Verilog Tutorial

TA: 陳炫均

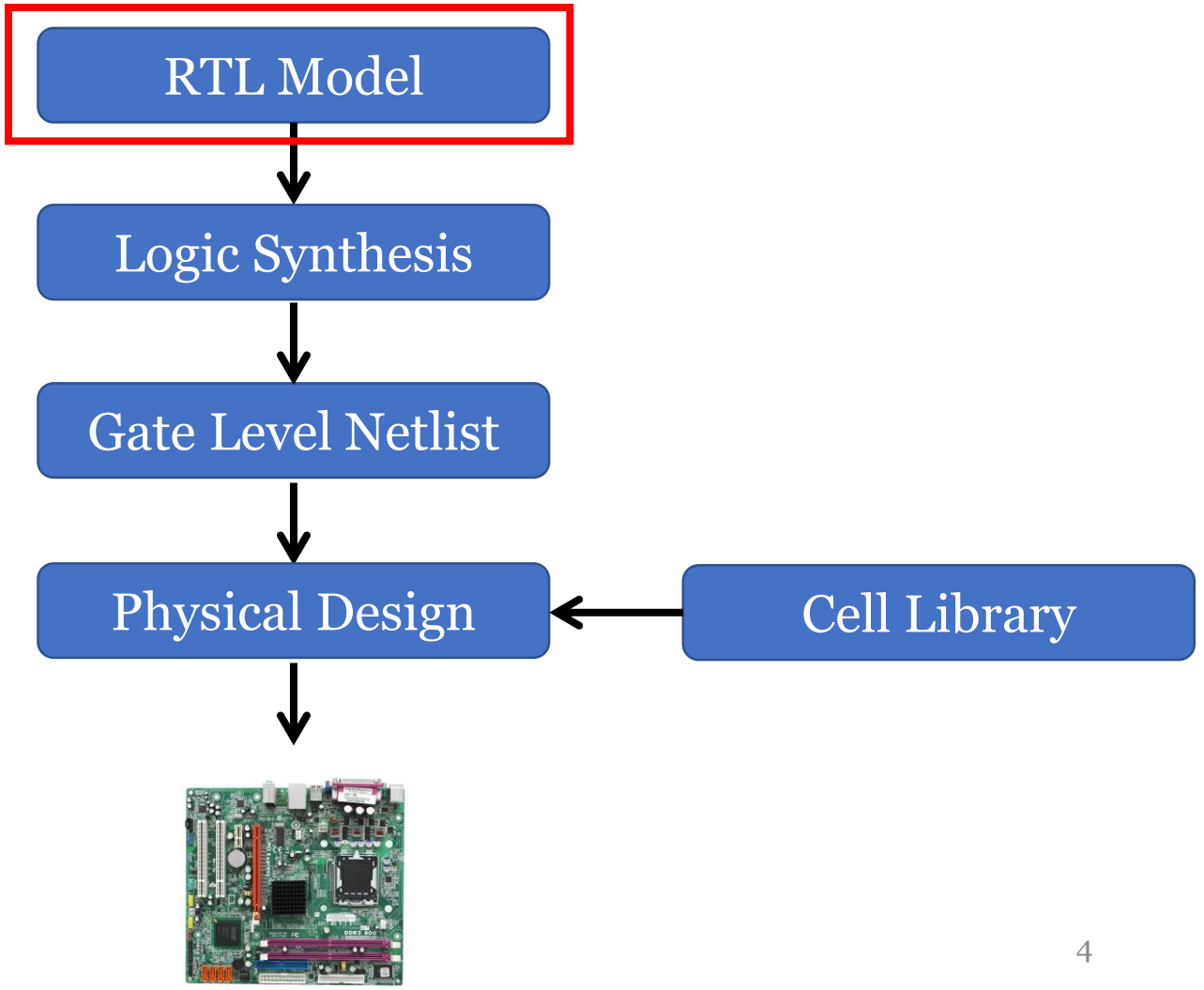
Outline

- Introduction
- Major Data Types
- Operations
- Behavior Modeling
- Structure Modeling

Outline

- Introduction
- Major Data Types
- Operations
- Behavior Modeling
- Structure Modeling

IC Design Flow



Full Adder as an Example


```
module module_name (port_name);
```

- Port Declaration
- Data Type Declaration
- Task & Function Declaration
- Module Functionality or Structure
- Timing Specification

```
endmodule
```

Full Adder Example 1

module_name

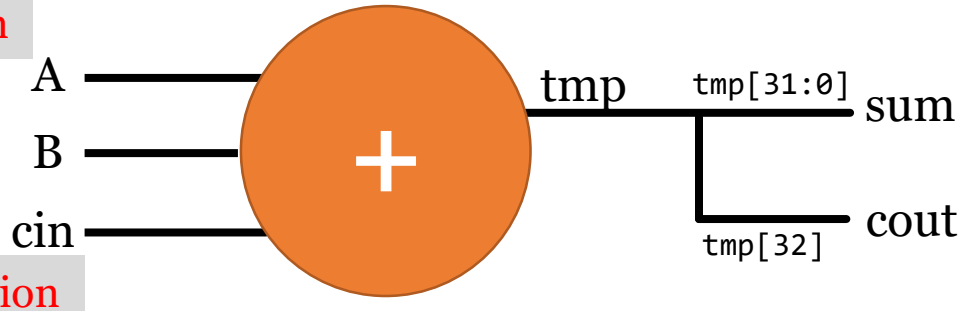
port_name

```
 1 module Full_Adder(sum, cout, a, b, ci);
 2
 3 //Interface
 4 input       [31:0] a, b;
 5 input              ci;
 6 output      [31:0] sum;
 7 output             cout;
 8
 9 wire        [32:0] tmp;
10
11 // Calculation (with continuous assignment)
12 assign tmp = a + b + ci;
13 assign sum = tmp[31:0];
14 assign cout = tmp[32];
15
16 endmodule
```

port declaration

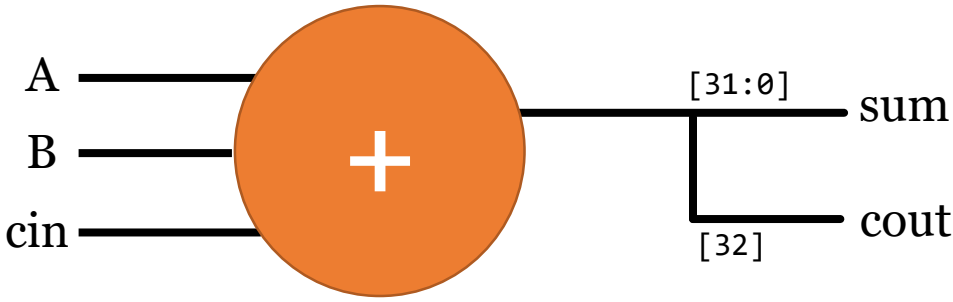
data type declaration

module functionality



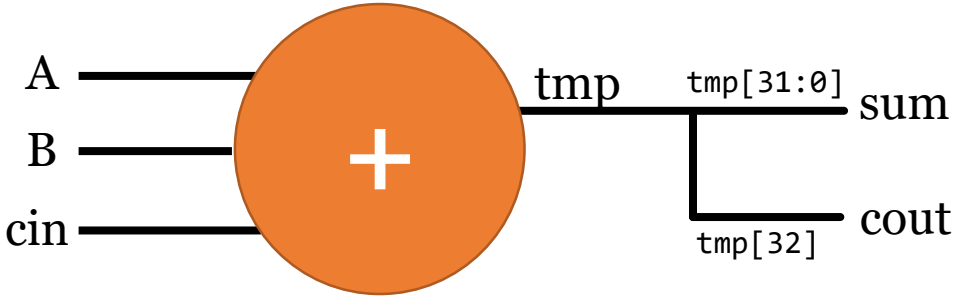
Full Adder Example 2


```
 1 module Full_Adder(sum, cout, a, b, ci);
 2
 3 //Interface
 4 input       [31:0] a, b;
 5 input              ci;
 6 output      [31:0] sum;
 7 output             cout;
 8
 9 // Calculation (with continuous assignment)
10 assign {cout, sum} = a + b + ci;
11
12 endmodule
```



Full Adder Example 3

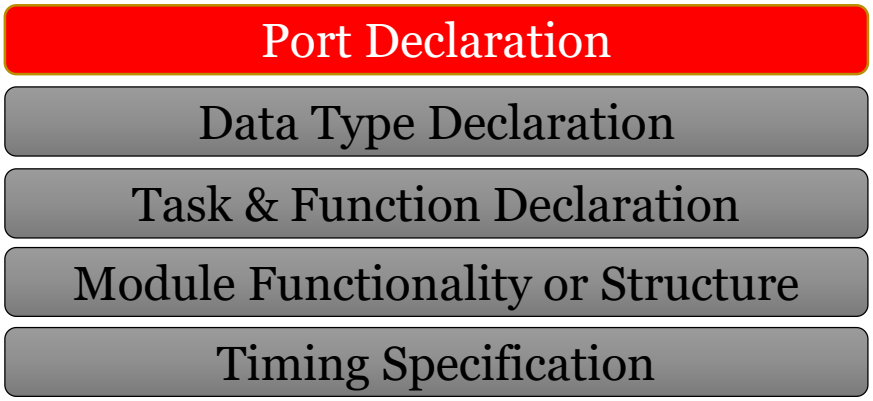

```
 1 module Full_Adder(sum, cout, a, b, ci);
 2
 3 //Interface
 4 input       [31:0] a, b;
 5 input              ci;
 6 output      [31:0] sum;
 7 output             cout;
 8
 9 reg         [32:0] tmp;
10 // Calculation (with Always Procedural Block)
11 assign sum = tmp[31:0];
12 assign cout = tmp[32];
13
14 always@ (a or b or ci) begin
15     tmp = a + b + ci;
16 end
17
18 endmodule
```



Port declaration

- declare input and output ports
  - `input a;`
  - `output b;`

```
module module_name (port_name);
```




```
endmodule
```

Data Type Declaration

- Declare the data types used in the module
  - `wire [32:0] tmp;`
  - `reg [32:0] tmp;`

```
module module_name (port_name);
```

Port Declaration

Data Type Declaration

Task & Function Declaration

Module Functionality or Structure

Timing Specification

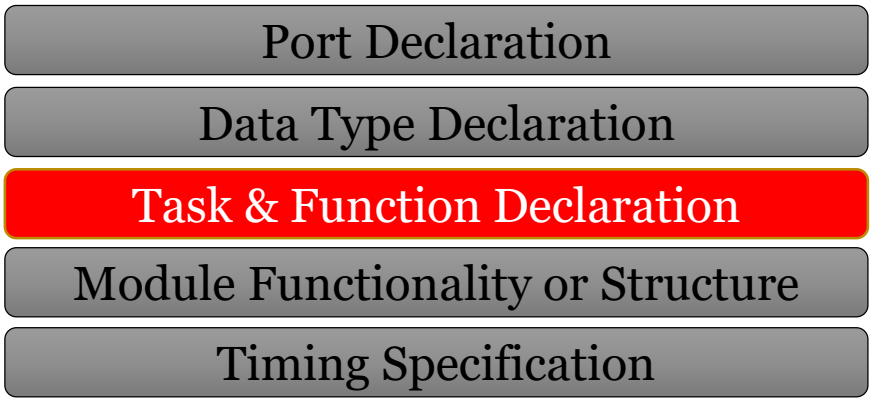
```
endmodule
```

Task & Function Declaration

- Declare the modules used

```
alu ALU_instance(
    .src_a(src_a),
    .src_b(src_b),
    .c(c), .data_out(data_out));
```

```
module module_name (port_name);
```

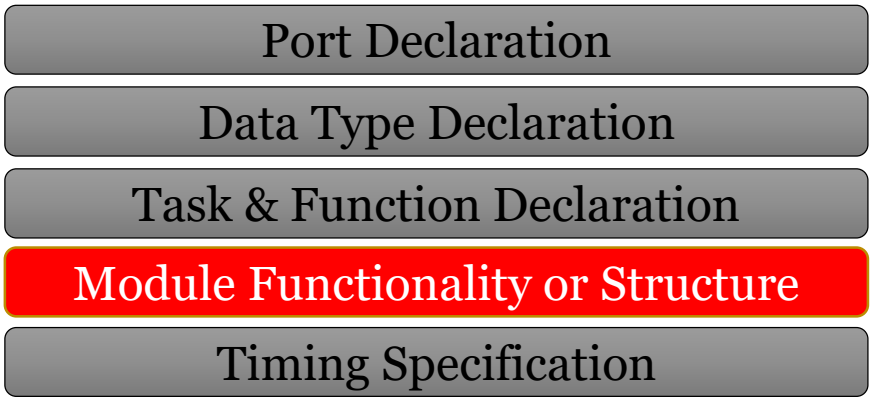



```
endmodule
```

Module Functionality

- Describe the module behavior
  - `assign {cout, sum} = a + b + ci;`
  - `tmp = a + b + ci;`

```
module module_name (port_name);
```




```
endmodule
```

Timing Specification

- Specify the time elapsed
  - #50;

```
module module_name (port_name);
```

Port Declaration

Data Type Declaration

Task & Function Declaration

Module Functionality or Structure

Timing Specification

```
endmodule
```

```
10 initial begin
11
12   $dumpfile("simpleALU.vcd");
13   $dumpvars;
14
15   src_a = 8'h55;  // Time = 0
16   src_b = 8'h1a;
17   c = 3'b000;
18   #50;         // Time = 50
19   c = 3'b001;
20   #50;         // Time = 100
21   c = 3'b010;
22   #50;         // Time = 150
23   c = 3'b011;
24   #50;         // Time = 200
25   c = 3'b100;
26   #50;         // Time = 250
27   c = 3'b101;
28   #50;         // Time = 300
29   c = 3'b110;
30   #50;         // Time = 350
31   c = 3'b111;
32 end
```

Compiler Directives

- `define
  - `define CYTLE_TIME 50
  - Defining a name and gives a constant value to it
  - Like #define CYCLE_TIME 50 in C
- `include
  - `include adder.v
  - Including the entire contents of other Verilog source file
  - Like #include “adder.h” in C
- `timescale
  - `timescale 100ns/1ns
  - Setting the reference time unit and time precision of the simulation

System Tasks

- `$monitor`
  - `$monitor ($time, “%d %d %d”, a, b, cin);`
  - Displays the values of the argument list whenever any of the arguments change except $time
- `$display`
  - `$display (“%d %d %d”, a, b, cin);`
  - Prints out the current values of the signals in the argument list
  - Like “`printf`” in C
- `$finish`
  - `$finish;`
  - Terminate the simulation

Outline

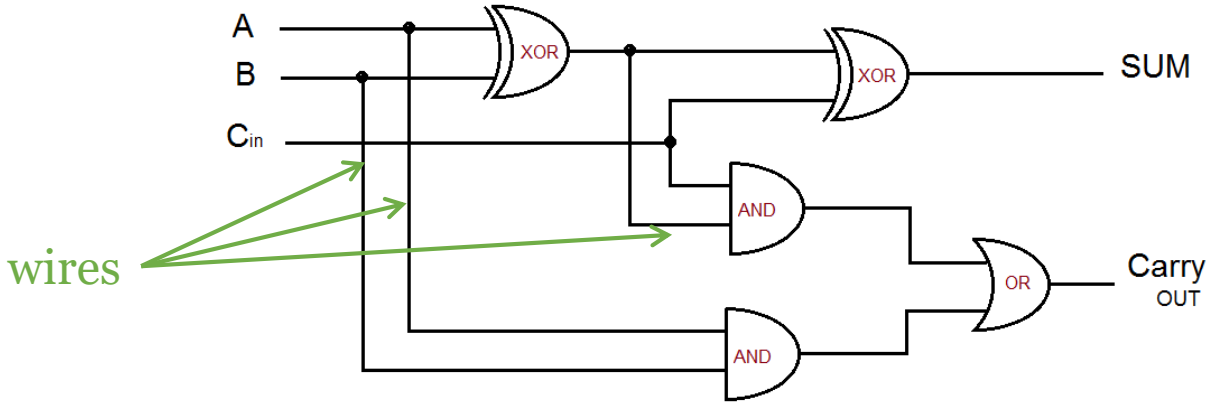
- Introduction
- Major Data Types
- Operations
- Behavior Modeling
- Structure Modeling

Major Data Types

- Wire (Net)
- Register
- Parameter
- Numbers

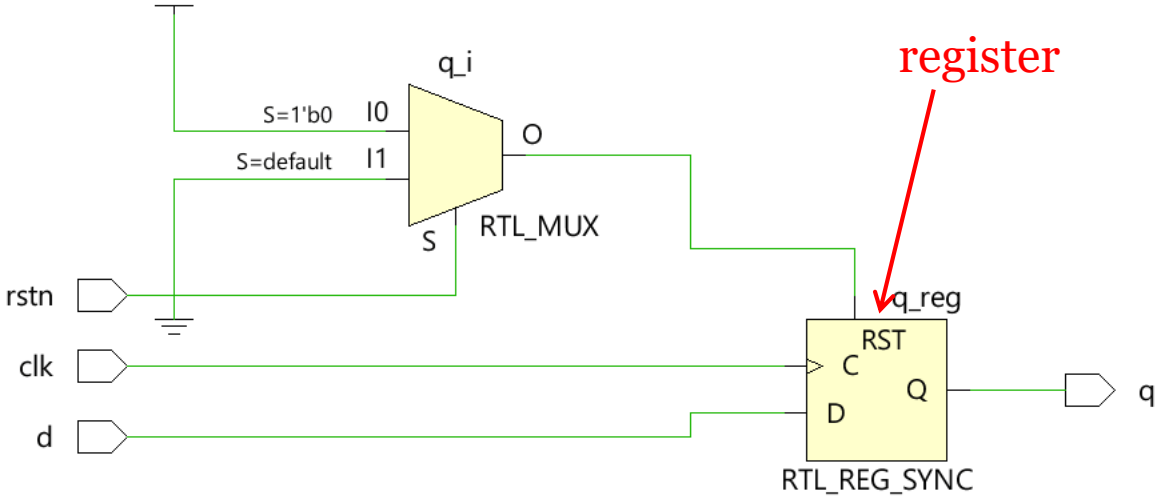
Wire (Net)

- **Wires** represent physical connections between structural entities
- A **wire** must be driven by a driver, such as a gate or a continuous assignment.
- Verilog automatically propagates new values onto a wire when drivers change value.



Register

- Registers represent abstract storage elements
- A register holds its value until a new value is assigned to it
- Registers are used extensively in behavior modeling



Assignment

- Continuous assignment
  - e.g. `assign c = a + b;`
  - the wire c will continuously be updated to value of a+b no matter what
- Procedural assignment
  - e.g. `c = 1;`
  - only executed when under some condition

Parameter

- **Parameters** are not variables, they are constants.
- Typically parameters are used to specify delays and width of variables.

```
module var_mux(out, i0, i1, sel);

parameter width=2, delay=1;
output [width-1:0] out;
input  [width-1:0] i0, i1;
input              sel;

assign #delay out = sel? i1 : i0;

endmodule
```

Wire and Registers

- Wire
  - Doesn't store value, just a connection
  - Cannot appear in "`always`" block assignment
  - input, output are default "wire"
  - Default value is Z (high-impedance)
- Register
  - A storage element
  - Assignment in "`always`" block
  - Event-driven modeling
  - Default value is X (unknown)

Number Representation

- Format: <size>'<base_format><number>
- <size> - # of bits
  - Default is unsized and machine-dependent (but at least 32 bits)
- <base_format> - arithmetic base of *number*
  - <d> <D> - decimal, which is default base if *base_format* is not given
  - <h> <H> - hexadecimal
  - <o> <O> - octal
  - <b> <B> - binary
- <number> - the value in base of *base_format*
  - _ can be inserted in between for readability

Number Examples

Number	Binary value
6’b010_111	010111
8’b0110	00000110
4’bx01	xx01
16’H3AB	0000001110101011
24	000…00011000
5’O36	11110
16’Hx	xxxxxxxxxxxxxxxx
8’hz	zzzzzzzz
8’d16	00010000

Outline

- Introduction
- Major Data Types
- Operations
- Behavior Modeling
- Structure Modeling

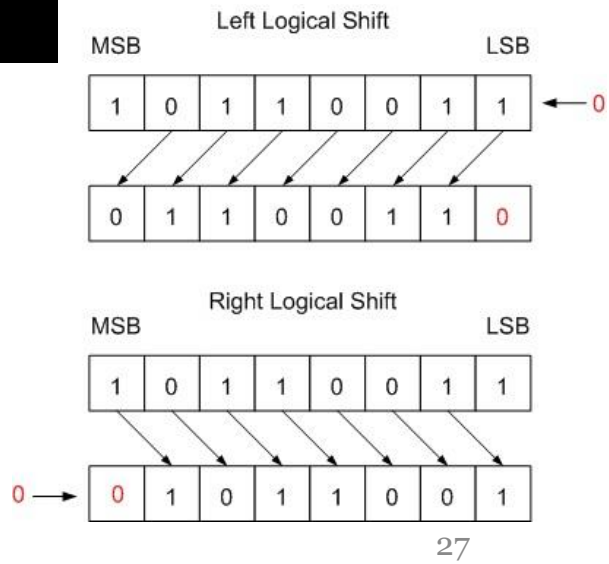
Operators

Operator	Name
[ ]	bit-select or part-select
( )	parenthesis
!, ~	logical and bit-wise NOT
&, |, ~&, ~|, ^, ~^, ^~	reduction AND, OR, NAND, NOR, XOR, XNOR; If X=3'B 101 and Y=3'B 110, then X&Y=3'B 100, X^Y=3'B011;
+, -	unary (sign) plus, minus; +17, -7
{ }	concatenation; {3'B 101, 3'B 110} = 6'B 101110;
{{ }}	replication; {3{3'B 110}} = 9'B 110110110
*, /, %	multiply, divide, modulus; */ and % not be supported for synthesis*
+, -	binary add, subtract.
<<, >>	shift left, shift right; X<<2 is multiply by 4
<, <=, >, >=	comparisons. Reg and wire variables are taken as positive numbers.
= =, !=	logical equality, logical inequality
= = =, != =	case equality, case inequality; not synthesizable
&	bit-wise AND; AND together all the bits in a word
^, ~^, ^~	bit-wise XOR, bit-wise XNOR
|	bit-wise OR; AND together all the bits in a word
&&	logical AND. Treat all variables as False (zero) or True (nonzero). logical OR. (7||0) is (T||F) = 1, (2||-3) is (T||T) =1, (3&&0) is (T&&F) = 0.
||	
? :	conditional. x=(cond)? T : F;

Table 5.1: Verilog Operators Precedence

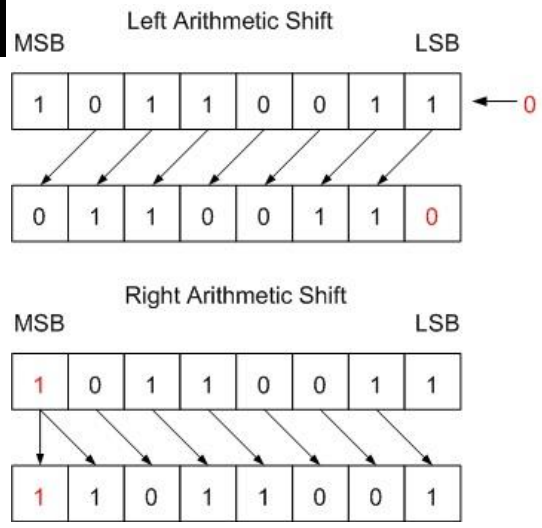
Shift operator (logically)

```
 1 modele shift_register(reg_out, reg_in);
 2
 3 output [5:0]  reg_out;
 4 input  [5:0]  reg_in;
 5
 6 parameter shift = 3;
 7
 8 assign reg_out = reg_in << shift;
 9
10 endmodule
```



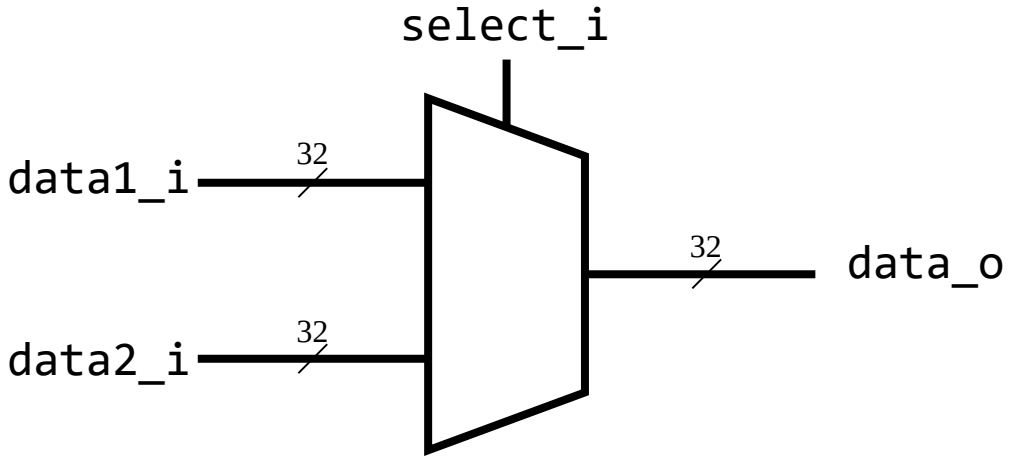
Shift operator (arithmetically)

```
 1 modele shift_register(reg_out, reg_in);
 2
 3 output [5:0]  reg_out;
 4 input  [5:0]  reg_in;
 5
 6 parameter shift = 3;
 7
 8 assign reg_out = reg_in <<< shift;
 9
10 endmodule
```



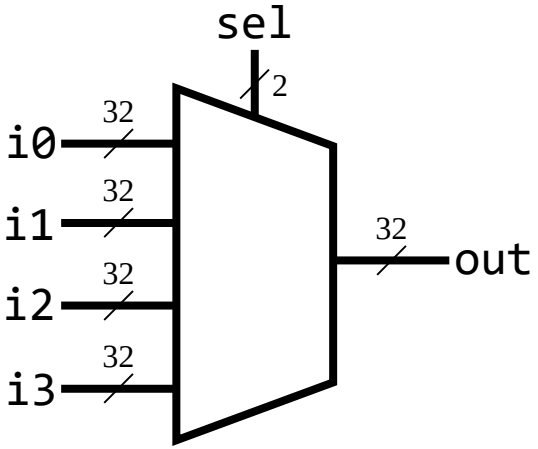
Conditional operator

```
1 module MUX32(data1_i, data2_i, select_i, data_o);
2 input  [31:0] data1_i, data2_i;
3 input         select_i;
4 output [31:0] data_o;
5
6 assign data_o = select_i? data2_i : data1_i;
7
8 endmodule
```



Conditional operator (cont.)

```
 1 module MUX4(out, i0, i1, i2, i3, sel);
 2
 3 output  [3:0] out;
 4 input   [3:0] i0, i1, i2, i3;
 5 input   [1:0] sel;
 6
 7 assign out = (sel == 2'b00)? i0:
 8              (sel == 2'b01)? i1:
 9              (sel == 2'b10)? i2:
10              (sel == 2'b11)? i3:
11              4'bx;
12 endmodule
```



Concatenation & Replication Operator

- Concatenation operator in LHS
  - `assign {co, sum} = a + b + c;`

- Bit replication to produce 01010101
  - `assign byte = {4{2'b01}};`

Outline

- Introduction
- Major Data Types
- Operations
- Behavior Modeling
- Structure Modeling

Behavior Modeling

- In behavior modeling, you must describe your circuits'
  - Action
    - How your circuits behave?
  - Timing control
    - At what time doing what thing
    - Under what condition do what thing
- Verilog supports followings to model circuits' behavior
  - Procedural block
  - Procedural assignment
  - Timing control
  - Control statement

Procedural Blocks

- In Verilog, procedural blocks are the basic of behavior modeling
  - You can describe one behavior in one procedural block
- Procedural blocks are of two types
  - "`initial`" procedural block
    - execute only once
  - "`always`" procedural block
    - execute in a loop

Procedural Blocks (cont.)

- All procedural blocks are activated at simulation time 0
- With enabling condition, the block will not be executed until the enabling condition is evaluated to TRUE
- Without enabling condition, the block will be executed immediately

initial Block

```
10 initial begin
11
12   $dumpfile("Full_Adder.vcd");
13   $dumpvars;
14
15   // Time = 0
16   a  = 1'b0;
17   b  = 1'b0;
18   ci = 1'b0;
19
20 end
```

```
35 initial #1000
36     $finish;
```

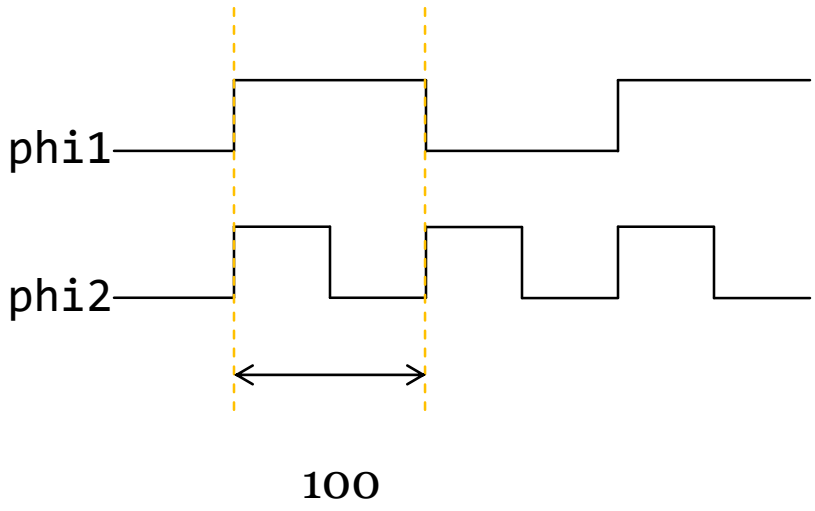
always Block

```
21 always@(posedge clk_i or posedge rst_i) begin
22     if(rst_i) begin
23         pc_o <= 32'b0;
24     end
25     else begin
26         if(start_i)
27             pc_o <= pc_i;
28         else
29             pc_o <= pc_o;
30     end
31 end
```

```
14 always@ (a or b or ci) begin
15     tmp = a + b + ci;
16 end
```

always Blocks (cont.)

```
 1 module clock_gen(phi1, phi2);
 2
 3 output phi1, phi2;
 4 reg phi1, phi2;
 5
 6 initial begin
 7     phi1 = 0; phi2 = 0;
 8 end
 9
10 always #100 phi1 = ~phi1;
11
12 always @(posedge phi1)
13 begin
14     phi2 = 1;
15     #50 phi2 = 0;
16     #50 phi2 = 1;
17     #50 phi2 = 0;
18 end
19
20 endmodule
```



Procedural Assignment

- Procedural assignments drive values or expressions onto registers

```
 1 module Full_Adder(sum, cout, a, b, ci);
 2
 3 //Interface
 4 input       [31:0] a, b;
 5 input              ci;
 6 output      [31:0] sum;
 7 output             cout;
 8
 9 reg         [32:0] tmp;
10 // Calculation (with Always Procedural Block)
11 assign sum = tmp[31:0];
12 assign cout = tmp[32];
13
14 always@ (a or b or ci) begin
15     tmp = a + b + ci;
16 end
17
18 endmodule
```

Procedural assignments

Common mistakes

```
 1 module f_adder(sum, co, a, b, ci);
 2
 3 output  sum, co;
 4 input a, b, ci;
 5 reg sum_reg;
 6
 7 sum_reg = a ^ b ^ ci;
 8
 9 always @(a or b or ci)
10     assign co = (a&b) | (b&ci) | (ci&a);
11
12 endmodule
```

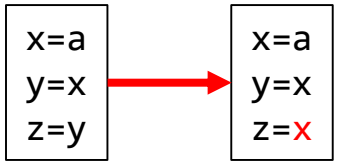
A procedural assignment **must** be inside procedural blocks

A continuous assignment statement **connot** be inside procedural blocks

Blocking and Non-blocking Procedural Assignment

- Blocking procedural assignment

```
always @(posedge clock) begin
    x = a;
    y = x;
    z = y;
end
```



- Non-blocking procedural assignment

```
always @(posedge clock) begin
    x <= a;
    y <= x;
    z <= y;
end
```

```
x=a
y=x_old
z=y_old
```

Conditional Statements

- `if` and `if-else` statement

```
if (expression)
     statement
```

```
if (expression)
    statement
else
    statement
```

```
if (expression)
    statement
else if (expression)
    statement
else
    statement
```

```
if (rega >= regb)
    result = 1;
else
    result = 0;
```

Ex1

```
if (index > 0)
    if (rega > regb)
        result = rega;
    else
        result = 0;
else
    $display("[WARNING] index is equal or smaller than 0!")
```

Ex2

Conditional Statements

- `case` statement

```
10 `define AND 3'b000
11 `define OR  3'b001
12 `define ADD 3'b010
13 `define SUB 3'b110
14 `define SLT 3'b111
15 `define MUL 3'b011
16
17 always @ (data1_i or data2_i or ALUCtrl_i)
18 begin
19     case (ALUCtrl_i)
20         `AND: data_reg = data1_i & data2_i;
21         `OR:  data_reg = data1_i | data2_i;
22         `ADD: data_reg = data1_i + data2_i;
23         `SUB: data_reg = data1_i - data2_i;
24         `SLT: data_reg = (data1_i < data2_i)? 1 : 0;
25         `MUL: data_reg = data1_i * data2_i;
26     endcase
27 end
```

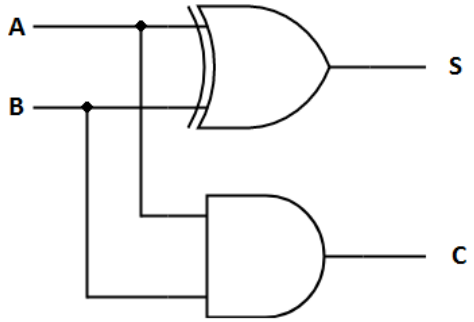
Outline

- Introduction
- Major Data Types
- Operations
- Behavior Modeling
- Structure Modeling

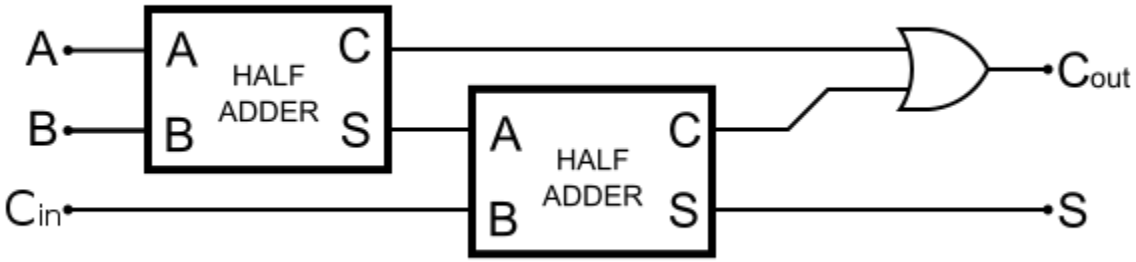
Structure Modeling

- Connecting components with each other to create a more complex component

Half adder

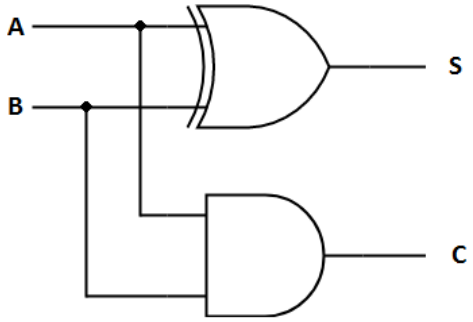


Full adder



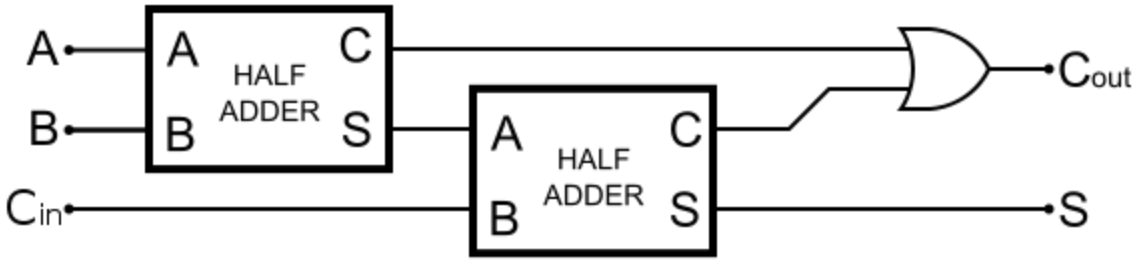
Structure Modeling

Half adder




```
 1 module HA(a, b, sum, co);
 2
 3 input a, b;
 4 output sum, co;
 5
 6 assign sum = a ^ b;
 7 assign co = a & b;
 8
 9 endmodule
```

Full adder




```
 1 module FA(A, B, cin, sum, cout);
 2
 3 input A, B, cin;
 4 output sum, cout;
 5
 6 wire sum0, co0, co1;
 7
 8 HA ha0(A, B, sum0, co0);
 9 HA ha1(sum0, cin, sum, co1);
10
11 assign cout = co0 ^ co1;
12
13 endmodule
```