Verilog HDL – I
Combinational Logic Circuit

黃元豪
Yuan-Hao Huang

國立清華大學電機工程學系
Department of Electrical Engineering
National Tsing-Hua University

Outline

- Introduction
- Verilog Modules
- Verilog HDL Coding
- Examples of Combinational Circuits

Introduction

- Verilog HDL是一種高階且模組化的硬體描述語言
  - 完整功能：整合電路描述、合成與模擬驗證之設計功能。
  - 彈性設計：模組化設計，易做重新組合，容易設計。
  - 多種描述型式：電路連接關係，順序性與共時性敘述，布林代數式等。
  - 跨平台可攜性：可用不同編譯軟體編譯，適用不同工作平台與製程。
  - 容易學習：語法與C 語言相似。
- VHDL為另一個類似的硬體描述語言。

Verilog HDL

- Verilog 硬體描述語言(Verilog Hardware Description Language)
  - 在積體電路設計（特別是超大型積體電路的計算機輔助設計）的電子設計自動化領域中，**Verilog HDL**是一種用於描述、設計電子系統（特別是數位電路的硬體描述語言。
  - Verilog HDL是電力電子工程師學會（IEEE）1364號標準。

- Verilog 硬體描述語言在邏輯設計上的用途
  - 用途一: 邏輯電路設計的模擬與驗證
  - 用途二: FPGA邏輯電路(**IC電路)的設計與實作

    ** IC電路的設計與實作在【EE4292積體電路設計實驗】教授。

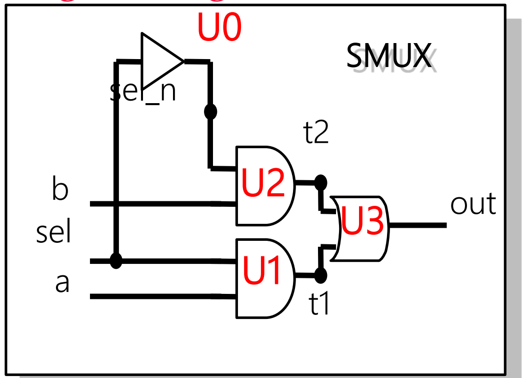
Verilog HDL Utilization Scenario I

- Simulation and verification of logic circuits on PC.

Specification

2-to-1 Multiplexer

Logic Design



Verilog HDL Coding

```
module SMUX(out, a, b, sel);

output out;
input a,b,sel;
wire sel_n,t1,t2;

not U0(sel_n,sel);
and U1(t1,a,sel);
and U2(t2,b,sel_n);
or    U3(out,t1,t2);

endmodule
```

Test Pattern

a, b, sel
000
001
010
…
111

Verilog HDL Simulator

- Imaging what happens when the circuit is as complex as a CPU or MP3 player processor.

Verilog HDL Utilization Scenario II

- Design and Implementation of logic circuits in FPGA (IC)

Specification

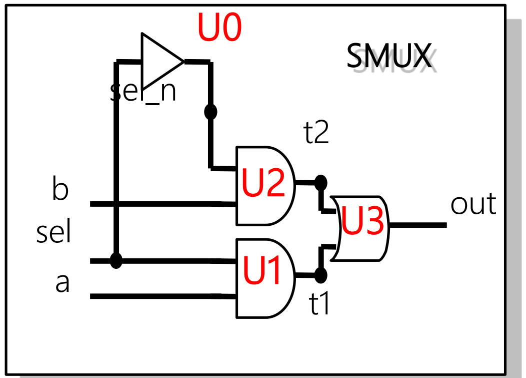
2-to-1 Multiplexer

Verilog HDL Coding

```
module SMUX(out, a, b, sel);

output out;
input a,b,sel;
wire sel_n,t1,t2;

 assign out = sel ? a:b;

endmodule
```

Verilog HDL Logic **Synthesizer**

Synthesized Logic Circuits



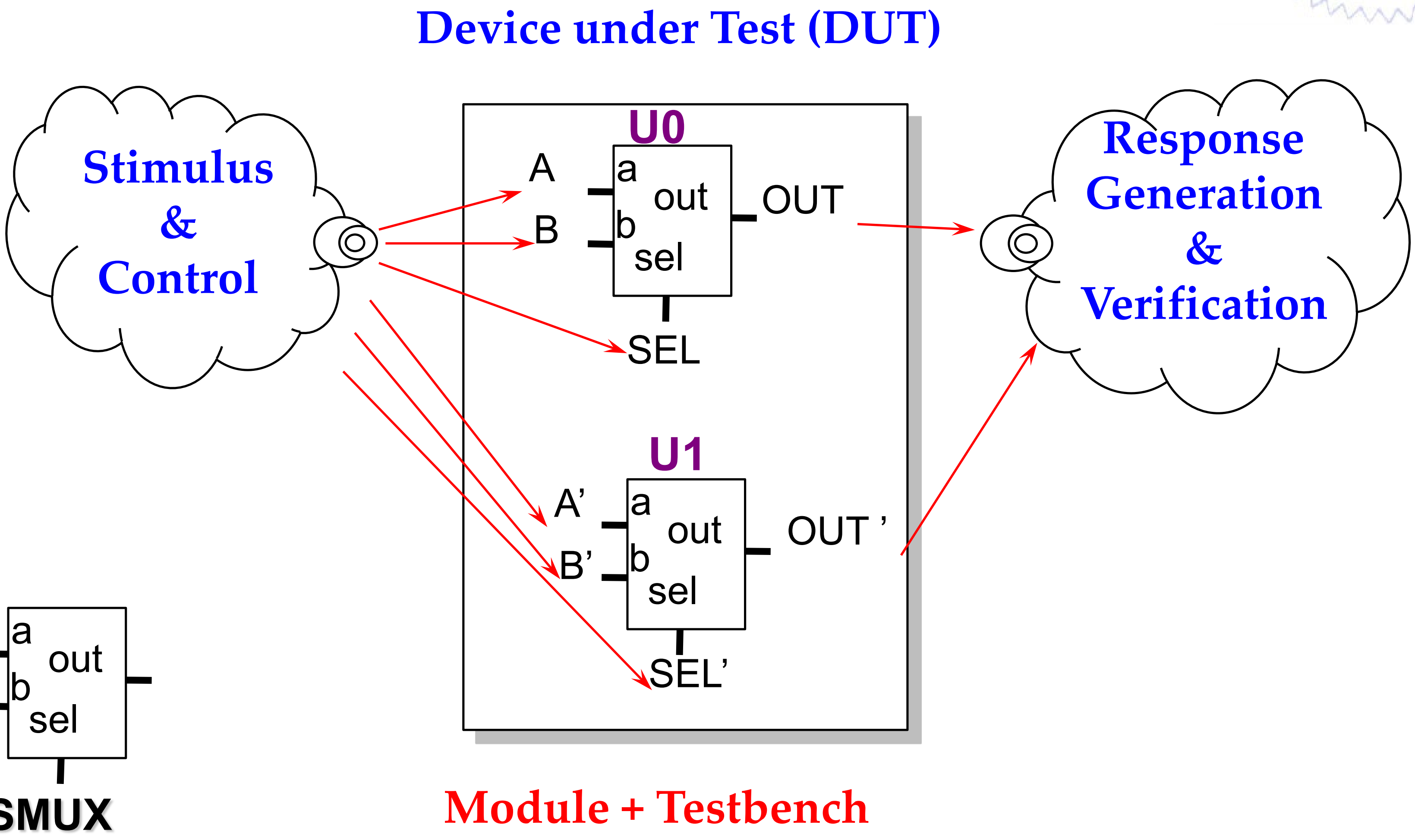
FPGA **Implement Design** and **Programming**

Now, it can work !

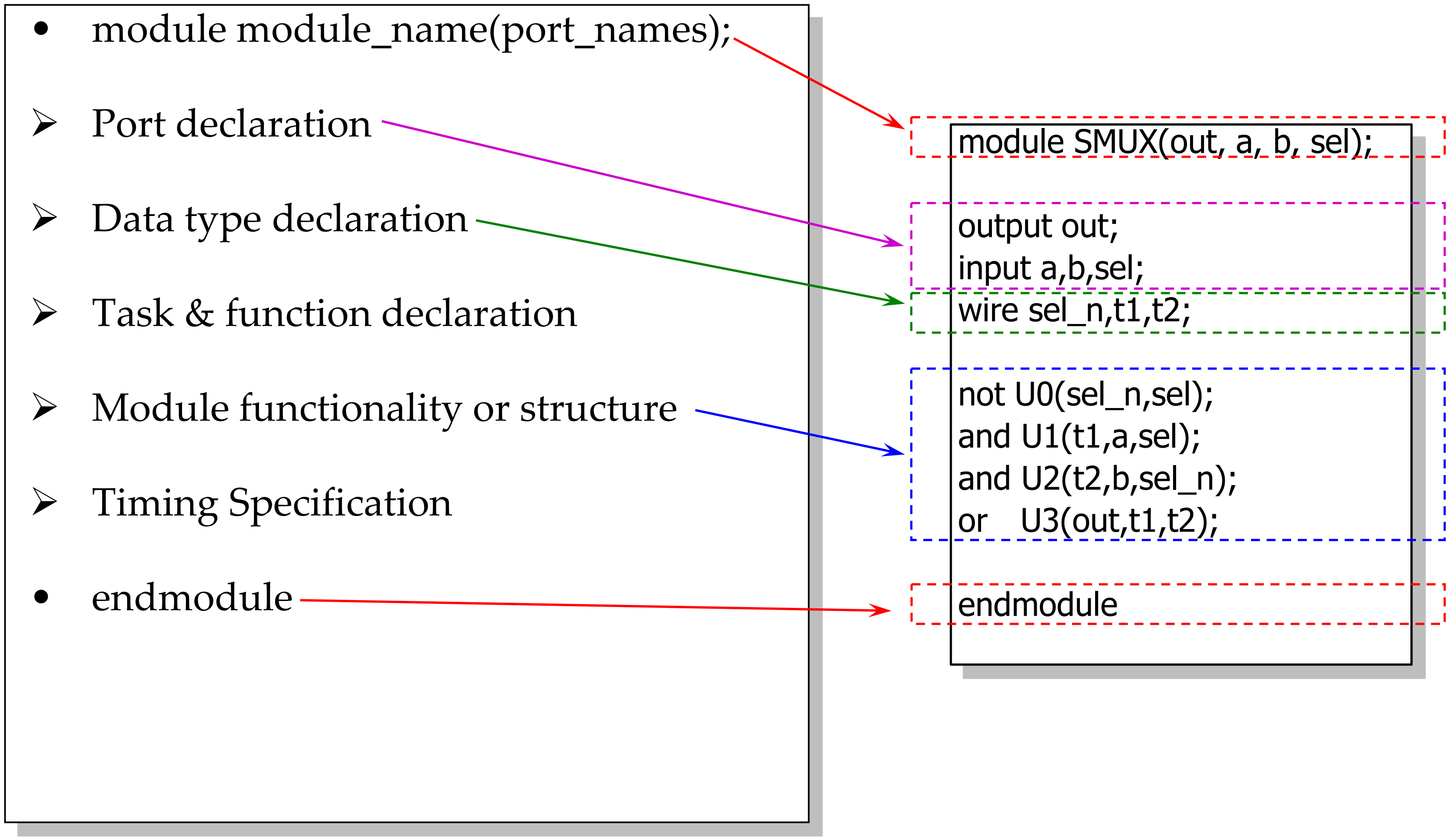
Outline

- Introduction
- Verilog Modules
- Verilog HDL Coding
- Examples of Combinational Circuits

Verilog Module Architecture



Verilog Module



- module module_name(port_names);
  - Port declaration
  - Data type declaration
  - Task & function declaration
  - Module functionality or structure
  - Timing Specification
- endmodule

```
module SMUX(out, a, b, sel);

output out;
input a,b,sel;
wire sel_n,t1,t2;

not U0(sel_n,sel);
and U1(t1,a,sel);
and U2(t2,b,sel_n);
or   U3(out,t1,t2);

endmodule
```

Testbench (1/4)

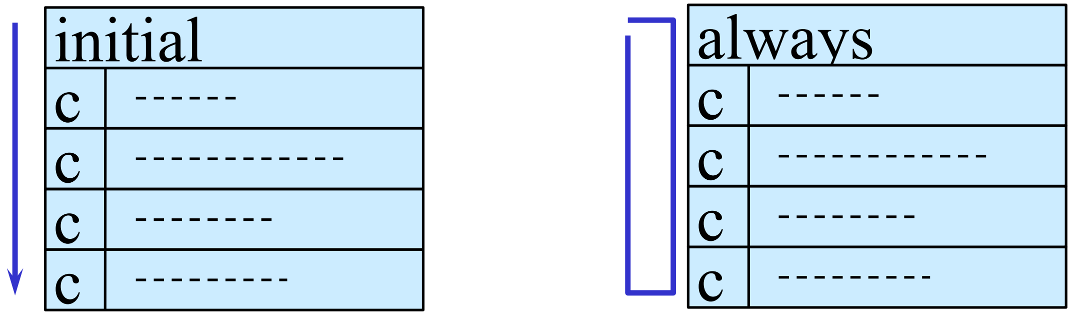
- module testfixture;
  - Declare signals
  - Instantiate modules
  - Applying stimulus
  - Monitor signals
- endmodule

Testbench (2/4)

- Declare signals
  - Test pattern must be stored in storage elements first and then apply to DUT (Device under Test)
    - Use "reg" to declare the storage element
- Instantiate modules
  - Both behavioral level or gate level model can be used.

Testbench (3/4)

- Describing Stimulus
  - The testbench always be described behaviorally.
  - Procedural blocks are bases of behavioral modeling.
  - The simulator starts executing all procedure blocks at time 0 and executes them concurrently.
  - Two types of procedural blocks
    - initial
    - always



Testbench (4/4)

```
•   module test_SMUX;

•   reg                   A,B,SEL;
•   wire     OUT;

•   SMUX U0(.out(OUT),.a(A),.b(B),.sel(SEL));

•   initial
•   begin
•            A=0;B=0;SEL=0;
•            #10        A=0;B=1;SEL=1;
•            #10        A=1;B=0;
•            #10        SEL=0;
•            .....
•            #10        SEL=1;
•   end

•   endmodule
```

Declare signals

Make an instance

Assign values to storage elements

#10 to specify 10 time unit delay

Example: always

```
initial clk = 0;
always #10 clk = ~clk;
```

```
initial clk=0;
always
begin
clk = 0;
#10;
clk = 1;
#10;
end
```

Outline

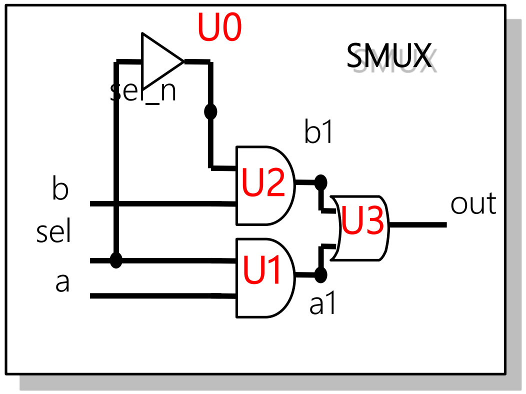
- Introduction
- Verilog Modules
- Verilog HDL Coding
- Examples of Combinational Circuits

Verilog HDL Coding Tokens

- Verilog HDL coding tokens
  - White space (空白)
  - Comments (註解)
  - Keyword (關鍵字)
  - Identifier (識別子)
  - Operator (運算子)
  - Number (數字)
  - ….

White Space and Comment

- White space makes code more readable
  - Include blank space (\b), tabs (\t), and carriage return (\n).
- Comments
  - /* … */ : mark more than one line
  - // : mark only one line.



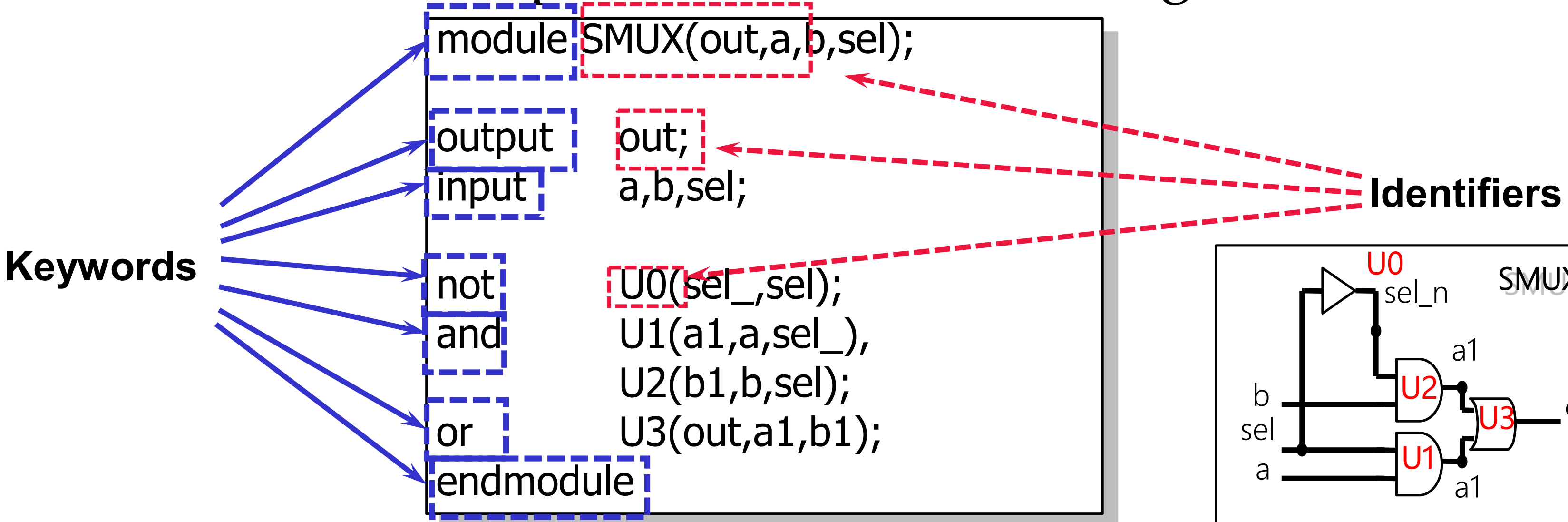
```
module SMUX(out,a,b,sel);
output      out;
input       a,b,sel;
// The following are logic gates
not         U0(sel_,sel);
and         U1(a1,a,sel_),
            U2(b1,b,sel);
or          U3(out,a1,b1);   /* This
     comment marks more than one
     line*/

endmodule
```

Keywords

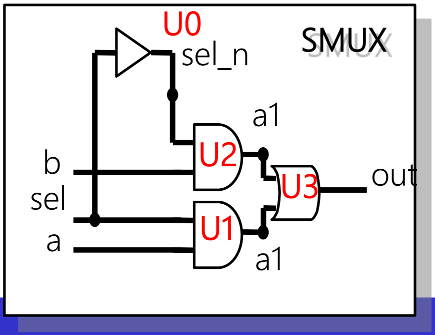
- Pre-defined non-escaped identifiers that used to define the language construct.
- All keywords are defined in lower cases.
- Examples
  - module, endmodule
  - input, output, inout, wire
  - reg, integer, real, time
  - not, and, or, nand, nor, xor
  - parameter
  - begin, end
  - fork, join, assign
  - always, for, if, else
  - negedge, posedge
- Verilog is a case sensitive language.
  - Use "-u" option in command line option for case-insensitive.

Identifiers

- Names of modules, ports, and instances are identifiers.
- Identifiers are **user-defined** names for Verilog objects within a description.
- Legal characters in identifiers:
  - a-z, A-Z, 0-9, _, $
- The first character of an identifier must be an alphabetical character (a-z, A-Z) or an underscore (_).
- Identifiers can be up to 1023 characters long.



```
module SMUX(out,a,b,sel);

output    out;
input     a,b,sel;

not       U0(sel_,sel);
and       U1(a1,a,sel_),
          U2(b1,b,sel);
or        U3(out,a1,b1);
endmodule
```



Operators (1/3)

Bitwise Operators		
OP	**Usage**	**Description**
~	~m	Invert each bit of m
&	m & n	AND each bit of m with each bit of n
|	m | n	OR each bit of m with each bit of n
^	m ^ n	Exclusive OR each bit of m with n
~^ or ^~	m ~^ n or m ^~ n	Exclusive NOR each bit of m with n

Unary Reduction Operators		
OP	**Usage**	**Description**
&	&m	AND all bits in m together (1-bit result)
~&	~&m	NAND all bits in m together (1-bit result)
|	|m	OR all bits in m together (1-bit result)
~|	~|m	NOR all bits in m together (1-bit result)
^	^m	Exclusive OR all bits in m (1-bit result)
~^ or ^~	~^m or ^~m	Exclusive NOR all bits in m (1-bit result)

Operators (2/3)

Arithmetic Operators		
OP	Usage	Description
+	m + n	Add n to m
-	m - n	Subtract n from m
-	-m	Negate m (2's complement)
*	m * n	Multiply m by n
/	m / n	* Divide m by n
%	m % n	* Modulus of m / n

*** Synthesis not supported : The divisor for divide operator may be restricted to constants and a power of 2**

Logical Operators		
OP	Usage	Description
!	!m	Is m not true? (1-bit True/False result)
&&	m && n	Are both m and n true? (1-bit True/False result)
||	m || n	Are either m or n true? (1-bit True/False result)

Equality Operators (compares logic values of 0 and 1)		
OP	Usage	Description
==	m == n	Is m equal to n? (1-bit True/False result)
!=	m != n	Is m not equal to n? (1-bit True/False result)

Identity Operators (compares logic values of 0, 1, x, and z)		
OP	Usage	Description
===	m === n	* Is m identical to n? (1-bit True/False result)
!==	m !== n	* Is m not identical to n? (1-bit True/False result)

Synthesis not supported

Synthesis not supported

Operators (3/3)

Relational Operators		
OP	Usage	Description
<	m < n	Is m less than n? (1-bit True/False result)
>	m > n	Is m greater than n? (1-bit True/False result)
<=	m <= n	Is m less than or equal to n? (True/False result)
>=	m >= n	Is m greater than or equal to n? (True/False result)

Logical Shift Operators		
OP	Usage	Description
<<	m << n	Shift m left n-times
>>	m >> n	Shift m right n-times

Misc Operators		
OP	Usage	Description
? :	sel?m:n	If sel is true, select m: else select n
{}	{m,n}	Concatenate m to n, creating larger vector
{{}}	{n{m}}	Replicate m n-times

Integer and Real Numbers

- Numbers can be integer or real numbers.
- Integer can be sized or unsized. Sized integer can be represented as
  - <size>’<base><value>
    - size : size in bits
    - base : can be b(binary), o(octal), d(decimal), or h(hexadecimal)
    - value : any legal number in the selected base and x, z, ?.
- Real numbers can be represented in decimal or scientific format.

Integer and Real Numbers

- 16 : 32 bits decimal
- 8’d16
- 8’h10
- 8’b0001_0000
- 8’o20
- 32’bx : 32 bits x
- 2’b1? : ? represents a high impedance bit
- 6.3
- 5.3e-4
- 6.2e3

Concatenation and Replication Operators

- Bit replication for 01010101
  - assign byte = {4{2'b01}};
- Sign extension
  - wire [7:0] byte;
  - assign word = {{8{byte[7]}},byte};

Input/Output Ports Declaration

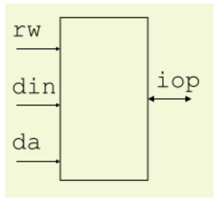
- Input/output port declaration
  - **input** : 輸入埠
  - **output** : 輸出埠
  - **inout :** 雙向埠具輸出輸入特性

Input/Output Ports

```
module MUXS(out,a,b,sel);
output    out;
input     a,b,sel;
// The following are logic gates
not       U0(sel_,sel);
and       U1(a1,a,sel_),
          U2(b1,b,sel);
or        U3(out,a1,b1);
endmodule
```

Inout Ports

```
module bidir(rw, din, da, iop);
input  rw, din, da;

inout  iop;

assign iop = rw ? ((din) ? da : iop): 1'bz;

endmodule
```



Data Types and I/O Ports

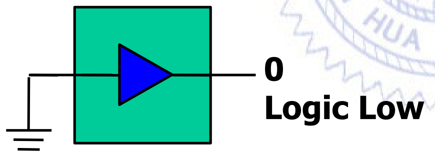
- Verilog basic data types
  - **wire :** 一條接線(只能被動顯示硬體運算結果，無法設定特定值)
  - **wand :** wired AND 接線 (FPGA not support)
  - **wor :** wired OR 接線 (FPGA not support)
  - **reg :** 暫存接線(主要來用來儲存設定特定值)
  - **integer (32-bit signed), real, float, time:** 特性如reg，但只能用於test bench module的運算。
- 使用於所設計的邏輯模組內的通常只有**wire (net property)**與**reg (reg property)**。
- input/output/inout 輸出入埠預設為**wire**的特性。

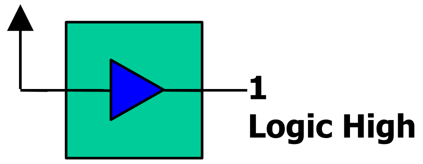
Characteristics of reg and wire

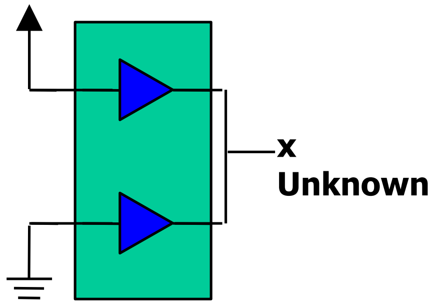
- wire
  - 資料型態為wire 的變數為連接硬體元件之連接線。
  - 變數必須被驅動，才能改變它的內容。
  - 除非被宣告為一向量，否則wire 型態的變數內定為一個位元的值，且其內定值為z。
  - 使用wire 所宣告的變數必須配合assign 敘述來改變其值，不能在always 區塊中作為敘述的等號左值 。
- reg
  - 宣告為資料型態reg 之變數的功能和一般程式語言中的變數類似，可以直接給定一個數值。
  - 除非被宣告為一向量，否則reg 型態的變數內定為一個位元的值，且其值為X。
  - 使用reg 所宣告的變數必須使用在always 區塊中作為敘述的等號左值。

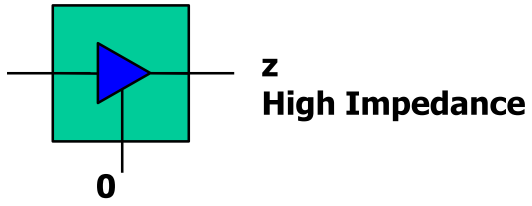
Value Sets

- 4-value logic system in Verilog
  - 0: Logic 0, Zero, False, Ground, Vss, Negative
  - 1: Logic 1, One, True, Power, Vdd, Positive
  - X: Unknown value
  - Z: High Impedance, Floating State, Tri-State, Disable driver.









Behavior Modeling

- Behavior model concerns only the function of the module without considering the operation of the circuits.
- Verilog is an event-driven timing control language.
  - Events:
    - Value change of reg or wire
    - Value change of input port
  - Synthesizable description of event control
    - Positive edge, negative edge, and value of changes of signals

always Description

- **always** monitors the change of input signals and update the output signals once the input changes.
  - **begin/end** are used when multiple statements respond to a specific event.

```
always@(event description)
logic statement;
```

```
always@(event description)
begin
logic statement 1;
logic statement 2;
end
```

- Event description
  - Single signal (level trigger)

```
always@(in1)
```

  - Multiple signals (level trigger)

```
always@(in1 or in2 or in3)
```

  - Edge trigger event (edge trigger)

```
always@(posedge clk)
```

```
always@(posedge clk or negedge reset)
```

if or if-else Description

- **if** is a conditional operation that checks the logic value (1 or 0)(true or false) to perform the following statement.
- **if** must be placed inside the **always** statement.
  - If always@(event) is level-triggered, if(expression) is level-triggered.
  - If always@(event) is edge-triggered, if(expression) is edge-triggered.
- **begin/end** are used when multiple statements are determined by a specific expression.

```
if(expression)
statement;
```

```
if(expression)
begin
statement1;
statement2;
end
```

```
if(expression)
statement1_true;
else
statement1_false;
```

```
if(expression)
begin
statement1_true;
statement2_true;
end
else
begin
statement1_false;
statement2_false;
end
```

case Description

- **case** is a conditional operation that checks multiple cases for perform the multiple statements in a tabular form.
  - If all conditions are stated, it is called "full case"; otherwise, it is not "full case"
  - The **default** case and statement are optional, but should be always included for circuit stability even if it is "full case".

```
case (signal)
case0: statement0;
case1: statement1;
…
default: def_statement;
endcase
```

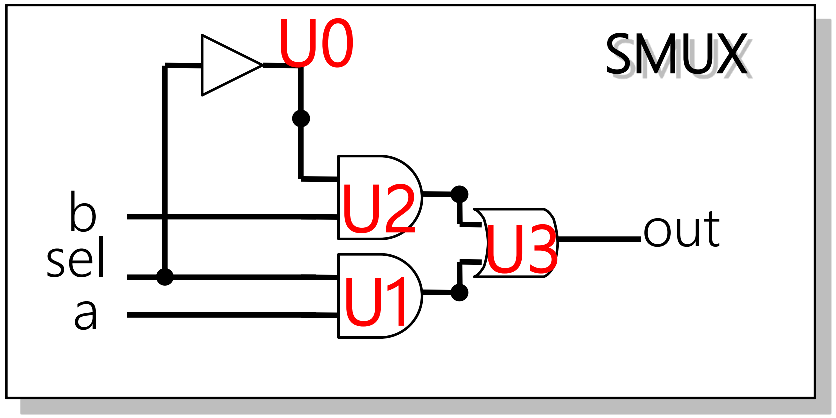
```
module decoder(X, out);
input  [0:1] X;
output [0:3] out;
reg    [0:3] out;
always@(X)  begin
case (X)
2'b00: out = 4'b0001;
2'b01: out = 4'b0010;
2'b10: out = 4'b0100;
default: out = 4'b0000;
endcase
end
endmodule
```

```
module decoder(X, out);
input  [0:1] X;
output [0:3] out;
reg    [0:3] out;
always@(X)  begin
case (X)
2'b00: out = 4'b0001;
2'b01: out = 4'b0010;
2'b10: out = 4'b0100;
2'b11: out =  4'b1000
```
~~default: out = 4'b0000;~~ $\rightarrow$ instable
```
endcase
end
endmodule
```

Example (1/3)

- **Continuous Assignment**
  - ? : ; Conditional operators (multiplexer)

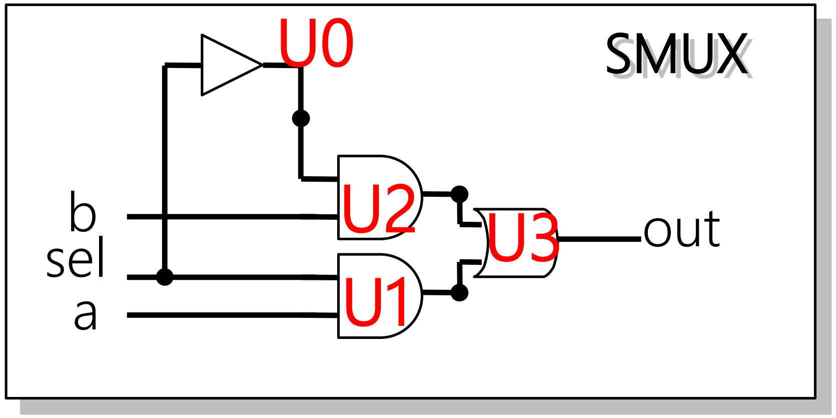
```
module SMUX(out,a,b,sel);
output out;
input a,b,sel;

assign out = (sel) ? a : b ;

endmodule
```



Example (2/3)

- **if-else** statements
  - Not for large multiplexers

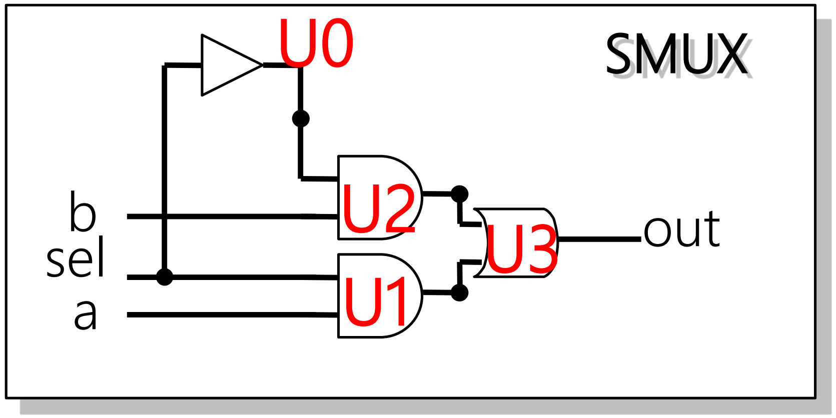
```
module SMUX(out,a,b,sel)
output out;
input a,b,sel;
reg out;

always @(sel or a or b)
  if (sel)
    out = a;
  else
    out = b ;

endmodule
```



Example (3/3)

- **case** statements
  - Usually for large multiplexers

```
module SMUX(out,a,b,sel);
output out;
input a,b,sel;

reg output;
always @(sel or a or b)
  case (sel)
    1'b0: out = b;
    1'b1: out = a;
  endcase

endmodule
```



Declaration of Vectors

- Vector declarations
  - reg <range>   <name> ,…,<name>;
  - Wire <range> <name>,…,<name>;
  - Example
    - reg   [3:0]   a,b;
    - reg   [0:3]   c;
    - wire [7:0]   in1;
    - wire [0:7]   in2;

Bit/Partial Bits Selection

- Bit Selection

```
wire  [7:0] a, b, c;
assign c[0] = a[0] & b[0];
```

- Partial Bits Selection

```
wire  [7:0] a, b, c;
assign c[3:0] = a[5:2] & b[7:4];
```

- Replication

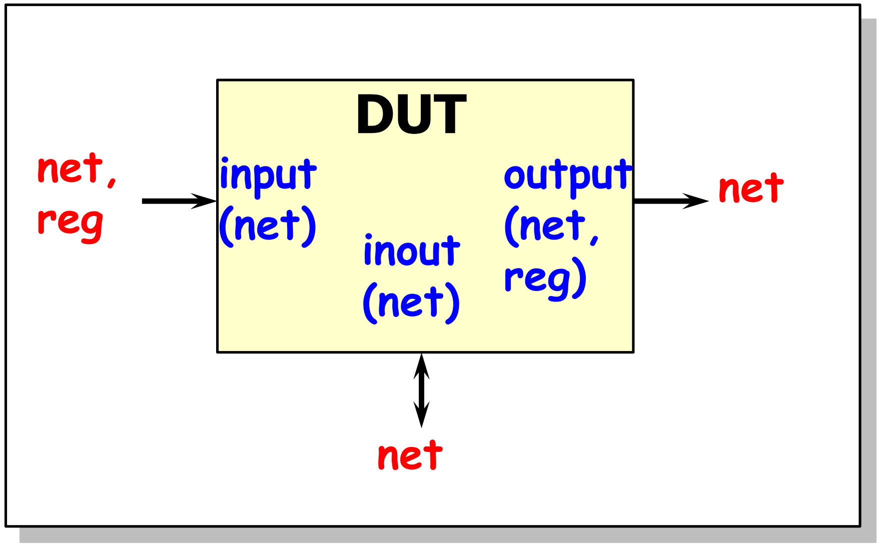
```
wire  [7:0] a, b, c;
assign c = {{2{a[7:6]}},{4{b[2]}}};
```

- Concatenation

```
wire  [7:0] a, b, c;
assign c = {a[3:0],b[5:2]};
```

Port Connection Between Modules

- An **input** or **inout** port must be a net (wire).
- An **output** port can be a register.
- A signal assigned a value in a procedural block (always, initial) must be a register data type.



Common Mistakes in Choosing Data Types

- Make a procedural assignment to a net
  - wire [7:0] databus;
  - always @(read or addr) databus=read ? mem[addr] : 'bz;
  - Illegal left-hand-side assignment
- Connect a register to an instance output
  - reg myreg;
  - and (myreg, net1, net2);
  - Illegal output port specification
- Declare a module **input** port as a register
  - input myinput;
  - reg myinput;
  - Incompatible declaration

Procedural Assignments

```
module assignment_test;
reg [3:0] a,b;
wire [4:0] sum1;
reg [4:0] sum2;

assign sum1 = a + b ;

initial
begin
     a=4'b1010;b=4'b0110;
     sum2 = a + b;
     $display("a b sum1 sum2);
     $monitor(a,b,sum1,sum2);
     #10   a=4'b0001;
end

endmodule
```

Continuous assignment

Procedural assignment

```
module FA(s,co,a,b,ci);
input a,b,ci;
output s,co;
reg s;

s=a^b^ci;

always @(a or b or ci)
begin
     assign co=(a&b)|(b&ci)|(a&ci);
end

endmodule
```

Error!   Illegal left-hand-side continuous assignment.

Error!   Illegal left-hand-side in assign statement.

Parameter

- parameter敘述可用於定義一個常數供模組用來定義輸出入埠的寬度、向量的大小等
  - parameter name = number;

```
module add8(a, b, c);
parameter width=8;
input  [width-1:0] a, b;
output [width-1:0] c;

assign c = a + b;

endmodule
```

Compiler Directives

- Verilog 語言跟C 語言一樣提供編譯命令 (compiler directives) 供使用者利用，指示編譯程式進行編譯的前置作業。
  - `define
  - `include

```
`define LEN  4
module buf4(in, out, clock);
input  [`LEN-1:0] in;
input  clock;
output [`LEN-1:0] out;
reg    [`LEN-1:0] out;
…
```

fa1.v

```
module fa1(a, b, ci, s, co);
input   a, b, ci;
output  s, co;
assign {co, s} = a + b + ci;
endmodule
```

fa2.v

```
`include “fa1.v”
module fa2(a, b, ci, s, co);
input   [1:0] a, b;
input   ci;
output  [1:0] s;
output  co;
wire    c1;
fa1 fa0(a[0],b[0],ci,s[0],c1);
fa1 fa1(a[1],b[1],c1,s[1],co);
endmodule
```

  - `define 用來定義模組中使用到的常數。
  - `include 用來引入其他verilog檔案一起編譯與合成。
  - 如果已經在proj list已經都加入fa1.v fa2.v，就沒有必要使用 `include fa1.v

Module Connections

- Module connection can be called by **port order** or **port names.**
- **call-by-name** can ignore the port order defined in the module.

```
module ha(a,b,sum,co)
input a,b;
output sum,co;
wire sum,co;
assign sum = a^b;
assign co=a&b;
endmodule
```

call by port order

```
module adder(a_in,b_in,sum_out,carry_cout)
input a_in,b_in;
output sum_out,co_out;
wire sum_out,co_out;
ha  adder0(a_in,b_in,sum_out,carry_out);
endmodule
```

call by port names

```
module adder(a_in,b_in,sum_out,carry_cout)
input a_in,b_in;
output sum_out,co_out;
wire sum_out,co_out;
ha  adder0(.b(b_in),
           .a(a_in),
           .sum(sum_out),
           .co(carry_out));
endmodule
```

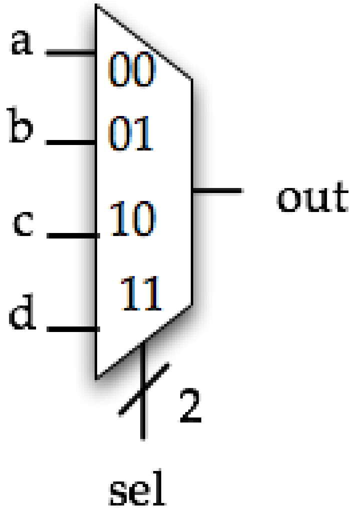
Outline

- Introduction
- Verilog Modules
- Verilog HDL Coding
- Examples of Combinational Circuits

MUX 1

```
module mux(
  out,  // output
  a,  // input a
  b,  // input b
  c,   // input c
  d,  // input d
  sel // selection control signal
);

output out;  // output
input a;  // input a
input b;  // input b
input c;  // input c
input d;  // input d
input [1:0] sel;  // selection control signal
reg out; // output (in always block)

always @(sel or a or b or c or d)
  if (sel==2'b00)  out = a;
  else if (sel==2'b01) out = b;
  else if (sel==2'b10) out=c;
  else out=d;

endmodule
```

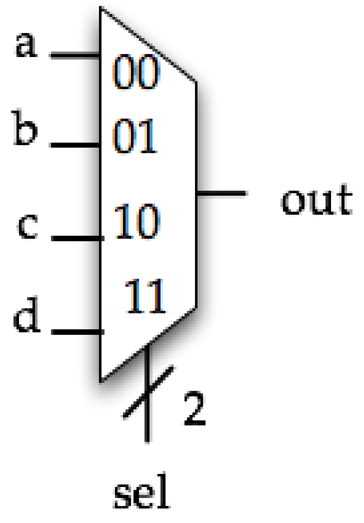
Final "else ---;" is required



MUX 2

```
module mux(
  out,  // output
  a,  // input a
  b,  // input b
  c,   // input c
  d,  // input d
  sel // selection control signal
);

output out;  // output
input a;  // input a
input b;  // input b
input c;  // input c
input d;  // input d
input [1:0] sel;  // selection control signal
reg out; // output (in always block)

always @(sel or a or b or c or d)
  case (sel)
    2'b00: out = a;
    2'b01: out = b;
    2'b10: out = c;
    2'b11: out = d;
    default: out = 0;
endcase
endmodule
```

Final "default : ---;" is required



Design Procedure

1 • From the ***specifications***, determine the inputs, outputs, and their symbols.

2 • Derive the ***truth table*** (***functions***) from the relationship between the inputs and outputs

3 • Derive the ***simplified Boolean functions*** for each output function.

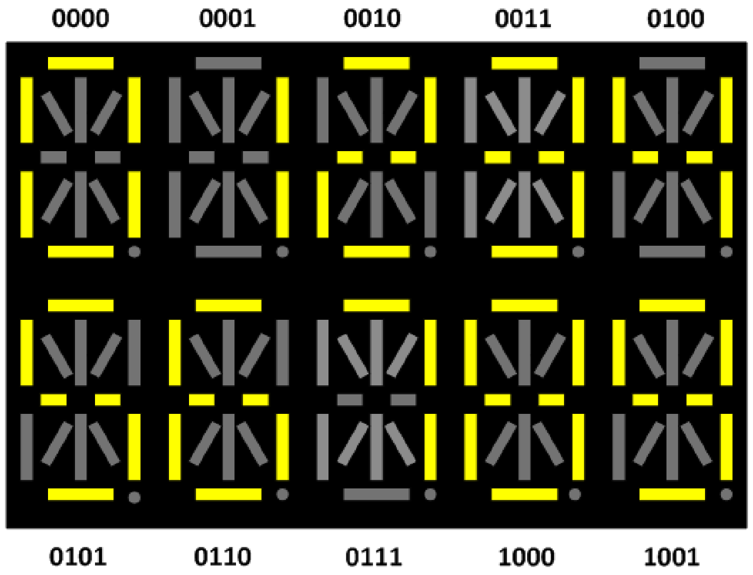
4 • Draw the logic diagram.

5 • Construct the Verilog code according to the logic diagram.

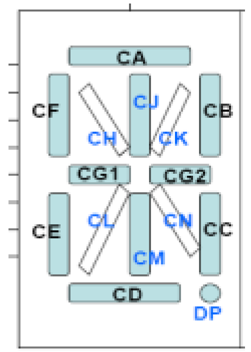
6 • Write the testbench and verify the design.

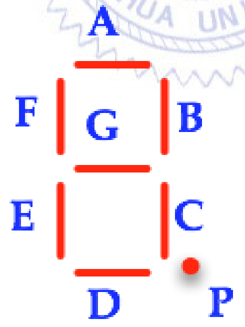
Seven-Segment Display Decoder (1/2)

1 input: bcd[3:0]
output: display[7:0]



2

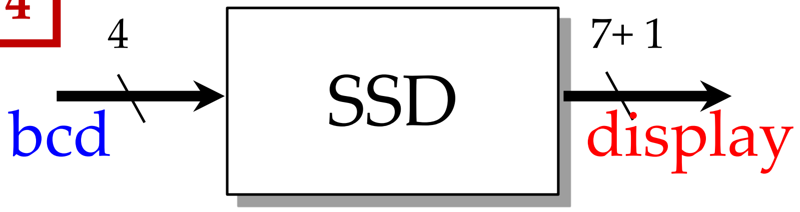




3

BCD	-> 7-segment_star_dot
0000	-> 0000001_1
0001	-> 1001111_1
0010	-> 0010010_1
0011	-> 0000110_1
0100	-> 1001100_1
0101	-> 0100100_1
0110	-> 0100000_1
0111	-> 0001111_1
1000	-> 0000000_1
1001	-> 0000100_1
others	-> 0111000_1 (F)

4



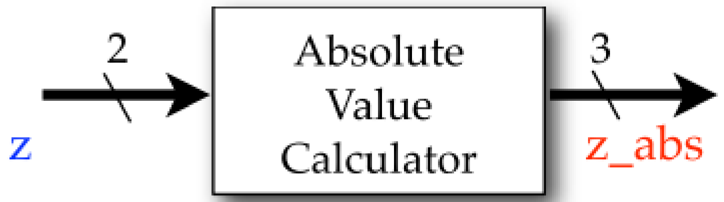
Seven-Segment Display Decoder (2/2)

5

```
module ssd(
  display,  // SSD display output
  bcd  // BCD input
);

output [7:0] display;  // SSD display output
input [3:0] bcd;  // BCD input

reg [7:0] display;  // SSD display output (in always)

// Combinational logics:
always @(bcd)
  case (bcd)
    4’d0: display = 8‘b 0000001_1; //0
    4’d1: display = 8‘b 1001111_1; //1
    4’d2: display = 8‘b 0010010_1; //2
    4’d3: display = 8‘b 0000110_1; //3
    4’d4: display = 8‘b 1001100_1; //4
    4’d5: display = 8‘b 0100100_1; //5
    4’d6: display = 8‘b 0100000_1; //6
    4’d7: display = 8‘b 0001111_1; //7
    4’d8: display = 8‘b 0000000_1; //8
    4’d9: display = 8‘b 0000100_1; //9
    default: display = 8‘b 0111000_1 ; //F
  endcase

endmodule
```

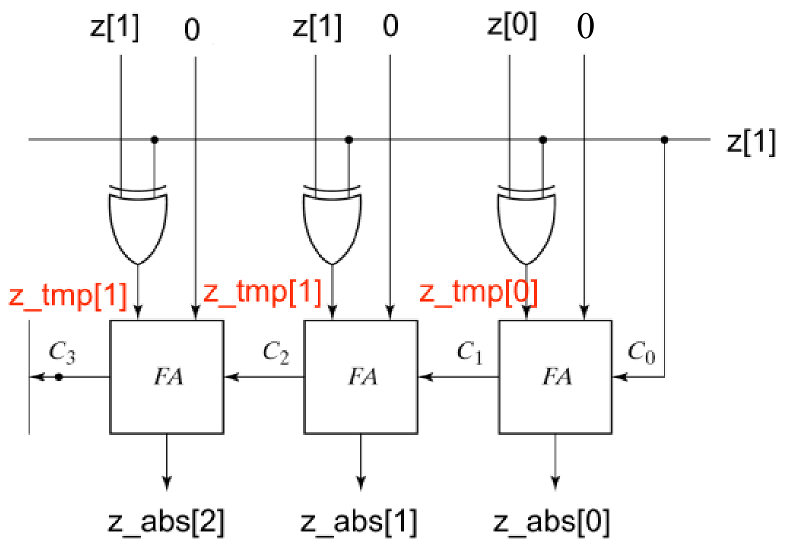
2-bit Absolute Value Calculator (1/3)

1 input: z[1:0]
output: z_abs[2:0]



2 If z is negative (MSB is 1), complement every bit and add 1.
If z is positive (MSB is 0), output remains the same as input.
Use XOR for MSB and every bit.

3



2-bit Absolute Value Calculator (2/3)

module : abs.v

5

```
module abs(
  z_abs,  // absolute value of z
  z  //  original value
);

output [2:0] z_abs;  // absolute value of z
input [1:0] z;  // original value

reg [1:0] z_tmp;  // XOR output
reg [2:0] z_abs;  // register for Z

// Combinational logics:
always @(z)
begin
  z_tmp[1]=z[1]^z[1];
  z_tmp[0]=z[0]^z[1];
  z_abs={z_tmp[1],z_tmp}+{2'b0,z[1]};
end

endmodule
```

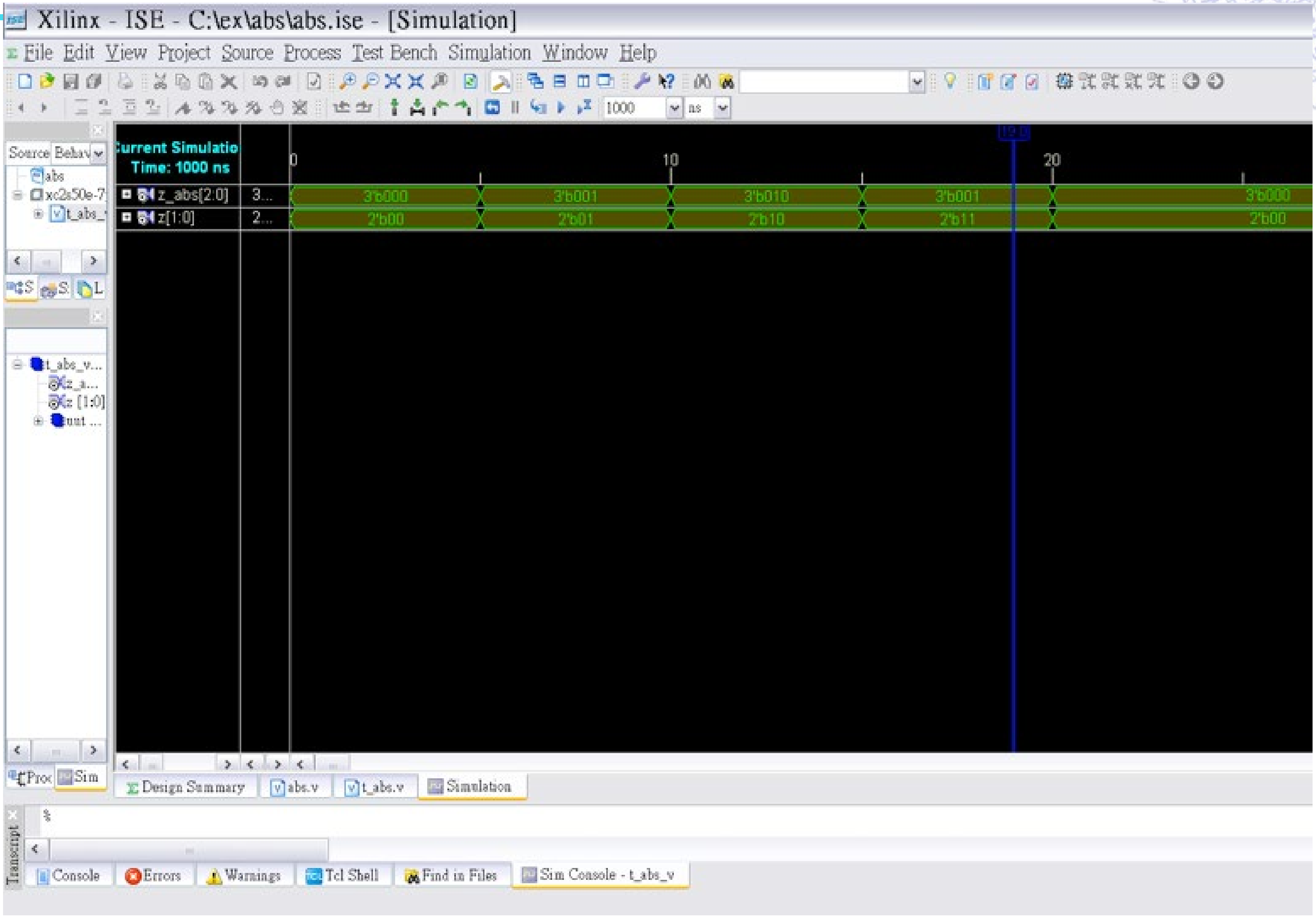
testbench module : t_abs.v

6

```
module t_abs;

wire [2:0] z_abs;  // absolute value of z
reg [1:0] z;  // original value

abs U0(.z_abs(z_abs),.z(z));

initial
begin
  z=2'b00;
  #5 z=2'b01;
  #5 z=2'b10;
  #5 z=2'b11;
  #5 z=2'b00;
end

endmodule
```

2-bit Absolute Value Calculator (3/3)

6


Xilinx - ISE - C:\ex\abs\abs.ise - [Simulation]
File Edit View Project Source Process Test Bench Simulation Window Help
Current Simulation Time: 1000 ns
z_abs[2:0]
z[1:0]
0
10
20
3'b000
3'b001
3'b010
3'b001
3'b000
2'b00
2'b01
2'b10
2'b11
2'b00
Design Summary
abs.v
t_abs.v
Simulation
Console
Errors
Warnings
Tcl Shell
Find in Files
Sim Console - t_abs_v