



IC OVERVIEW

RTL DESIGN AND VERIFICATION

COURSE INTRODUCTION

Khóa Học Thiết Kế Vi Mạch Cơ Bản - Trung Tâm Đào Tạo Thiết Kế Vi Mạch ICTC



KHÓA THIẾT KẾ VI MẠCH CƠ BẢN

Khóa học đào tạo cho các bạn các kiến thức kỹ năng cơ bản về vi mạch, chú trọng thực hành thiết kế và kiểm tra mạch để tạo nền tảng vững chắc cho sự nghiệp vi mạch sau này!

LỘ TRÌNH TỰ HỌC VI MẠCH 📖

KHÓA HỌC THIẾT KẾ VI MẠCH 🎓

- ✓ Giảng viên là các kỹ sư vi mạch hơn 5 - 10 năm trong nghề
- ✓ Giáo trình hiện đại đúc kết từ các công ty vi mạch toàn cầu
- ✓ Tập trung đào tạo thực hành về kỹ năng cần thiết khi làm kỹ sư vi mạch
- ✓ Phần mềm học trực tiếp trên Server đang được các công ty sử dụng
- ✓ Kinh nghiệm, kiến thức về tìm việc làm, phỏng vấn ngành vi mạch

COURSE INTRODUCTION



SUMMARY



HOMEWORK



QUESTION



SELF-LEARNING

Session 6: Verilog Fundamental – Part 1

Data Types and Operators



1. Verilog introduction
2. Data type
3. Assignment
4. Operator



1. Verilog introduction

2. Data type

3. Assignment

4. Operator

VERILOG FUNDAMENTAL

What is Verilog ?

- ❑ Verilog is standardized as IEEE 1364 standard and used to describe digital electronic circuits.
- ❑ Verilog HDL is used mainly in design and verification at the RTL level.
- ❑ Verilog enables designers to describe the structure and behavior of digital circuits using a textual representation, making it easier to design, simulate, and verify complex digital systems.
- ❑ Verilog code can be synthesized into hardware by synthesis tools.
- ❑ Verilog is used by simulation tools to verify the functionality and performance of digital designs before physical implementation.

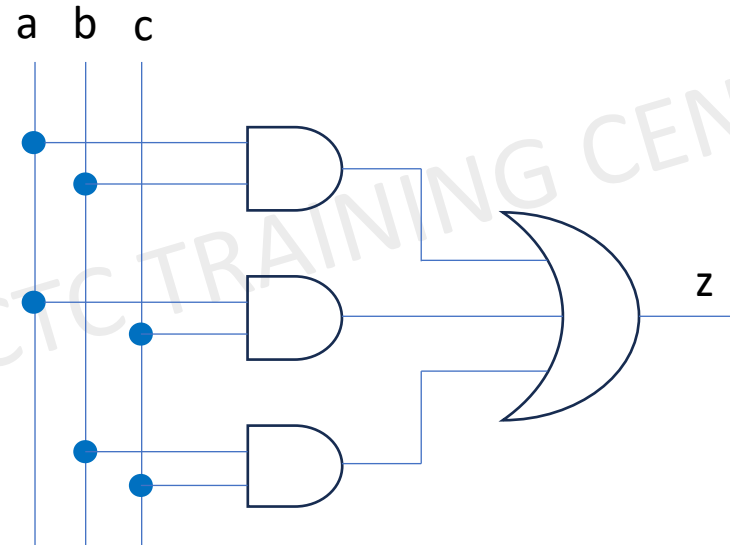


VERILOG FUNDAMENTAL

Synthesizable and Non-synthesizable

- ❑ **Synthesizable:** The code that is written in a way that can be directly translated into hardware circuit by synthesis tool. It represents the actual logic gates, flip-flops, and other hardware elements that will be synthesized into a digital circuit.

```
module combo (  
    input wire a,  
    input wire b,  
    input wire c,  
    output wire z  
);  
  
    assign z = (a & b) | (a & c) | (b & c);  
endmodule
```



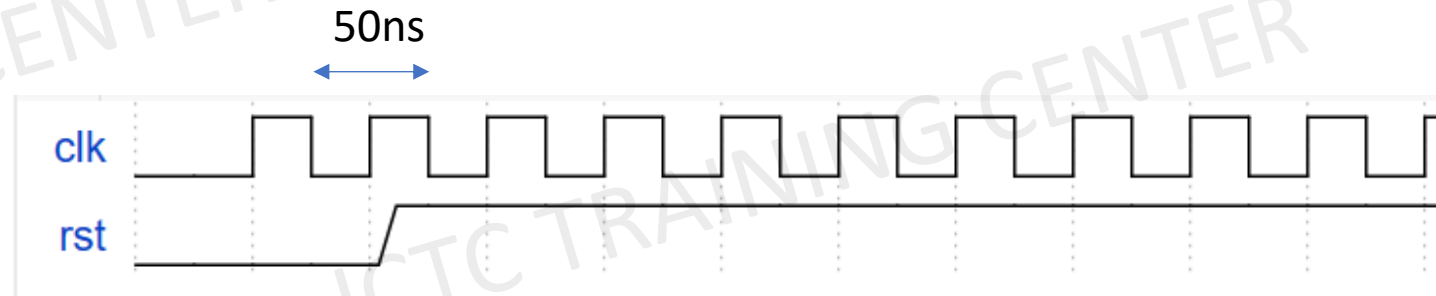
VERILOG FUNDAMENTAL

Synthesizable and Non-synthesizable

- ❑ **Non-synthesizable:** The code that cannot be synthesized into a digital circuit by synthesis tools. The non-synthesizable RTL is often used in simulation environment, testbench or writing behavioral model.



```
module tb ();  
  
    reg clk;  
  
    initial begin  
        clk = 0;  
        #25ns;  
        forever begin  
            clk = #25ns ~clk;  
        end  
    end  
  
    initial begin  
        rst = 0;  
        #100ns rst = 1;  
    end  
  
endmodule
```



VERILOG FUNDAMENTAL

Numbers

❑ Number format: *width* '*base* *value*

width: expressed in decimal integer

'base: binary(b), octal(o), decimal(d) or hexadecimal(h). Default is decimal

value: value of the number.

■ Decimal number

- 50

- 8'd50

■ Octal number

- 8'o050

■ Hexadecimal number

- 10'h1FF

- 5'h3 or 5'h03

■ Binary number

- 8'b10010111 or 8'b1001_0111 or 8'b10_01_01_11

■ Real number (not synthesizable)

- 3.14

- .28e2



VERILOG FUNDAMENTAL

Numbers



- ❑ **Sign bit:** can use “s” to indicate a sign number
 - With “s” in base number: negative value if MSB is 1, positive value if MSB is 0
 - Without “s” in base number: always positive even if MSB is 1

Unsigned number: 4'b1110 → decimal 14

3	2	1	0
1	1	1	0

Signed number: 4'sb1110 → decimal -2

3	2	1	0
1	1	1	0



Signed bit

VERILOG FUNDAMENTAL

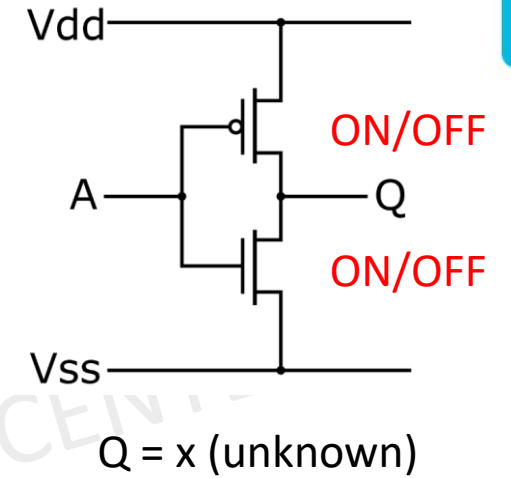
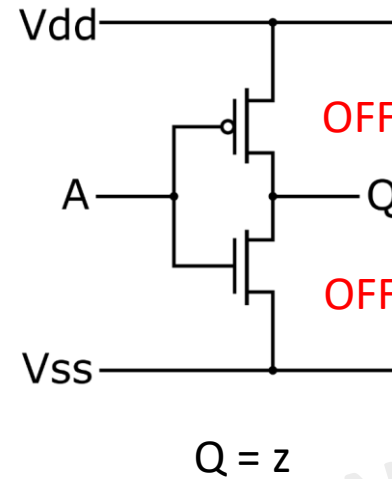
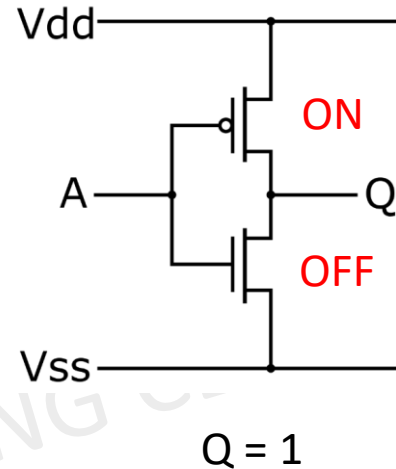
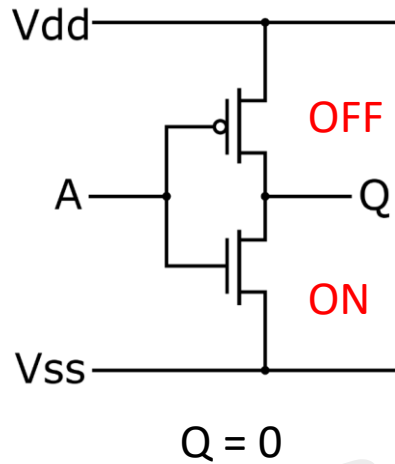
Numbers



- **4 basic values in verilog:**
 - 0 – logic zero, or false condition
 - 1 – logic one, or true condition
 - x – unknown/undefined logic value.
 - z – high-impedance(hi-Z)/floating state.
- Decimal numbers are represented as plain integers, and they cannot directly contain 'x' (unknown) or 'z' (high-impedance) values.
- Octal, binary and hex numbers can have x and z value
- **Example:**
 - 4'bxx01: 4-bit binary number where the two most significant bits are unknown.
 - 12'o3z21: 12-bit octal number having 3 bits 6,7,8 are undriven (floating).
 - 16'hxffz: 16-bit hexadecimal number where the 4 MSBs are unknown and 4 LSBs are undriven (floating).

VERILOG FUNDAMENTAL

Numbers



- “x” value can only be used in simulation, to check the unknown state of the circuit.
- “x” is not appeared in actual chip because the circuits always have a state.
- The actual chip only has 3 state: 0,1,z.





1. Verilog introduction

2. Data type

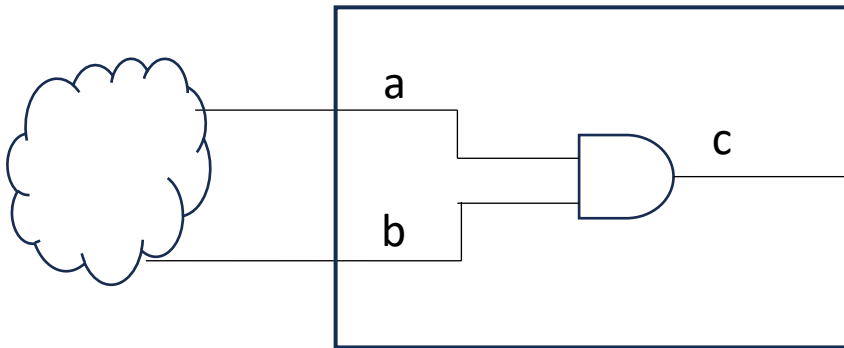
3. Assignment

4. Operator

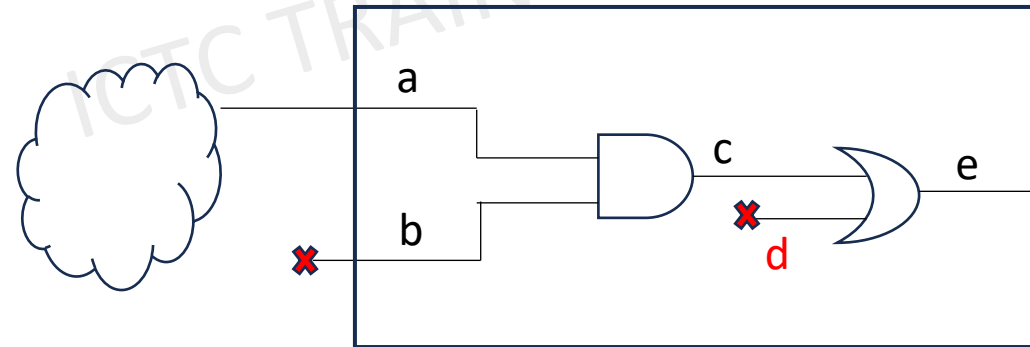
VERILOG DATA TYPES

Net

- Net data types are used to represent connections between hardware elements.
- Net data types do not hold values.
- Net data types have the value of their drivers. If a net has no driver, then net has high-impedance value (z)



All signals are net types and have drivers.



All signals are net types.

b and d have no driver

→ b & d are un-driven or floating net
and has z (high-impedance) value

VERILOG DATA TYPES

Net declaration

- Keyword: **wire**
 - Default: One-bit values, can declared as vectors
 - Default value: z (high-impedance)
 - Examples:
 - `wire a;`
 - `wire b,c;`
 - `wire [2:0] d; //vector`
 - `wire [4:2] e; //vector partial selection`



VERILOG DATA TYPES

Variable



- Variable data types can hold value during simulation.
- Some variable data types keyword:
 - **reg**: represent data storage elements. It can be synthesizable.
 - **integer**: represent 32-bit signed integer variable. It can be synthesizable.
 - **real**: represent 64-bit floating point variable. It can not be synthesizable.
 - **time**: 64-bit unsigned integer to store simulation time. It can not be synthesizable.

Reg data type will be described in detail in session 8 !!!

VERILOG DATA TYPES

Vector data type



- Vector is multibit net or reg data type with range specification.
- Net and register data types can be declared as vector.
- Syntax: **wire/reg** [msb_index:lsb_index] var_name;
- Examples:

wire [7:0] bus; //8-bit vector net type

reg [31:0] addr; //32-bit vector reg type

- Each bit of a vector can take any value: 0,1,z,x
- Each bit of a vector is accessible independently.

VERILOG DATA TYPES

Vector data type – part select

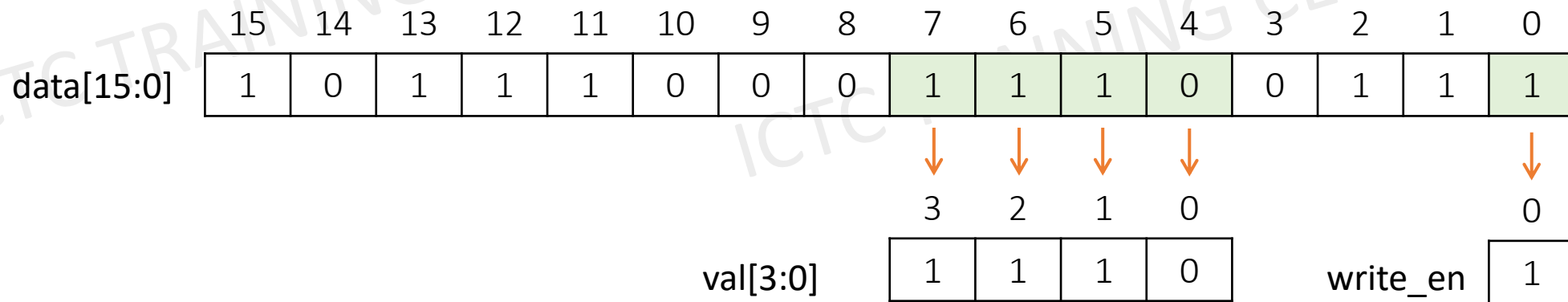


Example:

Having data[15:0] signal as below, perform the assignment for val[3:0] and write_en from data[15:0]

val[3:0] = data[7:4]

write_en = data[0]



VERILOG DATA TYPES

Vector data type – part select



Practice1: find the value of data signal

`data[15:12] = val[3:0]`

`data[7:4] = data[3:0]`

	3	2	1	0
val[3:0]	1	1	1	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
data[15:0]	0	1	0	1	1	0	0	0	1	1	1	0	0	1	1	1

VERILOG DATA TYPES

Vector data type – part select



Practice2: find the value of data[15:0] signal (net type)

data[17:14] = val[3:0]

																3 2 1 0							
																val[3:0]				1	0	0	1

VERILOG DATA TYPES

Vector data type – part select



Practice3: find the value of tmp[3:0] signal

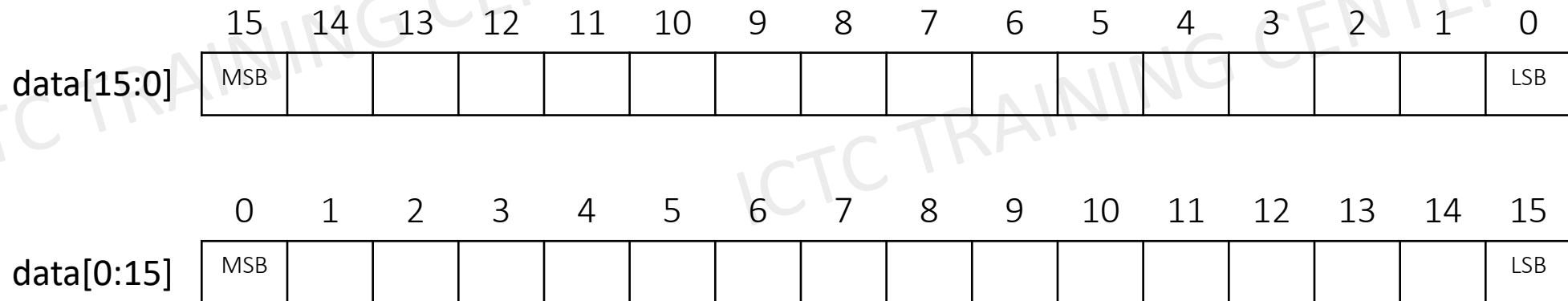
tmp[3:0] = data[17:14]

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
data[15:0]	1	1	1	0	1	0	0	0	1	1	1	0	0	1	1	1

VERILOG DATA TYPES

Vector data type

- Sometimes we can see the below declaration:
`wire/reg [0:15] data;`
- The difference is just the indexing and MSB/LSB position.



VERILOG DATA TYPES

Vector data type



- Practice:

```
reg [7:0] data_1;
```

```
reg [0:7] data_2;
```

```
data_1 = 8'b1000_1000;
```

```
data_2 = 8'b1000_1000;
```

What's the value of data_1[0] and data_2[0] ?

ICTC TRAINING CENTER

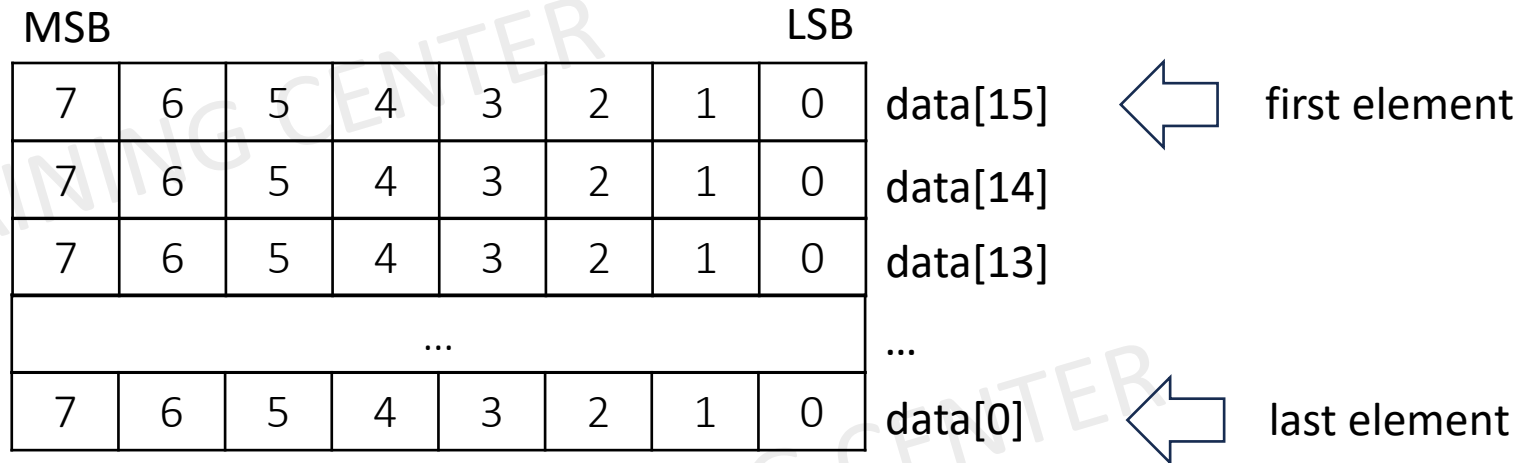
ICTC TRAINING CENTER

VERILOG DATA TYPES

Array data type

- Arrays can be used to group elements of a data type into multidimensional objects.
- Syntax: `<data_type> <MSB:LSB> <array_name> [first_element_idx:last_element_idx];`
- Example:

```
integer    count[7:0];           //array of 8 integer
reg        [7:0] data[15:0];      //array of 16 byte reg
```



- Only can access 1 element of the array at a time
 - data[0]: access to the last byte of the array
 - data[15][7]: access to the MSB of the first element of the array
 - data[2:0][5]: is illegal
- Multi-dimensional array is not mentioned in this course.
- Do not use array declaration unless it makes the code become simple and easy to understand.



VERILOG DATA TYPES

Array data type



- Memory are digital storage that help store a data and information in digital circuits.
- RAM and ROM are good example of such memory elements.
- Storage elements can be modeled using one-dimensional array of **reg** type.
- Syntax: **reg** [**WIDTH**-1:0] <name> [0:**DEPTH**-1];
WIDTH: width of each memory word (in bits)
DEPTH: depth of size of the memory (number of words)
- Can read from and write to specific locations in the memory using indexing
- Example:

```
reg [7:0] mem[0:1023]; //1024x8 memory
```

```
reg [7:0] tmp;
```

```
//write to memory
```

```
mem[2] = 8'b1011_0111;
```

```
tmp = mem[2]; //read mem[2], tmp's value is 8'b1011_0111
```

0	
1	
2	1011_0111
.	
.	
1023	



1. Verilog introduction

2. Data type

3. Assignment

4. Operator

VERILOG ASSIGNMENT

Continuous Assignment

- Continuous assignment: drive values onto nets. LHS must be net data type.

- Syntax: `assign y = expression;` //y is wire data type

Example:

```
wire [3:0] y;
```

```
assign y = a + b;
```

- Range is allowed in the assignment

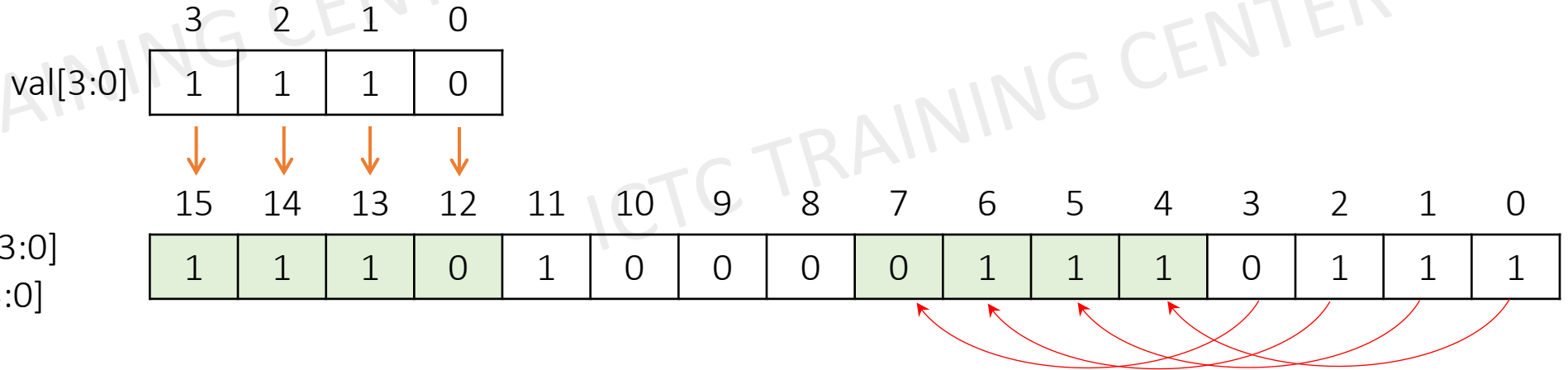
```
wire [15:0] data;
```

```
wire [3:0] val;
```

....

```
assign data[15:12] = val[3:0]
```

```
assign data[7:4] = data[3:0]
```



```
assign a = b & c;
```

```
assign b = d | e;
```



```
assign b = d | e;
```

```
assign a = b & c;
```

When d or e is updated, b will change. The change of b causes a change. The order of assignment does not affect the result.



VERILOG ASSIGNMENT

Procedural assignment

Procedure: a block of code that executes in-order.

This is just about the definition. It will be described more in later sessions.



Example of
procedure using
always block.
Will be learned
detail in session 8



```
reg y;  
always @( a or b or c)  
begin  
    if( a )  
        y = b;  
    else  
        y = c;  
end
```

```
reg clk;  
  
initial begin  
    clk = 0;  
    #25ns;  
    forever begin  
        clk = #25ns ~clk;  
    end  
end
```



Example of
procedure using
initial block.
Will be learned
detail in session 10

Assignment inside a procedure called procedural assignment.

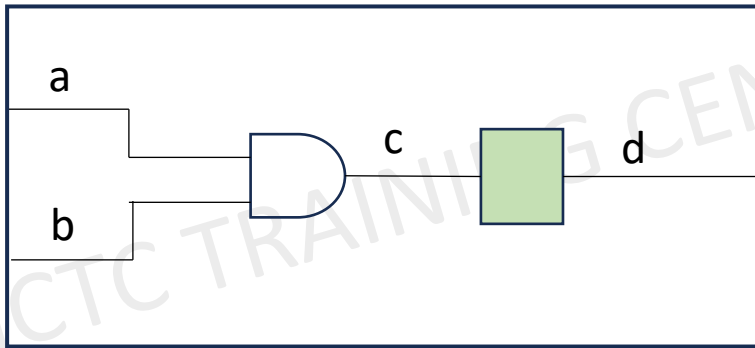
LHS of the procedural assignment need to be reg data type.

PRACTICE

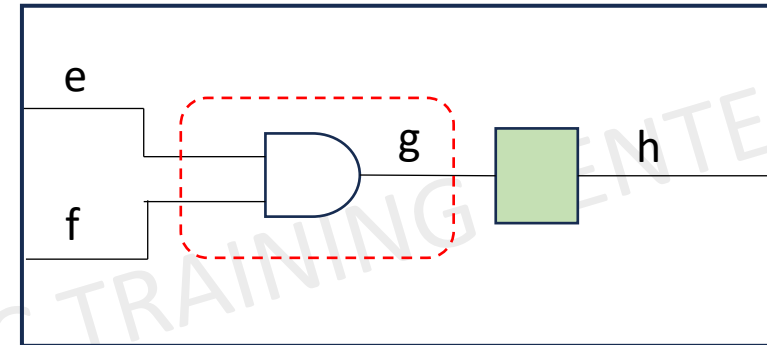
Practice: identify the data type for a,b,c,d,e,f,g,h in below cases



module1



module2



memory element



procedure



1. Verilog introduction

2. Data type

3. Assignment

4. Operator

VERILOG OPERATOR

Bitwise operator

- Bit-wise operators:

Syntax: <operand 1> *operator* <operand 2> (except the bitwise invert)



Operator	Name	Example
\wedge	Bitwise XOR	$4'b1010 \wedge 4'b0011 \rightarrow 4'b1001$
$\&$	Bitwise AND	$4'b1010 \& 4'b0011 \rightarrow 4'b0010$
$ $	Bitwise OR	$4'b1010 4'b0011 \rightarrow 4'b1011$
\sim	Bitwise invert	$\sim 4'b1010 \rightarrow 4'b0101$

VERILOG OPERATOR

Arithmetic operators

- Arithmetic operators:

Syntax: `<operand 1> operator <operand 2>`



Operator	Name	Example
+	Addition	$4'b1010 + 4'b0011 \rightarrow 4'b1101$
-	Subtraction	$4'b1010 - 4'b0011 \rightarrow 4'b0111$
*	Multiplication	$4'b0101 * 4'b0010 \rightarrow 4'b1010$
/	Division (non-synthesizable)	$4'b1010 / 4'b0010 \rightarrow 4'b0101$
%	Modulo (non-synthesizable)	$4'b1011 \% 4'b0010 \rightarrow 4'b0001$

VERILOG OPERATOR

Relational operators

- Relational operators:

Syntax: `<operand 1> operator <operand 2>`



Operator	Name	Example
>	Greater	4'b1010 > 4'b1000 → TRUE (1'b1)
>=	Greater or equal	4'b1010 >= 4'b1010 → TRUE (1'b1)
<	Less	4'b1010 < 4'b1001 → FALSE (1'b0)
<=	Less or equal	4'b1010 <= 4'b1010 → TRUE (1'b1)

VERILOG OPERATOR

Equality operators

- Equality operators:

Syntax: <operand 1> *operator* <operand 2>



Operator	Name	Example
==	Logical equality	4'b1100 == 4'b1100 → 1'b1 4'b1100 == 4'b110x → UNKNOWNED (1'bx)
!=	Logical inequality	4'b1100 != 4'b1100 → 1'b0 4'b1100 != 4'b110x → UNKNOWNED (1'bx)
===	Case Equality	4'b1100 === 4'b1100 → TRUE (1'b1) 4'b1100 === 4'b110x → FALSE (1'b0)
!==	Case inequality	4'b1100 !== 4'b1100 → FALSE (1'b0) 4'b1100 !== 4'b110x → TRUE (1'b1)

VERILOG OPERATOR

Logical operators

- Logical operators:

Syntax: <operand 1> *operator* <operand 2>

Operator	Name	Example
&&	Logical and	$a = 5, b = 3 \rightarrow ((a == 5) \&\& (b == 3)) = \text{TRUE} (1'b1)$ $4'b1010 \&\& 4'b0001 \rightarrow \text{TRUE} (1'b1)$
	Logical or	$a = 5, b = 3 \rightarrow ((a != 5) (b != 3)) = \text{FALSE} (1'b0)$ $4'b0010 4'b0001 \rightarrow \text{TRUE} (1'b1)$
!	Logical invert	$!1'b1 = 1'b0$ $!4'b0001 \rightarrow \text{FALSE} (1'b0)$
<<	Logical shift left	$1 \ll 2 \rightarrow 4'b0100$
>>	Logical shift right	$4'b1101 \gg 2 \rightarrow 4'b0011$

- ❑ One operand has one bit value is 1 \rightarrow its value is 1 seen by the logical operator
- ❑ One operand has all bit value is 0 \rightarrow its value is 0 seen by the logical operator
- ❑ If above 2 cases are wrong \rightarrow its value is x by the logical operator



VERILOG OPERATOR

Unary reduction operators

- Unary reduction operators:
Syntax: *operator* <operand 1>

Operator	Name	Example
&	Reduction and	$\& a[3:0] \rightarrow a[3] \& a[2] \& a[1] \& a[0]$ $a[3:0] = 4'b1011 \rightarrow \&a[3:0] = 0$
	Reduction or	$ a[3:0] \rightarrow a[3] a[2] a[1] a[0]$ $a[3:0] = 4'b1000 \rightarrow a[3:0] = 1$
^	Reduction xor	$\^ a[3:0] \rightarrow a[3] \^ a[2] \^ a[1] \^ a[0]$ $a[3:0] = 4'b1010 \rightarrow \^a[3:0] = 0$

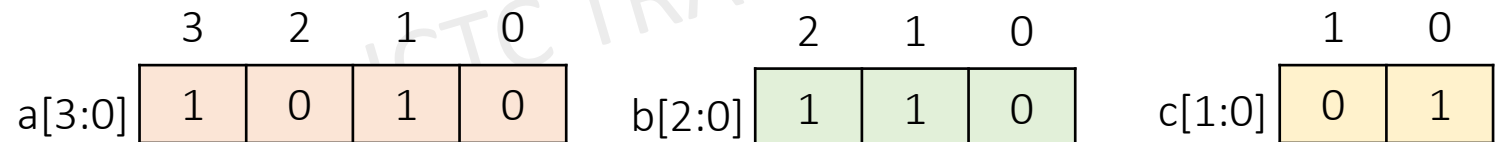


VERILOG OPERATOR

Concatenation operators

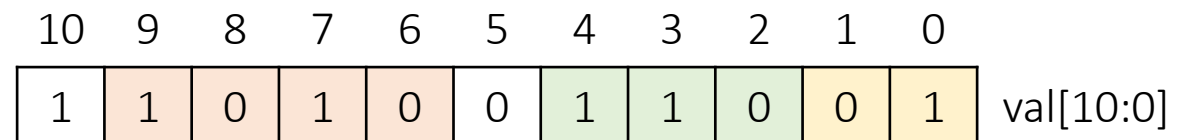
■ Concatenation Operators

Operator	Name	Example
{ }	Concatenation	$\{a[3:0], b[2:0]\} \rightarrow$ to create a 7bit signal that has bit range [2:0] equal to b[2:0] and bit range [6:3] equal to a[3:0] $a = 4'b1100$ $b = 3'b001$ $\rightarrow \{a[3:0], b[2:0]\} = 7'b1100_001$
{m, { }}	Replication	$\{4 \{2'b10\}\} \rightarrow 8'b10_10_10_10$



wire [10:0] val;

assign val[10:0] = {1'b1, a[3:0], 1'b0, b[2:0], c[1:0]}



VERILOG OPERATOR

Concatenation operators

Practice:

Find the result of following expressions:

1. $\{ \{1'b1\}, \{5\{1'b0\}\}, \{4\{1'b1\}\}, \{1'b0\} \}$

2. $\{ \{3\{2'b10\}\}, \{5'h8\} \}$



VERILOG OPERATOR

Conditional operators



- Conditional Operators
- Syntax: (cond)? (result if cond true) : (result if cond false)
- Example:

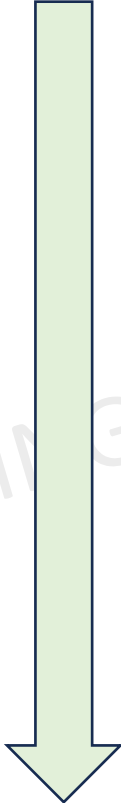
assign y = (s==1)? a : b;

→ y is assigned to a if s is 1

→ y is assigned to b if s is 0

VERILOG OPERATOR

Precedence

Operators	Symbol	Precedence
Unary	+ - ! ~ & ^	 <p>Highest</p> <p>Lowest</p>
Arithmetic	* / %	
	+ -	
Shift	<< <<< >> >>>	
Relational	< <= > >=	
Equality	== != === !==	
Bitwise	&	
	^	
Logical operator	&&	
Conditional operator	? :	
Assignment operator	= <=	



SESSION 6

SUMMARY



SUMMARY:

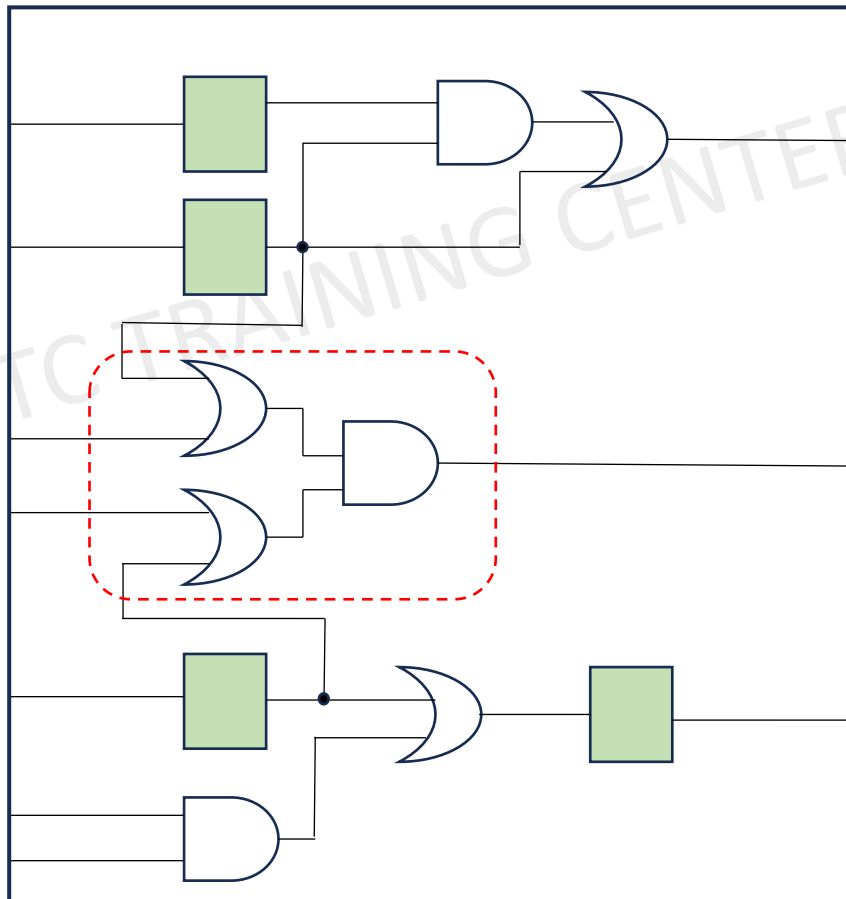
- ☐ There are 4 basic value in Verilog: 0,1,x,z. “x” does not exist in actual silicon.
- ☐ Net data types are used to represent connections between hardware elements and can not hold value. Use “wire” for net data type declaration.
- ☐ Reg data types are used to represents storage elements or combinational logic in procedure. Use “reg” for reg data type declaration.
- ☐ Multiple bits are group into vectors and arrays
- ☐ Need to take care Verilog precedence to avoid issues during simulation

HOMEWORK

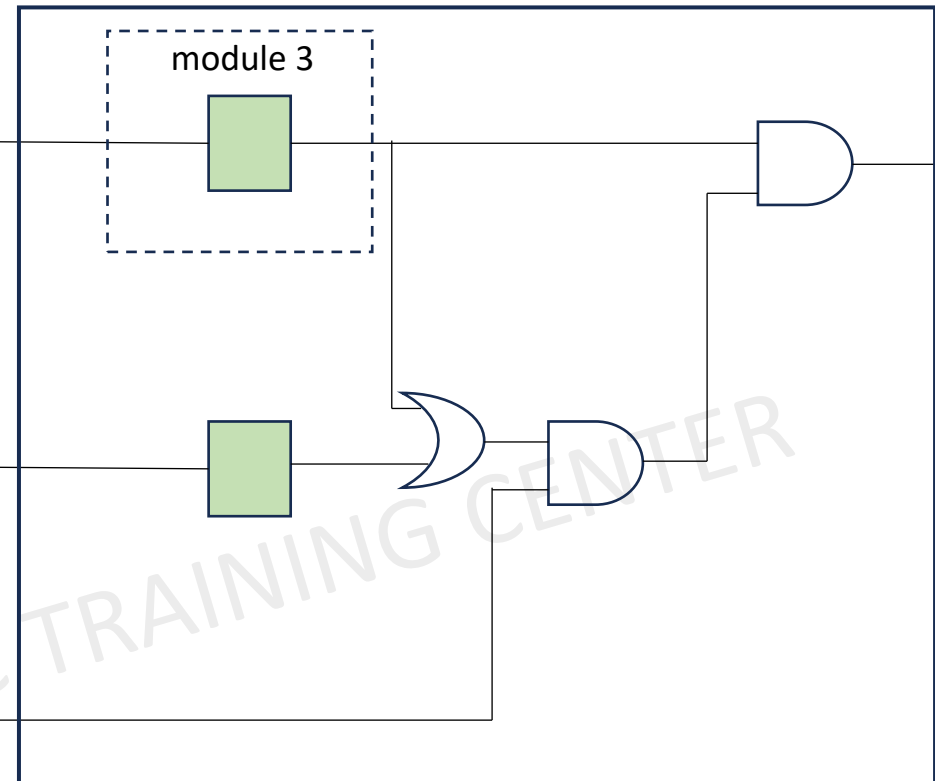
Homework1: identify data type wire/reg for below diagram.
Submit the result via your homework submit form.




module 1



module 2



 procedure

HOMEWORK

- Under your home directory, mkdir 06_ss6, copy lab3 of ss5 to 06_ss6 and modify the test_bench.v only
- Snapshot the result of the homework after running and submit to your form. Mentor will check your code in your dir.

Homework2(*): Perform all the Verilog operator examples in our today's session using testbench of lab3-ss5. Check the result of the testbench and make sure they're same as the result in our slides.

Output of testbench should be similar to the picture on the right.

Operator	Name	Example
^	Bitwise XOR	4'b1010 ^ 4'b0011 → 4'b1001
&	Bitwise AND	4'b1010 & 4'b0011 → 4'b0010
	Bitwise OR	4'b1010 4'b0011 → 4'b1011
~	Bitwise invert	~4'b1010 → 4'b0101

```
# Loading sv_std.sv
# Loading work.test_bench(fast)
# ** Note: (vsim-8900) Creating design debug database vsim.dbg.
# log -r /*
# run -all
# 01.Bitwise XOR      : 4'b1010 ^ 4'b0011      = 4'b1001
# 02.Bitwise AND      : 4'b1010 & 4'b0011      = 4'b0010
# 03.Bitwise OR       : 4'b1010 | 4'b0011      = 4'b1011
# 04.Bitwise INV      : ~4'b1010          = 4'b0101
# 05.Addition         : 4'b1010 + 4'b0011      = 4'b1101
# 06.Substraction     : 4'b1010 - 4'b0011      = 4'b0111
# 07.Multiplication   : 4'b0101 * 4'b0010      = 4'b1010
# 08.Division         : 4'b1010 / 4'b0010      = 4'b0101
# 09.Modulo           : 4'b1011 % 4'b0010      = 4'b0001
# 10.Greater          : 4'b1010 > 4'b1000      = 1'b1
# 11.Greater or Equal : 4'b1010 >= 4'b1010      = 1'b1
# 12.Less             : 4'b1010 < 4'b1001      = 1'b0
# 13.Less or Equal    : 4'b1010 <= 4'b1010      = 1'b1
# 14.1.Logical Equality : 4'b1100 == 4'b1100      = 1'b1
# 14.2.Logical Equality : 4'b1100 == 4'b110x      = 1'bx
# 15.1.Logical Inequality : 4'b1100 != 4'b1100      = 1'b0
# 15.2.Logical Inequality : 4'b1100 != 4'b110x      = 1'bx
# 16.1.Case Equality   : 4'b1100 === 4'b1100     = 1'b1
# 16.2.Case Equality   : 4'b1100 === 4'b110x     = 1'b0
# 17.1.Case Inequality  : 4'b1100 !== 4'b1100     = 1'b0
# 17.2.Case Inequality  : 4'b1100 !== 4'b110x     = 1'b1
# 18.1.Logical And     : a=5,b=3 == ((a==5) && (b==3)) = 1
# 18.2.Logical And     : 4'b1010 && 4'b0001      = 1'b1
# 19.1.Logical OR      : a=5,b=3 == ((a!=5) || (b!=3)) = 0
# 19.2.Logical OR      : 4'b0010 && 4'b0001      = 1'b1
# 20.1.Logical Invert   : !1'b1          = 1'b0
# 20.2.Logical Invert   : !4'b0001        = 1'b0
# 21.Logical Shift Left : 1 << 2          = 4'b0100
# 22.Logical Shift Right : 4'b1101 >> 2     = 4'b0011
# 23.Reduction And     : a=4'b1011 --> &a[3:0] = 1'b0
# 24.Reduction Or      : a=4'b1000 --> |a[3:0] = 1'b1
# 25.Reduction XOR     : a=4'b1010 --> ^a[3:0] = 1'b0
# 26.Concatenation     : a=4'b1100, b=3'b001 --> {a[3:0],b[2:0]} = 7'b1100001
# 27.Replication       : {4 {2'b10}}          = 8'b10101010
# ** Note: $finish      : ../tb/test_bench.v(55)
# Time: 203 ns Iteration: 0 Instance: /test_bench
# End time: 00:58:20 on Aug 24,2024, Elapsed time: 0:00:00
```

