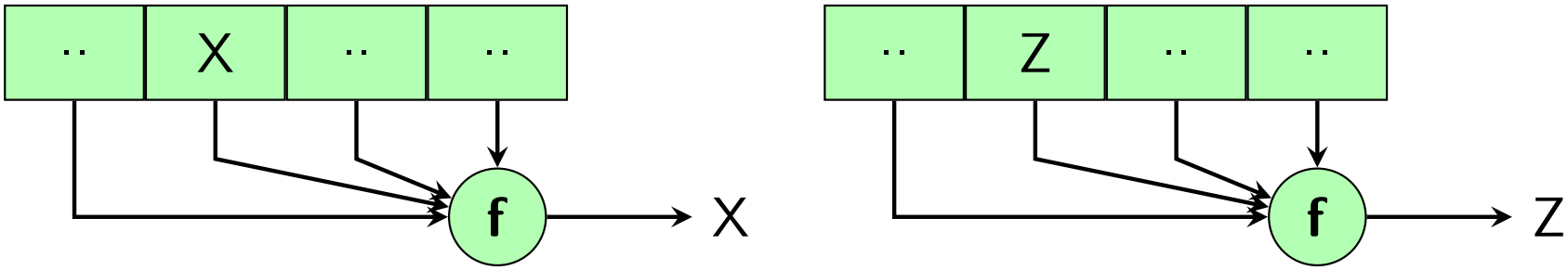
∧ (xor)	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

∧ ∼ (xnor)	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

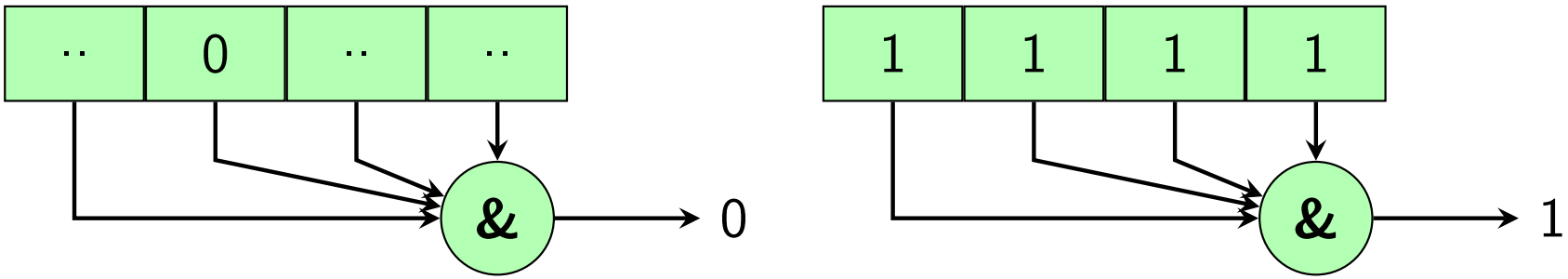
∼ (not)	0	1	x	z
	1	0	x	x

Reduction Operators

Dont care (X) and HighZ (Z)

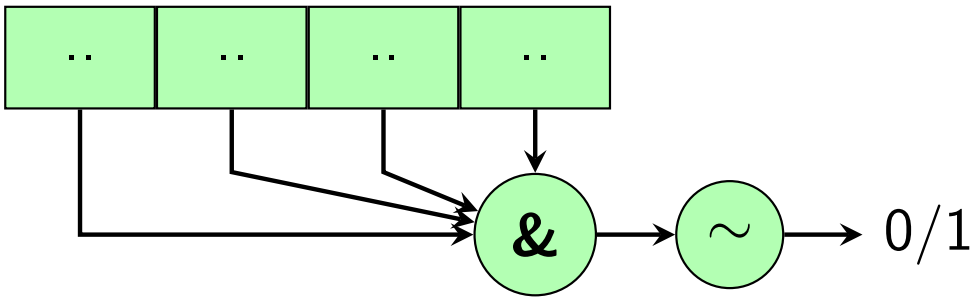


&(and reduction): $\&b_n b_{n-1} \ldots b_1 b_0$

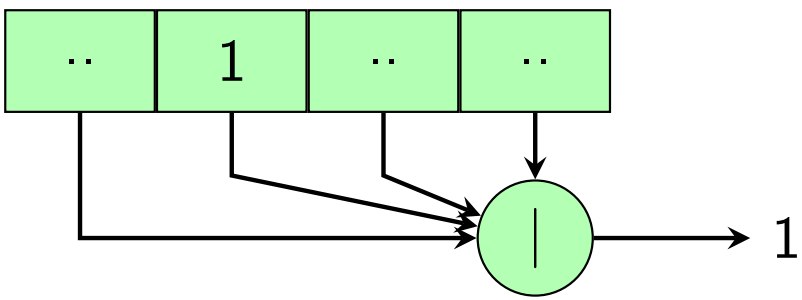


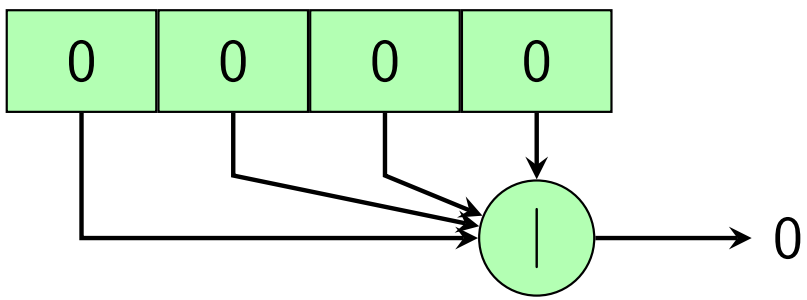
Reduction Operators

$\sim$ & (nand reduction): $\sim \& b_n b_{n-1} \ldots b_1 b_0$



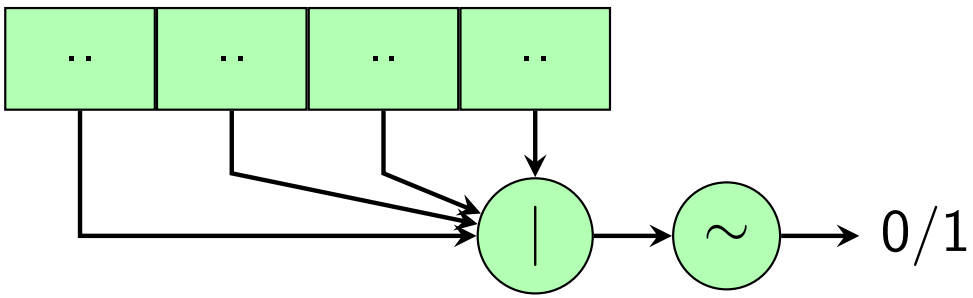
| (or reduction): $|b_n b_{n-1} \ldots b_1 b_0$





Reduction Operators

$\sim |$ (or reduction): $\sim |b_n b_{n-1} \ldots b_1 b_0$



$\wedge$ (xor reduction): $\wedge b_n b_{n-1} \ldots b_1 b_0$

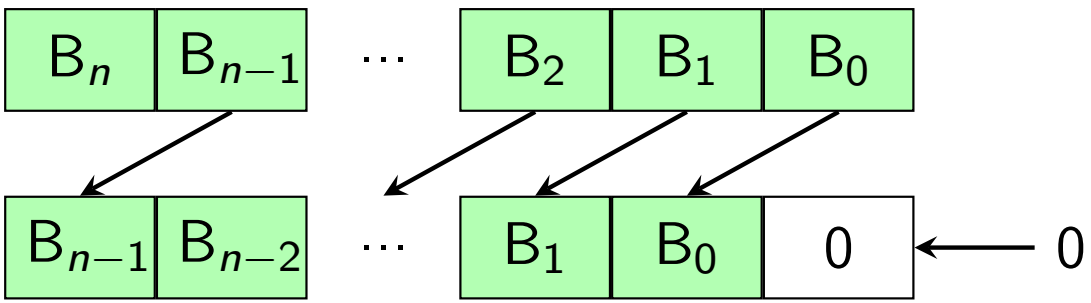
If count ($b_i = 1$) mod 2 == 0 then return 0;
Otherwise return 1

$\sim \wedge / \wedge \sim$(xnor reduction): $\sim \wedge b_n b_{n-1} \ldots b_1 b_0$

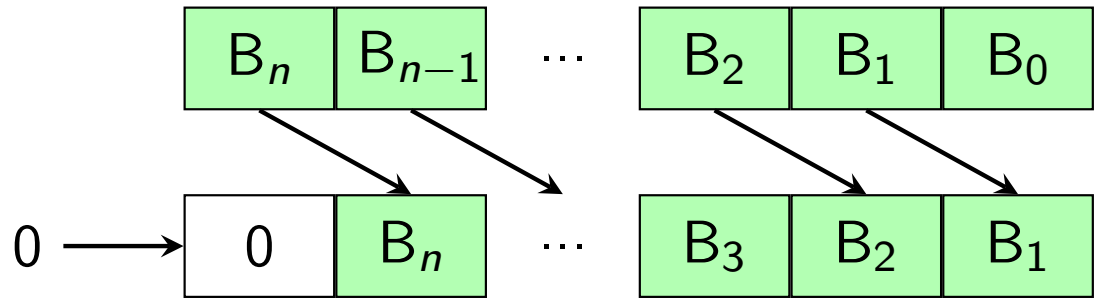
Shift Operators (<<, >>)

Shift the left operand the number of times represented by the right operand

Shift left



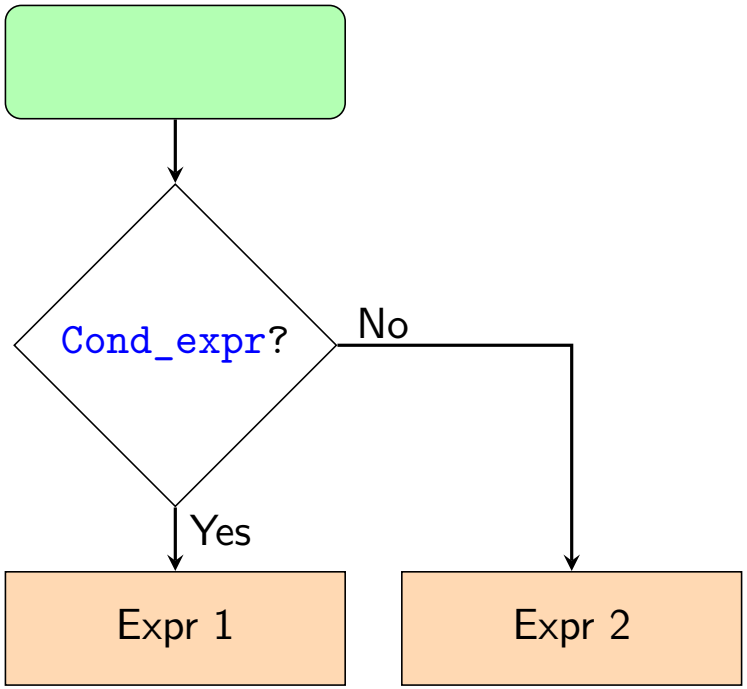
Shift right




```
reg [0:7] Qreg;
  Qreg = 4'b0111;
    // 8'b0000_0111
  Qreg >> 2;
    // 8'b0000_0001
  Qreg = 4'd1 << 5;
    // 8'b0010_0000
```

Conditional Operator `Cond_expr ? Expr1: Expr2`

- If Cond_expr includes any x bit or z bit, the result will be a "bitwise operation" of Expr1 and Expr2 as following:
  - 0 ♣ 0 ⇒ 0
  - 1 ♣ 1 ⇒ 1
  - otherwise x
- Infinite nested conditional operator




```
wire[15:0] bus_a = drive_a ? data : 16'bz;
  /* drive_a = 1 data is copied to bus_a
   * drive_a = 0 bus_a is high-Z
   * drive_a = x bus_a is x
   */

```

Concatenation and Replication Operators

- Concatenation {expr1, expr2, ... ,exprN}
  - Does not work with **un-sized constants**

```
wire [7:0] Dbus;
wire [11:0] Abus;
assign Dbus[7:4] = {Dbus[0],Dbus[1],Dbus[2],Dbus[3]};
assign Dbus = {Dbus[3:0], Dbus[7:4]};
//{Dbus, 5} // not allowed
```

- Replication {rep_number{expr1, expr2, ... , exprN}}

```
Abus = {3{4'b1011}}; // 12'b1011_1011_1011
{3{1'b1}}            // 111
{3{Ack}}             // {Ack, Ack, Ack}
```

Expression Bit Lengths

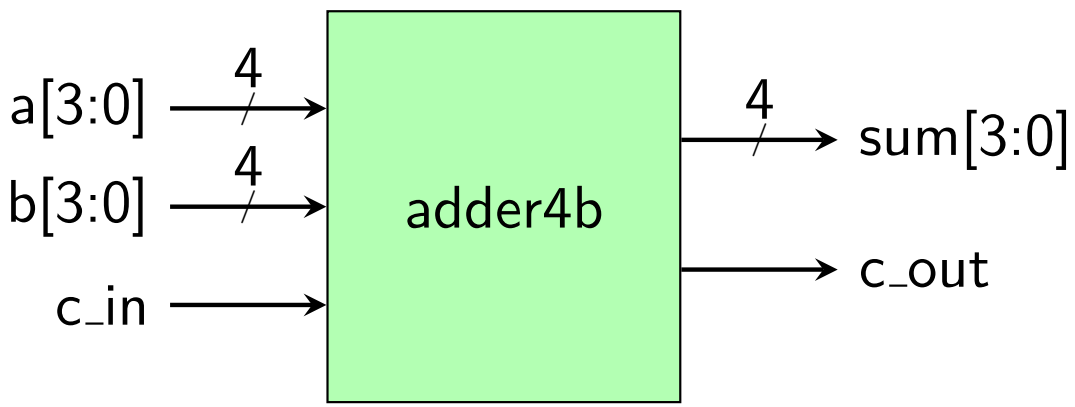
Expression	**Bit length**	**Comments**
Unsized constant number	Same as integer (32 bit)	
Sized constant number	As given	
a <op> b, where <op> is: $+, -, *, /, \%, \mid, \wedge, \sim \wedge$	Max(L(a),L(b))	L(a): length (a)
a <op> b, where <op> is: ===, !==, ==, !=, &&, $\mid\mid$, >, >=, <, <=	1 bit	Operands are sized to Max(L(i),L(j))
a <op> b, where <op> is: $\&, \sim \&, \mid, \sim \mid, \wedge, \sim \wedge$, !	1 bit	
a <op> b, where <op> is: >>, <<	L(i)	

Example: adder4b

```
module adder4b (sum, c_out, a, b, c_in);
  input[3:0] a, b;
  input c_in;
  output[3:0] sum;
  output c_out;

  assign{c_out, sum} = a + b + c_in;

endmodule
```



Example: Unsigned MAC Unit

Design a multiply-accumulate (MAC) unit that computes

`Z[7:0] = A[3:0]*B[3:0] + C[7:0]`

It sets overflow to one, if the result cannot be represented using 8 bits.

```
module mac(output [7:0] Z, output overflow,
           input[3:0] A, B, input[7:0] C);
```

Solution: Unsigned MAC Unit

```
module mac(output [7:0] Z, output overflow,
           input[3:0] A, B, input[7:0] C);
  wire [8:0] P;
  assign P = A*B + C;
  assign Z = P[7:0];
  assign overflow = P[8];
endmodule
```

Alternative method:

```
module mac(output[7:0] Z, output overflow,
           input[3:0] A, B, input[7:0] C);
  assign {overflow, Z} = A*B + C;
endmodule
```

Example: Multiplexer

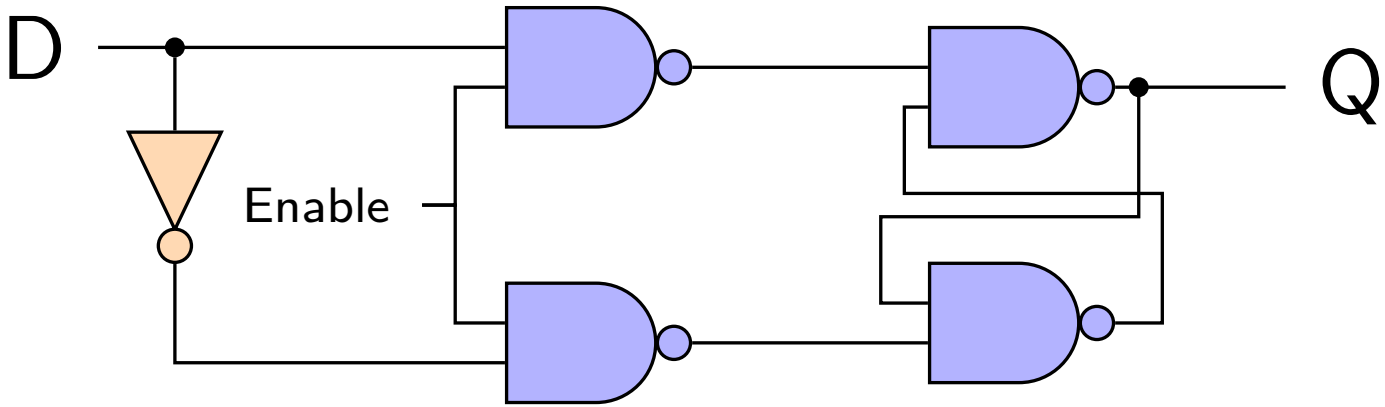
Use the conditional operator and a single continuous assignment statement

```
module mux_8_to_1(output out,
  input in0, in1, in2, in3, in4, in5, in6, in7,
  input[2:0] sel);




endmodule
```

Latches



Latches

- Continuous assignments with feedback

```
module latch(output[7:0] q_out,
             input[7:0] data_in, enable);
  assign q_out = enable ? data_in: q_out;
endmodule

module latch_reset(output q_out,
                   input data_in, enable, reset);
  assign q_out = reset ? 0 : (enable ? data_in: q_out);
endmodule
```

- How would we change these for 8-bit latches?
- How would we make the enable active low?

Example: Rock-Paper-Scissors (optional homework)

- `module rps(win, player, p0guess, p1guess);`
- Assumptions:
  - Input: p0guess, p1guess = 0 for rock, 1 for paper, 2 for scissors
  - Output: player is 0 if p0 wins, 1 if p1 wins, and don't care if there is a tie
  - Output: win is 0 if there is a tie and 1 if a player wins
- Reminders
  - Paper beats rock, scissors beats paper, rock beats scissors
  - Same values tie
- Two possible approaches
  - Figure out the Boolean equations for win and player and implement these using continuous assignments
    - Use bitwise operators
  - Examine what the various items equal and do logical operations on these
    - Use equality and logical operators

Draw synthesized hardware of following verilogHDL

```
wire [3:0] a, b, c;
wire [7:0] d;
assign c = d[7:4] + b;
assign d = a * b;
```

```
wire [3:0] a, b, c;
assign c = !a && b ? a + b: a - b;
```

Take away message

- Using continuous assign: `assign LHS = RHS`.
- Operation and its order.
- Length of inputs, output.