



Digital Circuits and Systems

Lecture 2 Verilog and SystemVerilog

Tian Sheuan Chang

Outline

- Introduction to hardware description language (HDL)
 - Why it is better than schematic entry
- Verilog語法 1 (Functional viewpoints)
 - Modules and ports
 - Basic syntax rules
 - Structural Verilog
 - How to model combinational logic
 - How to model sequential logic
- Verilog語法 2 (Language viewpoints)
 - One language, many coding styles
 - Continuous v.s. procedural assignments
 - Blocking vs NonBlocking
 - For loop and parameterized design
- SystemVerilog
- Gotchas 正確的使用方法 <= (會踩到的陷阱)

Learning Plan for SystemVerilog



- Familiarize with the **basics of digital design**
 - Logic design
- Learn the **basic syntax and concepts** of SystemVerilog
 - Logic vs HDL
- **Practice** writing simple SystemVerilog code
 - Lab.
- Learn more advanced concepts in SystemVerilog
 - such as object-oriented programming, interfaces, and constraints.
 - Not covered in this course, learn them in IC Lab
- Work on **projects** that involve using SystemVerilog
 - Homeworks and final projects
 - Architecture design and optimization
 - E.g. pipeline, microcode, parallelism

Learning Verilog

```
counter;
```

```
unter;
```

```
_overflow = (counter ==
```

```
_underflow = (counter ==
```

```
edge clk or negedge xres
```

```
xreset == 0)
```

```
    counter <= 0;
```

```
if (en && in && !counte
```


```
    counter <= counter +
```

```
if (en && !in && !counte
```

```
    counter <= counter -
```

```
    counter <= counter;
```

What is Verilog?

- **Verilog** is a hardware description language (HDL) 
- Used to model digital systems and design integrated circuits
- Syntax is similar to the C programming language




Benefits of Learning Verilog

- **Design** digital systems and integrated circuits
- **Simulate** and test designs
- **Synthesize** designs into hardware



Steps to Learn Verilog

- **Understand** the basics of digital logic
- **Learn** the Verilog syntax and language 
- **Practice** writing Verilog code

如何學好Verilog IC設計

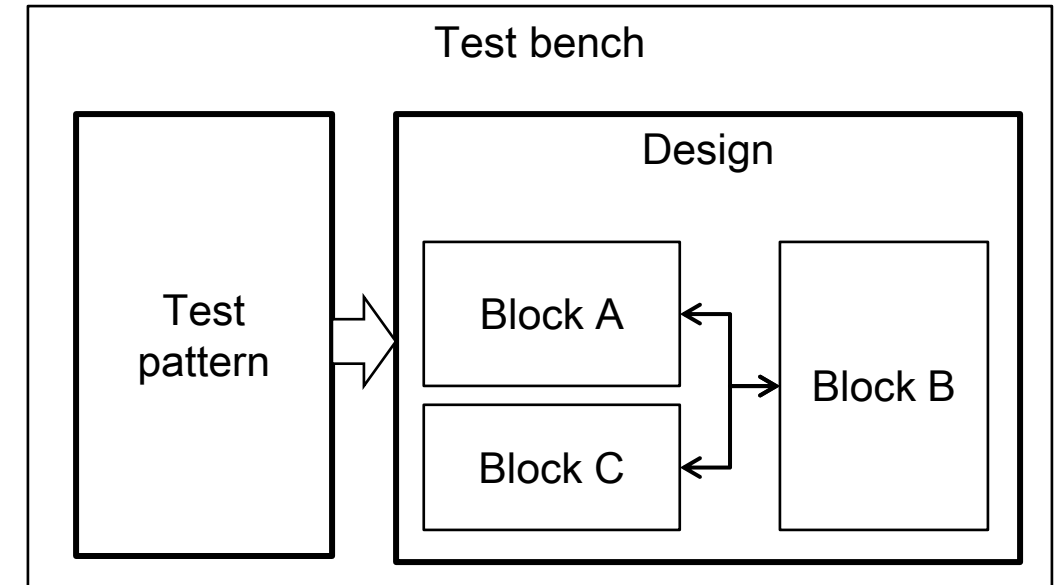
- 心態要正確: 先有硬體架構(aka.邏輯設計), 再找合適的Verilog語法實現
- 課程重點: 從架構設計到Verilog 實現
- Verilog 實現
 - Verilog語法(基礎)=> 正確的使用方法(進階)<= (會踩到的陷阱)(中階)
 - Combinational logic/sequential logic
- 架構設計
 - 基礎:搬出你學過的邏輯設計/VLSI/計算機組織
 - 中階:數位電路
 - 進階:論文
- 設計最佳化: 如何讓你的設計變得更快/更省面積/更省電

Outline

- Introduction to hardware description language (HDL)
 - Why it is better than schematic entry
- Verilog語法 1
 - Modules and ports
 - Basic syntax rules
 - Structural Verilog
 - How to model combinational logic
 - How to model sequential logic
- Verilog語法2
 - Functional Verilog
 - One language, many coding styles
 - Continuous v.s. procedural assignments
 - Blocking vs NonBlocking
 - For loop and parameterized design
- SystemVerilog
- Gotchas 正確的使用方法 <= (會踩到的陷阱)

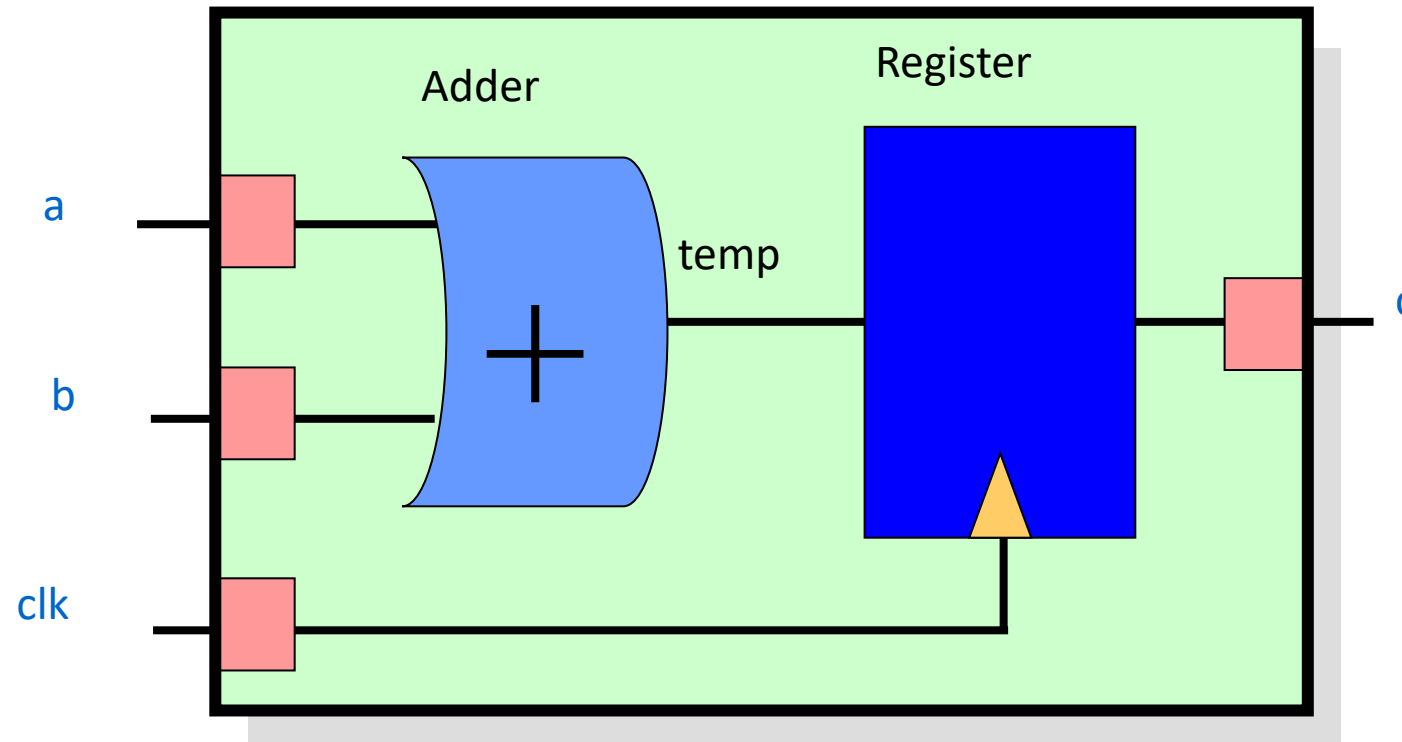
Coded components

- Design
 - Consists of blocks (modules)
 - Synthesizable Verilog
- Test pattern
 - Stimulus for verification
 - Behavior-level Verilog
- Test bench
 - instantiates the test pattern and design.



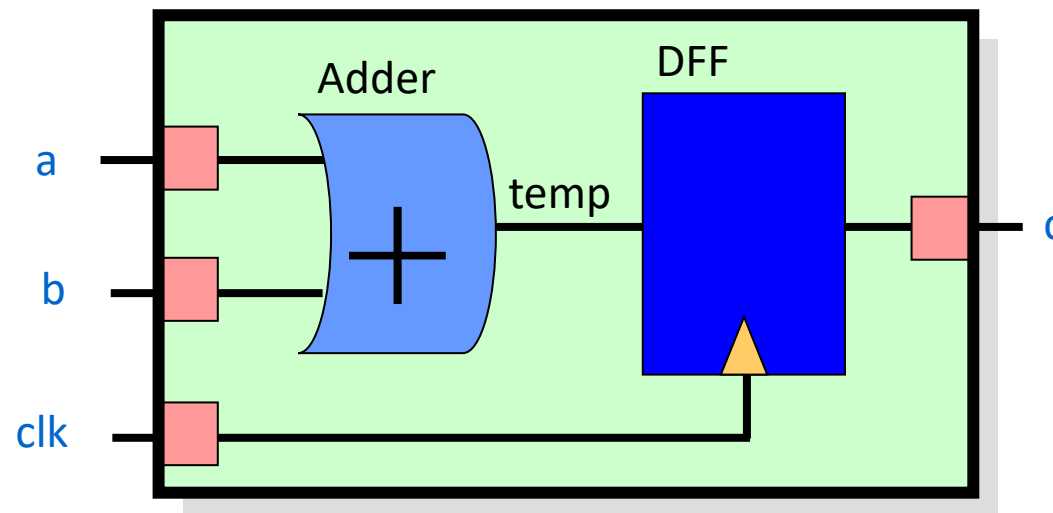
Example – Adder with Register (1/2)

- $c = a + b$



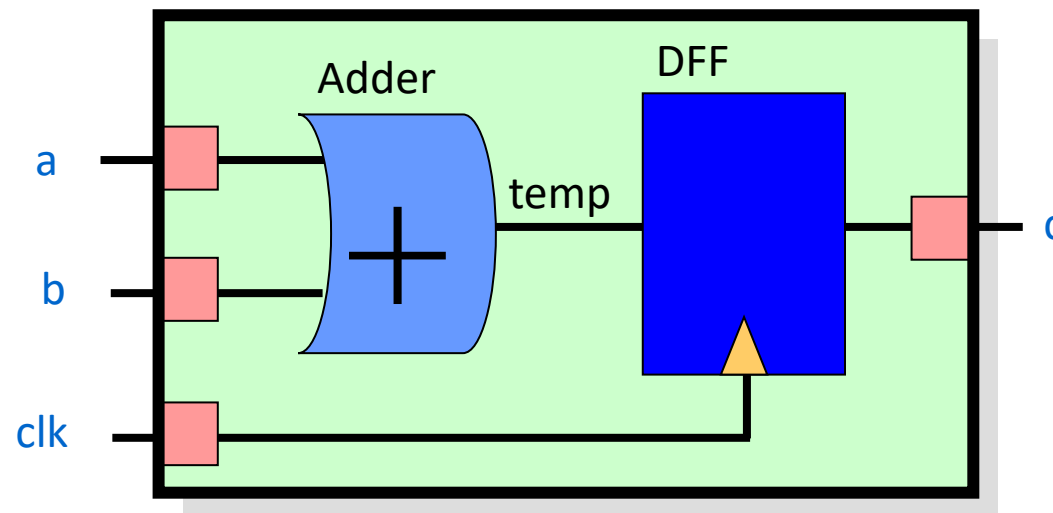
Example – Adder with Register (2/2)

```
module adder_reg (a, b, clk, c);  
    // input port  
    input [7:0] a;  
    input [7:0] b;  
    input      clk;  
    // output port  
    output [8:0] c;  
  
    // Internal signal  
    wire [8:0] temp;  
    wire [8:0] c;  
  
    // Adder process  
    adder u_adder (a, b, temp);  
    DFF u_dff(temp, clk, c);  
endmodule
```



Example – Adder with Register (2/2)

```
module adder_reg (a, b, clk, c);  
    // input port  
    input [7:0] a;  
    input [7:0] b;  
    input      clk;  
    // output port  
    output [8:0] c;  
  
    // Internal signal  
    wire [8:0] temp;  
    reg [8:0] c;  
  
    // Adder process  
    assign temp = a + b;  
    always @(posedge clk)  
        c <= temp;  
endmodule
```



Example – Adder with Register (2/2)

```
module adder_reg (input [7:0] a, b,  
                  input clk, output reg [8:0] c);
```

```
    // Internal signal
```

```
    wire [8:0] temp;
```

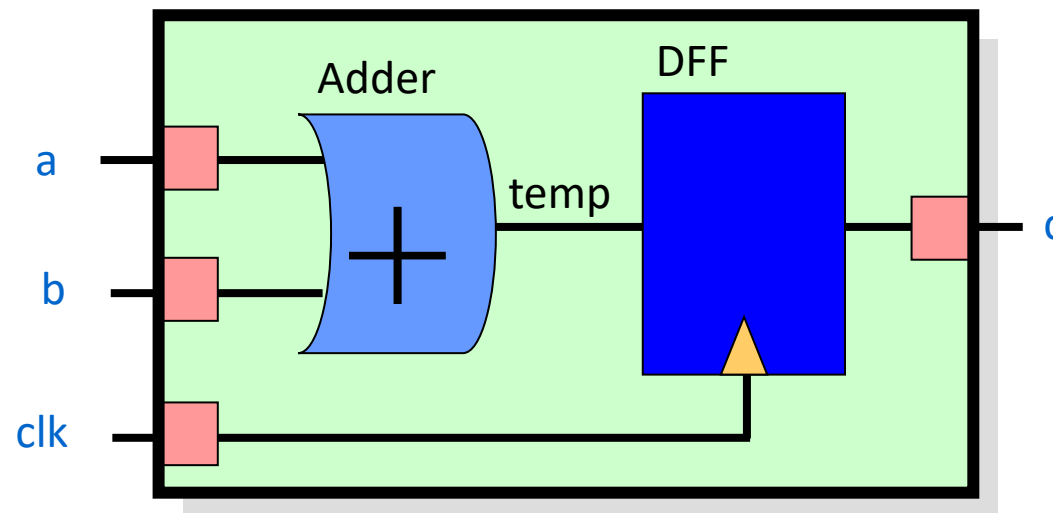
```
    // Adder process
```

```
    assign temp = a + b;
```

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

```
        c <= temp;
```

```
endmodule
```



INTRODUCTION OF HARDWARE DESCRIPTION LANGUAGE

學習重點

- 了解HDL的好處
- 如何選擇Verilog 語法
 - 寫法很多種，哪一個比較好？
 - Verilog語法的差別

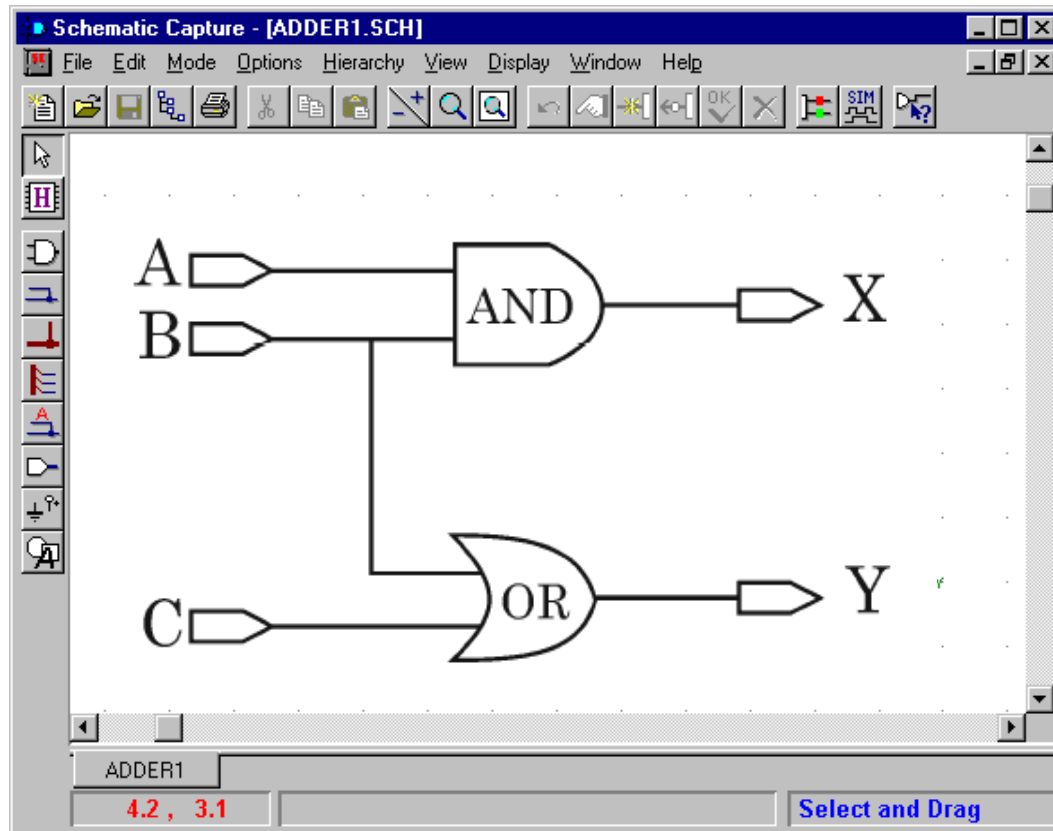
What Is a Hardware Description Language?

A Hardware Description Language (**HDL**) is a high-level programming language with special constructs used to model the function of hardware logic circuits.

- The special language constructs provide you the ability to:
 - Describe the **functionality** of a circuit
 - Describe the **connectivity** of the circuit, like in schematic
 - Express **concurrency** (hardware execution is parallel)
 - Describe a circuit at various **levels of abstraction** (from structure to behavior)
 - Describe the **timing** of a circuit

```
// Adder process  
assign temp = a + b;  
always @(posedge clk)  
    c <= temp;
```

Why HDL? From Schematic Entry to HDL

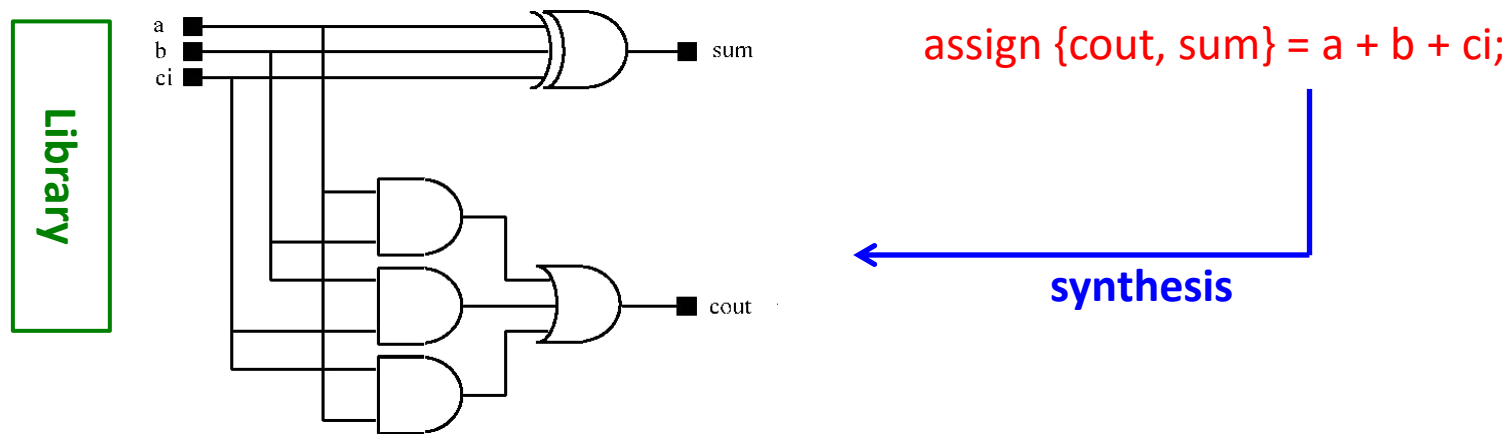


```
module AndOr (output X, Y, input A, B, C);  
    //  
    assign #10 X = A & B;  
    assign #10 Y = B | C;  
    //  
endmodule // AndOr.
```

- Hardware description languages (HDLs)
 - Not programming languages
 - Parallel languages tailored to digital design
 - Synthesize code to produce a circuit

Why HDL? Benefits of HDL for Design

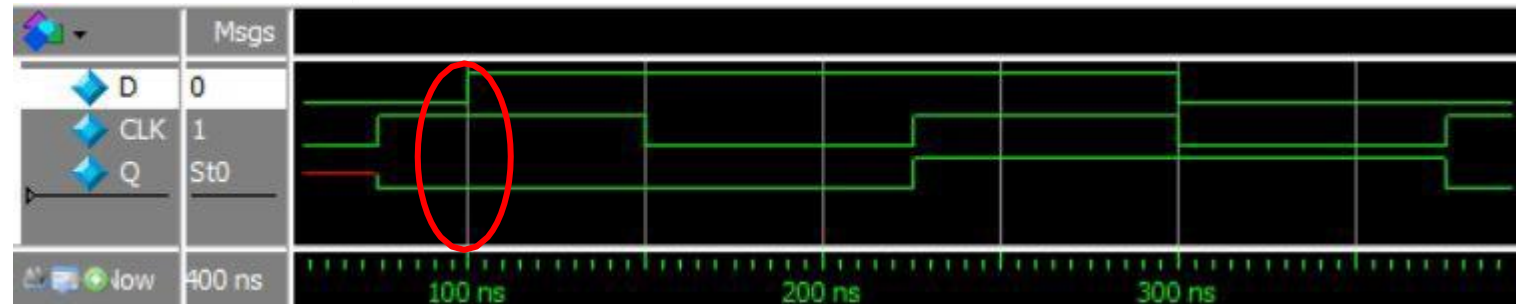
- **Reuse** same HDL code for different technologies
 - **Implementation independent**
 - Synthesis tool helps you decide the best implementation
 - Explore different architecture easier, e.g. fast adder or small area adder
 - 40nm, 22nm, 7nm
 - ASIC process, FPGA



Why HDL? Benefits of HDL for Simulation

- Easy and lower cost to find bugs earlier than on chip debugging
 - With assertions, waveform
- Faster than gate level

```
module latch_example(D, CLK, Q);  
    input D;  
    input CLK;  
    output reg Q;  
  
    always @(CLK) begin  
        if (CLK) begin  
            Q <= D;  
        end  
    end  
endmodule
```



HDL History

Two leading HDLs:

- Verilog/SystemVerilog

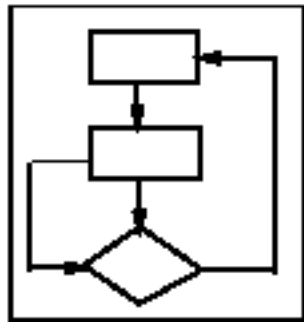
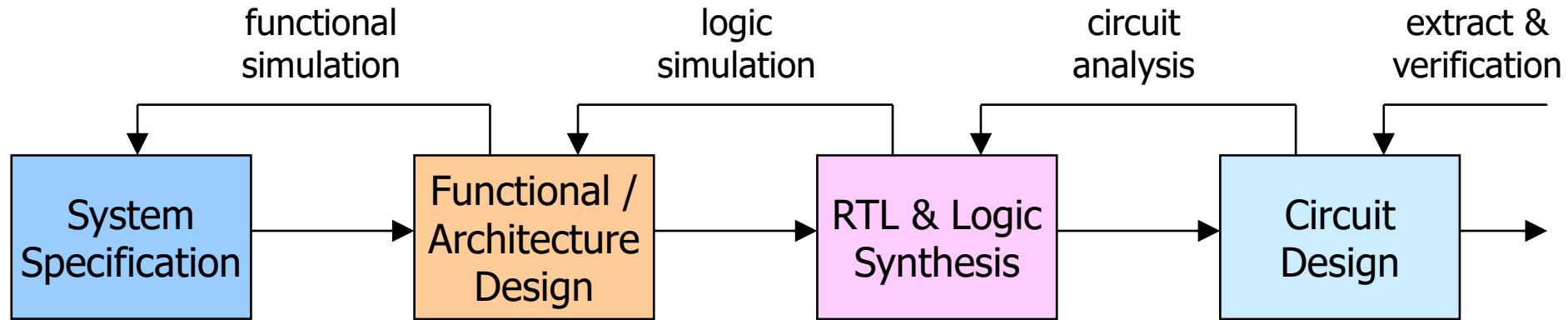


- Verilog developed in 1984 by Gateway Design Automation for logic simulation
- IEEE standard (1364) in 1995, 2001, for logic synthesis as well, similar to C
- SystemVerilog in 2005 (IEEE STD 1800-2009), similar to C++ and VHDL
 - First unified hardware description and verification language
 - Simple and easier to write and debug
 - Superset of Verilog
- We will talk Verilog first and then SystemVerilog

- VHDL 2008

- Developed in 1981 by the Department of Defense, IEEE standard (1076) in 1987
- Updated in 2008 (IEEE STD 1076-2008)

Typical HDL-Based Design Flow

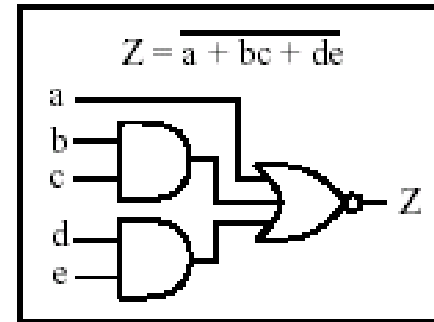


behavior
representation

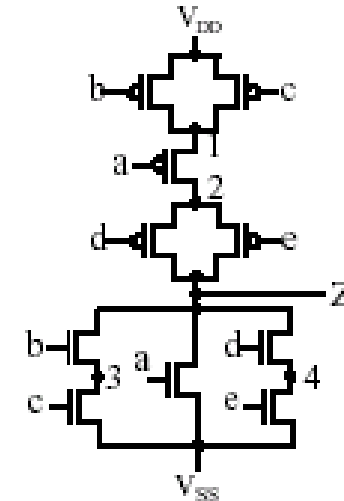
```

module fz(a,b,c,d,e,Z);
input a,b,c,d;
output Z;
assign Z =
  ~(a|(b&c)|(d&e));
endmodule
  
```

HDL description



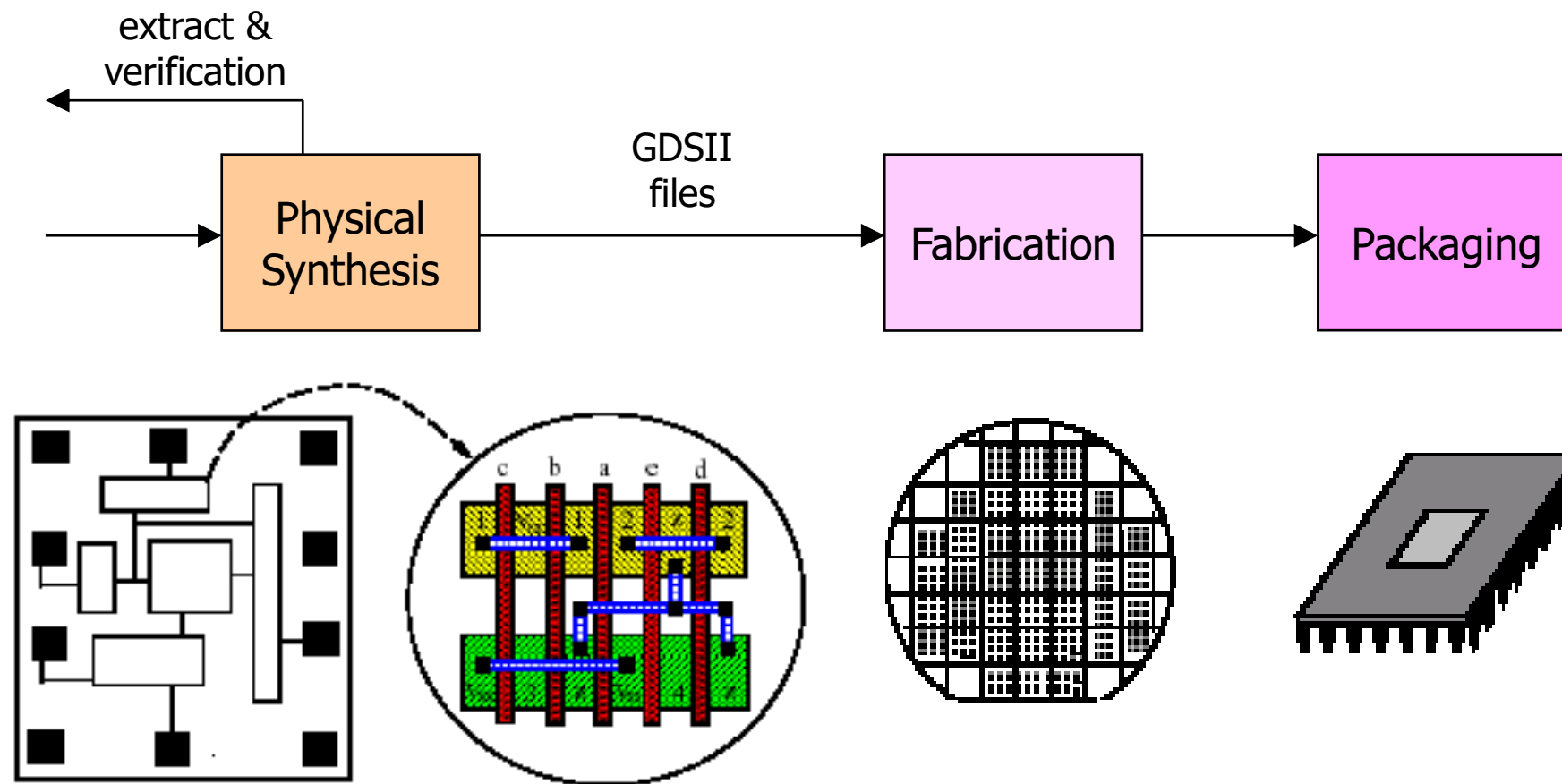
gate-level
representation



switch-level
representation

課程重點

Typical Cell-Based Design Flow



Standard cell (事先做好的gate，用兜積木方式組成邏輯)

Verilog-Supported Levels of Abstraction

- Behavioral
 - Describes a system by the flow of data between its functional blocks
 - Defines signal values when they change
- Functional or Dataflow or Register Transfer Level (RTL)
 - Describes a system by the flow of data and control signals between and within its functional blocks
 - Defines signal values with respect to a clock
- Structural (Gate)
 - Describes a system by connecting predefined components
 - Uses technology-specific, low-level components when mapping from an RTL description to a gate-level netlist, such as during synthesis.

Gate (Structural)

- The structural level in Verilog is appropriate for small components such as ASIC and FPGA cells.
 - Verilog has built-in primitives, such as the and gate, that describe basic logic functions. (用內建的邏輯閘(and, or ...)描述電路)
 - You can describe your own User Defined Primitives (UDPs).
 - The resulting **netlist from synthesis is often purely structural**, but you can also use structural modeling for glue logic.
- The function of the following structural model is represented in Verilog primitives, or gates. It also contains propagation delays:

```

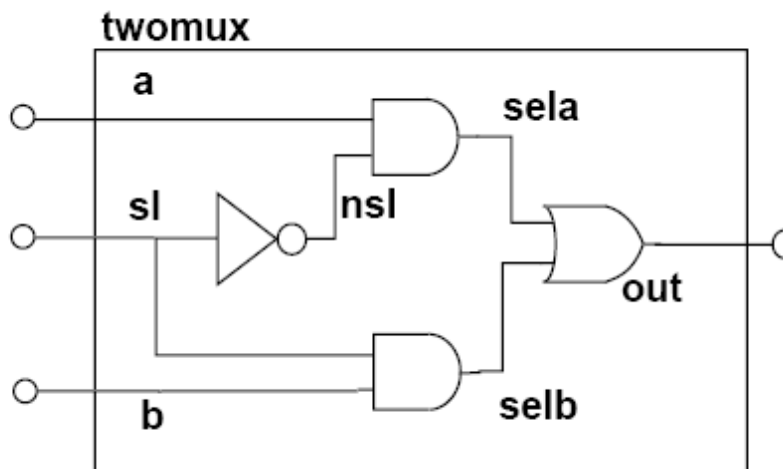
module twomux (out, a,b,s1);

input a,b,s1;
output out;

  not u1 (ns1, s1 );
  and #1 u2 (sela, a, ns1);
  and #1 u3 (selb, b, s1);
  or #2 u4 (out, sela, selb);

endmodule

```



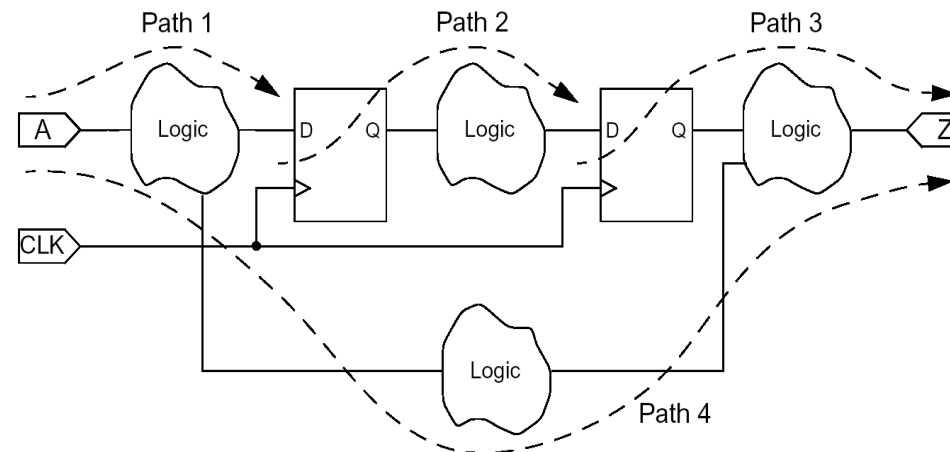
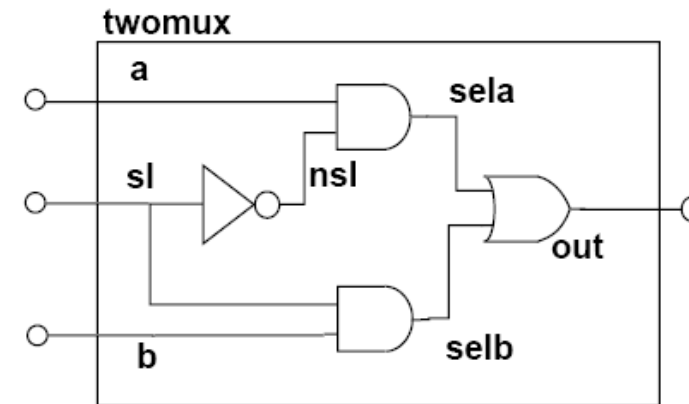
Dataflow (RTL)

- Describe the algorithm in terms of logical data flow. (描述電路中暫存器值資料流動方式)

```

module mux2_1 (out, a, b, sel);
output out;
input a, b, sel;
    assign out = (a&~sel) | (b&sel);
    // or assign out = (sel == 0) ? a : b;
endmodule

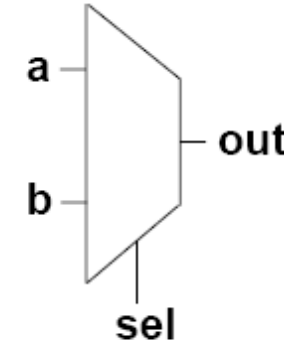
```



Behavioral

- Describe the algorithm without concern for the actual logic required to implement it.
- The behavior of the following MUX is : always, at any change in signals **a** or **b** or **sl**, if **sl** is 0 pass the value of **a** to **out**, else pass the value of **b** to **out**.

```
module muxtwo (out, a, b, sl);  
  input a,b,sl;  
  output out; reg out;  
  always @(sl or a or b)  
    if (!sl)  
      out = a;  
    else  
      out = b;  
endmodule
```



- In a behavioral model, the function of the logic is modeled using high level language constructs, such as **@**, **while**, **wait**, **if** and **case**.
- Test benches, or test fixtures, are typically modeled at the behavioral level. All behavioral constructs are legal for test benches.

DataFlow / Behavioral / Structural

Data Flow Modeling

```
module DF_AND (in1 , in2 , Out)
```

```
input in1, in2 ;
```

```
output Out ;
```

```
wrie Out ;
```

```
assign Out = in1 & in2 ;
```

```
and a1(Out, in1, in2);
```

```
endmodule
```

Gate/structural Modeling

Behavioral Modeling

```
module Beh_AND (in1 , in2 , Out)
```

```
input in1 , in2 ;
```

```
output Out ;
```

```
reg Out ;
```

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

```
begin
```

```
    Out = in1 & in2
```

```
end
```

```
endmodule
```

Mixed Styles Modeling

```
module FA_MIX(A, B, CIN, SUM, COUT);  
    input A, B, CIN;  
    output SUM, COUT;  
    reg COUT;  
    reg T1, T2, T3;
```

```
    wire S1;
```

```
    xor X1(S1, A, B); // gate instantiation
```

Structural

```
    always @ (A or B or CIN) // Always block  
    begin  
        T1 = A & CIN;  
        T2 = B & CIN;  
        T3 = A & B;  
        COUT = (T1 | T2 | T3);  
    end
```

Behavioral

```
    assign SUM = S1 ^ CIN; // Continuous assignment
```

Data Flow

```
endmodule
```

Choosing the right style...

- Structural Verilog
 - Use for **hierarchy** (instantiating other modules)
 - **Floorplanning tools** often require that modules which include structural verilog not include other styles..
- Dataflow Verilog: `assign target = expression`
 - Use for (most) **combinational logic**
 - Avoids problems with sensitivity list omissions
- Behavioral Verilog: `always @ (...) begin ... end`
 - Use to model **state elements** (e.g., registers)
 - Sometimes useful for **combinational logic** expressed using for or case statements
 - Simulates much faster than dataflow statements since no waveforms are produced for signals internal to behavioral block. Here's where you can make the tradeoff between simulation speed and debugability.

```
// Adder process
adder u_adder (a, b, temp);
DFF u_dff(temp, clk, c);
```

```
// Adder process
assign temp = a + b;
always @(posedge clk)
  c <= temp;
```

These two styles are often mixed in a single module

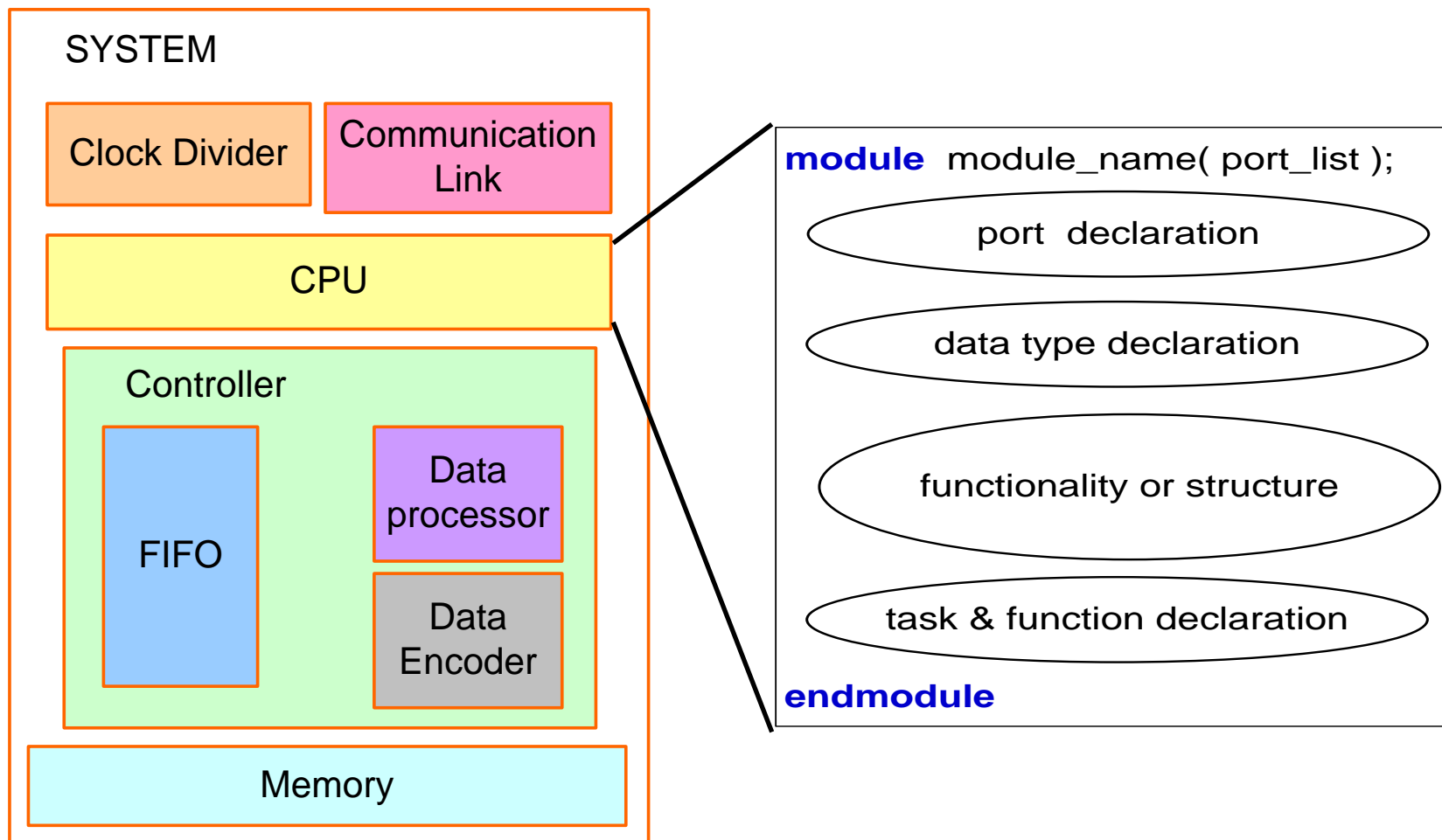
Outline

- Introduction to hardware description language (HDL)
 - Why it is better than schematic entry
- Verilog語法 1 (Functional viewpoints)
 - Modules and ports
 - Basic syntax rules
 - Structural Verilog
 - How to model combinational logic
 - How to model sequential logic
- Verilog語法 2 (Language viewpoints)
 - One language, many coding styles
 - Continuous v.s. procedural assignments
 - Blocking vs NonBlocking
 - For loop and parameterized design
- SystemVerilog
- Gotchas 正確的使用方法 <= (會踩到的陷阱)

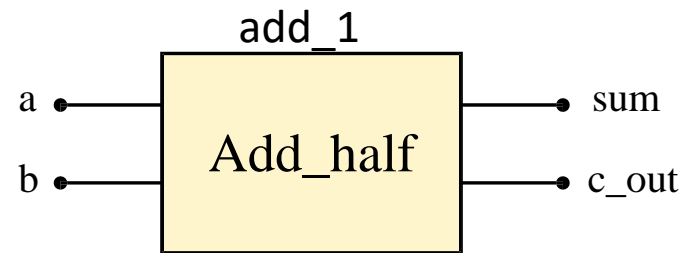
VERILOG MODULES AND PORTS

Verilog Module

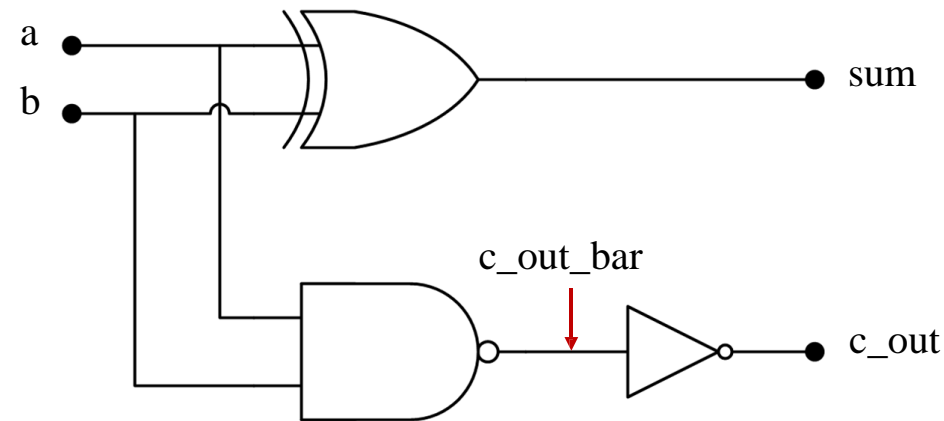
- 模組(module)是組成一個電路的基本單位



Block Diagram & Schematic



block diagram

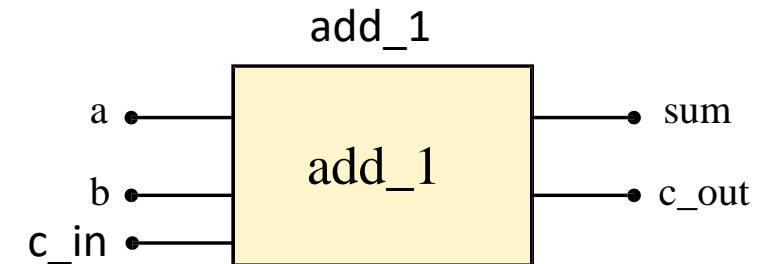


schematic

Modules

- A module is the basic building block in Verilog
 - Every module starts with the keyword **module**, has a name, and ends with the keyword **endmodule**
- A module can be
 - A design block
 - A simulation block, i.e., a testbench

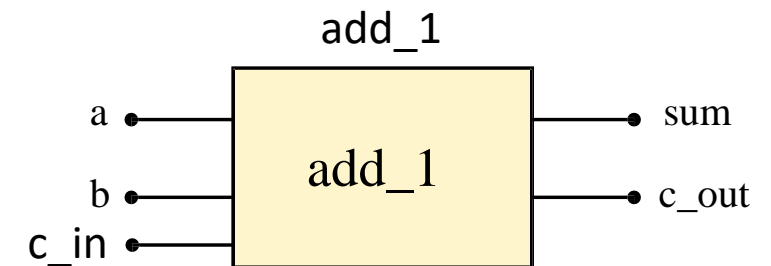
```
module add_1 (. . .);  
  . . .  
  . . .  
endmodule
```



Module Ports

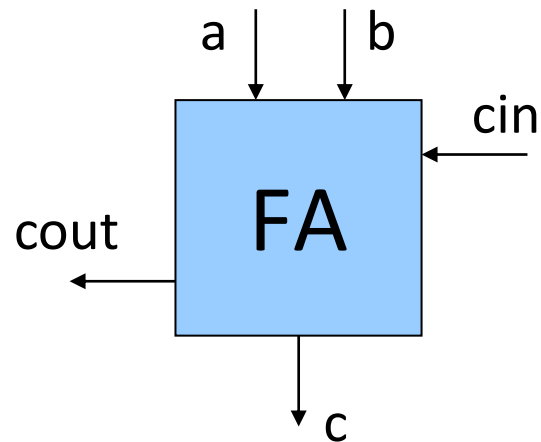
- Module ports describe the input and output terminals of a module
 - Listed in parentheses after the module name
 - Declared to be **input**, **output**, or **inout** (bi-directional)
 - **//** single line comment, **/* */** multi-line comments, **white space** is ignored

```
module add_1 (sum, a, b, c_in, c_out);  
  output sum;  
  input a;  
  input b;  
  input c_in;  
  output c_out;  
  
  // output sum, c_out;  
  /* input a, b, c_in;  
  */  
  . . .  
  . . .  
endmodule
```



Verilog Modules

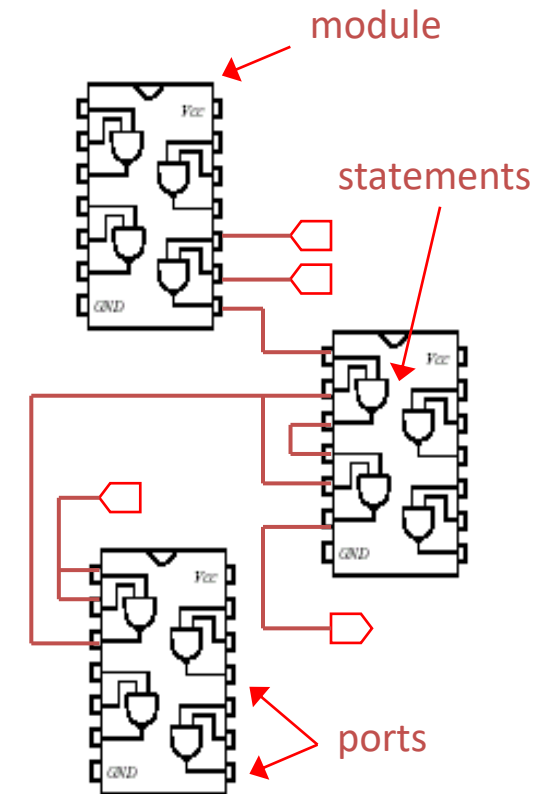
- Likes an IC or ASIC cell, logic block or a system



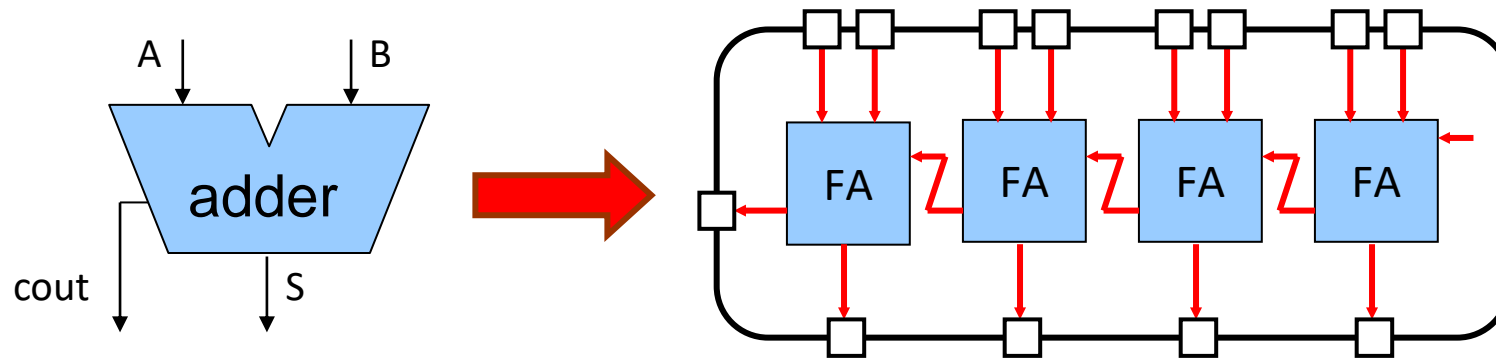
```
module FA( input  a, b, cin
           output cout, sum );
```

```
// HDL modeling of 1 bit
// adder functionality
```

```
endmodule
```



A module can **instantiate** other modules creating a **module hierarchy**



```
module adder( input  [3:0] A, B,
               output    cout,
               output [3:0] S );
```

```
module FA(
```

module name →

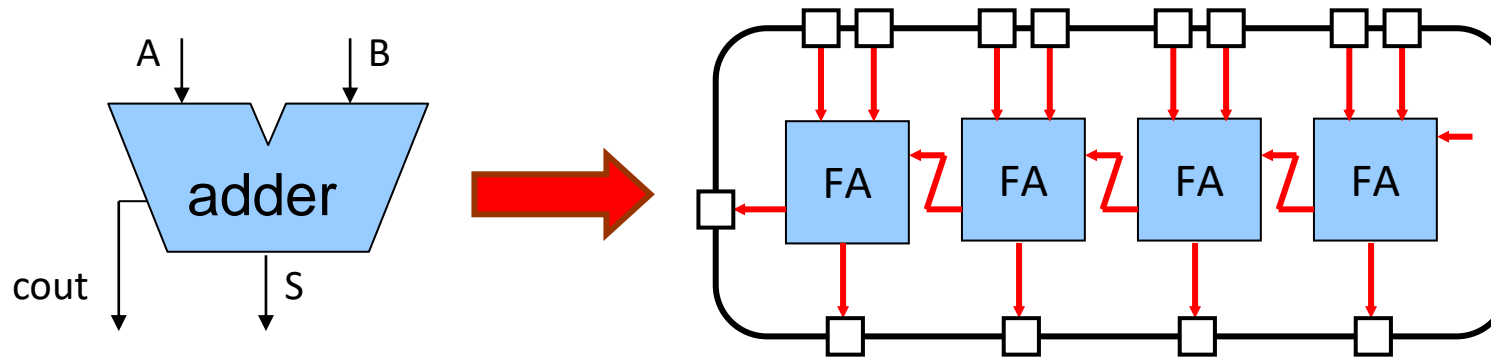
```
    wire c0, c1, c2;
    FA fa0( ... );
    FA fa1( ... );
    FA fa2( ... );
    FA fa3( ... );
```

Instance name →

```
endmodule
```

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

A module can **instantiate** other modules creating a **module hierarchy**



```
module adder( input  [3:0] A, B,
               output    cout,
               output [3:0] S );
```

```
  wire c0, c1, c2;
```

```
  FA fa0( A[0], B[0], 1'b0, c0, S[0] );
```

```
  FA fa1( A[1], B[1], c0, c1, S[1] );
```

```
  FA fa2( A[2], B[2], c1, c2, S[2] );
```

```
  FA fa3( A[3], B[3], c2, cout, S[3] );
```

```
endmodule
```

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

Carry Chain

Verilog supports connecting ports by position and by name

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

Connecting ports **by ordered list (Positional mapping: buggy code)**

```
FA fa0( A[0], B[0], 1'b0, c0, S[0] );
FA fa0( A[0], B[0], 1'b0, S[0], c0 ); //buggy code, wrong connection, hard to find
```

Connecting ports **by name (compact) (Name mapping: better)**

```
FA fa0( .a(A[0]), .b(B[0]),
        .cin(1'b0), .cout(c0), .sum(S[0]) );
```

Connecting ports **by name**

```
FA fa0
(
    .a      (A[0]),
    .cin    (1'b0), //no bug
    .b      (B[0]), //no bug
    .cout   (c0),
    .sum     (S[0])
```

For all but the smallest modules, connecting ports by name yields clearer and less buggy code.

Feel tedious to write such code?
Use SystemVerilog style .*

Connecting Ports by Order/Name List

```
module top (Out1, Out2, In1, In2);
  output Out1, Out2;
  input  In1, In2;
```

Order: `comp c1 (Out1, Out2, In1, In2);`

Name: `comp c1`
`(.i2(In2), .o1(Out1), .o2(Out2), .i1(In1));`

`endmodule`

```
module comp (o1, o2, i1, i2);
  output o1, o2;
  input  i1, i2;
  ..
  ..
  ..
endmodule
```

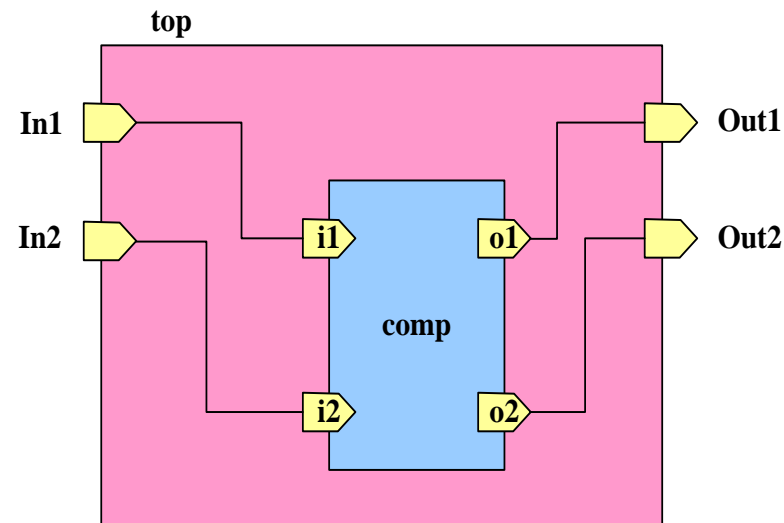
//One port left unconnected

```
module top (Out1, In1, In2);
  output Out1;
  input  In1, In2;
```

`comp c1 (Out1, , In1, In2);`

`comp c1 (.i2(In2), .o1(Out1), .i1(In1));`

`endmodule`



Combined Port/Type Declarations

```
module adder(sum, co, a, b, ci);  
    output [31:0] sum;  
    output      co;  
    input  [31:0] a, b;  
    input      ci;  
  
    reg      [31:0] sum;  
    reg      co;  
    wire     [31:0] a, b;  
    wire     ci;
```

Verilog-1995

```
module adder(sum, co, a, b, ci);  
    output reg [31:0] sum;  
    output reg      co;  
    input  wire [31:0] a, b;  
    input  wire      ci;
```

Verilog-2001

預設不宣告型態為wire

ANSI-C Style Port List

```
module adder(  
    output [31:0] sum,  
    output        co,  
    input  [31:0] a, b,  
    input        ci  
);  
  
    reg    [31:0] sum;  
    reg          co;  
    wire   [31:0] a, b;  
    wire          ci;
```

Verilog-2001

```
module adder(  
    output reg  [31:0] sum,  
    output reg          co,  
    input  wire [31:0] a, b,  
    input  wire          ci  
);
```

Verilog-2001

Can apply to **task** and **function** declarations

Module Port Parameter List

```
module adder(sum, co, a, b, ci);
    parameter MSB = 31,
               LSB = 0;

    output [MSB:LSB] sum;
    output          co;
    input  [MSB:LSB] a, b;
    input          ci;

    reg      [MSB:LSB] sum;
    reg      co;
    wire     [MSB:LSB] a, b;
    wire     ci;
```

Verilog-1995

```
module adder
    #(parameter MSB = 31, LSB = 0)
    ( output reg  [MSB:LSB] sum,
      output reg                co,
      input  wire [MSB:LSB] a, b,
      input  wire                ci
    );
```

Verilog-2001

adder #(16, 0) u_adder (...); //parameter values: 16, 0, will override the default ones
 //this is a positional mapping for parameters

BASIC SYNTAX RULE

Outline

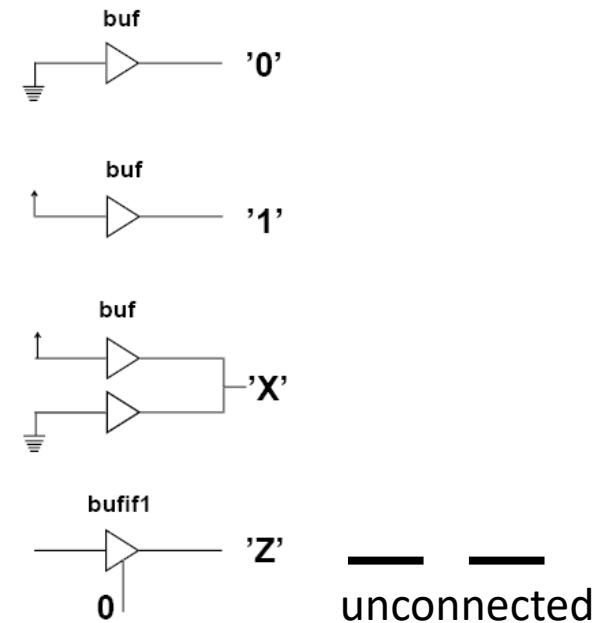
- 4 value logic system in Verilog
- Integer and Real Constants
- Identifiers
 - Special language tokens \$
 - Compiler directives
- Data types
- Verilog expressions and operators

4-Value Logic System in Verilog

Verilog data type is a bit-vector where bits can take on one of four values

Value	Meaning
0	Logic zero (false)
1	Logic one (True)
X	Unknown logic value or conflict
Z	High impedance , floating (unconnected input are set to z)

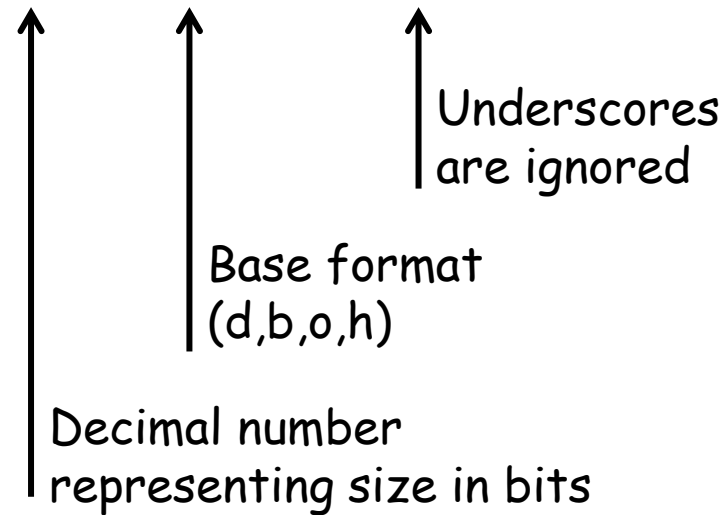
Hardware meaning



- The “unknown” logic value in Verilog is not the same as “don’t care”. It represents a situation where the value of a node cannot be predicted. In real hardware, this node will most be at either 1 or 0.
- An X bit might be a 0, 1, Z, or in transition. We can set bits to be X in situations where we don’t care what the value is. This can help catch bugs and improve synthesis quality.

Constants: Verilog includes ways to specify **Number** in various bases

4' b10_11



- Binary literals
 - 8' b0000_0000
 - 8' b0xx0_1xx1
- Hexadecimal literals
 - 32' h0a34_def1
 - 16' haxxx
- Decimal literals
 - 32' d42

**We'll learn how to
actually assign literals to
nets a little later**

Constants: Number Representation

- Integer
 - Syntax: `<size>'<base format><value>`
 - Decimal: `8'd79`
 - Binary: `8'b0100_1111` ← Underscore _ is ignored, for easy to read
 - Octal: `8'o117`
 - Hexadecimal: `8'h4f = 8'h4F`
 - Negative value stored by 2'd complement
 - Bit value
 - 1, 0: logic high and low
 - Z: high impedance
 - X: unknown
 - ?: don't care
 - Default size is 32-bits decimal number
 - When no base is specified

Constants: More examples

- Example

```
4'b0           // 4-bit 0000
4'b1001        // 4-bit 1001
8'b1111_1111   // 8-bit 1111 1111, '_' underscore can be use for readability
16'hfe12       // 16-bit 1111 1110 0001 0010
9'o457         // 9-bit 100 101 111
8'd220         // 8-bit decimal 220
-8'd12         // 8-bit decimal -12
8'd-12         // illegal
12'h12x        // 12-bit 0001 0010 xxxx
12'h12z        // 12-bit 0001 0010 zzzz
```

Constants: More examples

- Example

//Unsigned numbers (at least 32 bit)

```
123           // default: decimal radix, 32-bit width
`d123         // decimal 123
`h7B          // hexadecimal 7B
`hxx          // xxxx xxxx
```

//Sized numbers

```
16'd5         // 16-bit constant `b0000_0000_0000_0101
11'h1X?       // 11-bit constant `b001_XXXX_ZZZZ
```

//signed numbers

```
8'shFF        // 8-bit signed constant, 2's complement -1
-8'd12        // -12, 將負號放在<size>之前。
```

To be absolutely clear in your intent it's usually best to explicitly **specify the width and base**

Constants: Truncation and Extension

<size>'<base><value>

- Rule if <size> and <value> does not match
- **Smaller:** <size> < <value>
 - left-most bits of <value> are **truncated (MSB)**
 - 6'hca: 6-bit, store as 6'b00_1010 (truncated, not 1100_1010)
- **Larger:** <size> > <value>
 - If MSB is 0, X or Z number is extended to **fill MSBs with 0, X, Z** respectively
 - If MSB is 1 number is extend to **fill MSBs with 0/1**, depending on the **sign**
 - 6'ha: 6-bit, store as 6'b00_1010 (a => 1010, filled with 2-bit '0' on left)
 - 3'b1=-3'b01=3'b111, (negative number, extended with 1)
 - 3'b1 = 3'b001, (positive number, extended with 0)

Integer and Real Constants: *Real*

In Verilog, constant values (literals) can be integers or reals.

- Real numbers can be represented in decimal or scientific format
 - Real numbers in scientific notation are represented as
 $\text{<mantissa><e or E><exp>}$
 which is interpreted as $\text{<mantissa> * 10 <exp>}$
 - Real numbers expressed with a decimal point must have at least one digit on each side of the decimal point (Ex: 0.12 (可) .12 (不可))
- Example:

– 12	unsized decimal (zero-extended to 32 bits)
– 1.2E12	$= 1.2 \times 10^{12}$
– 0.1	decimal notation
– 236.123_76e-12	underscores are ignored $= 236.12376 \times 10^{-12}$
– 32e-4	scientific notation for 0.0032 $= 32 \times 10^{-4}$
– 4.1E3	scientific notation for 4100 $= 4.1 \times 10^3$
– .12	invalid real number
– .3E3	invalid real number
– .2e-7	invalid real number

IDENTIFIERS

Lexical Convention: Identifier

- Verilog is a **case sensitive** language - **keywords are lower case**
- Terminate lines with **semicolon ;**
- Identifiers:

- Starts only with a letter or an **_**(underscore), can be any sequence of letters (a-z, A-Z), digits (0-9), \$, _
- Case-sensitive **abc != Abc**

- Example

- shiftreg_a, _bus3, n\$657

和C一樣，大小寫有差
只能字母_開頭
後接字母數字\$_，

```
module MUX2_1 (out, a, b, sel);  
  output out;  
  input a, b, sel;  
  not not1 (sel_, sel);  
  and and1 (a1, a, sel_);  
  and and2 (b1, b, sel);  
  or or1 (out, a1, b1);  
endmodule
```

Verilog Identifiers

Reserved keywords

and	always	assign	attribute	begin	buf	bufif0	bufif1
case	cmos	deassign	default	defparam	disable	else	endattribute
end	endcase	endfunction	endprimitive	endmodule	endtable	endtask	event
for	force	forever	fork	function	highz0	highz1	if
initial	inout	input	integer	join	large	medium	module
nand	negedge	nor	not	notif0	notif1	nmos	or
output	parameter	pmos	posedge	primitive	pulldown	pullup	pull0
pull1	rcmos	reg	release	repeat	rnmos	rpmos	rtran
rtranif0	rtranif1	scalared	small	specify	specparam	strong0	strong1
supply0	supply1	table	task	tran	tranif0	tranif1	time
tri	triand	trior	triereg	tri0	tri1	vectored	wait
wand	weak0	weak1	while	wire	wor		

Find the bug

- **Illegal identifier**
 - 12_reg, \$abc, a*b_net, n@238
- Answer: Error part
 - 12_reg, \$abc, a*b_net, n@238
 - illegal, cannot start with digits, or \$
 - Cannot contain special character like *, @

Coding Style: Naming Conventions

- Consistent naming convention for the design
- Lowercase letters for signal names: **valid**
- Uppercase letters for constants: **MAX**
- *clk* sub-string for clocks: **in_clk**
- *rst* sub-string for resets: **g_rst**
- Suffix
 - *_n* for active-low, *_z* for tri-state, *_a* for async , ...: **rst_n**, **out_z**
- *[name]_cs* for current state, *[name]_ns* for next state
- Identical(similar) names for connected signals and ports
- Consistency within group, division and corporation

命名建議規則
讓三個月後的自己也可看的懂

Special Language Tokens: \$

For System Tasks and Functions

\$<identifier>

- The '\$' sign denotes Verilog system **tasks** and **functions**.
 - A number of system tasks and functions are available to perform different operations, such as:
 - Finding the current simulation time (**\$time**)
 - Displaying/monitoring the values of the signal (**\$display**, **\$monitor**)
 - Stopping the simulation (**\$stop**)
 - Finishing the simulation (**\$finish**)
- \$monitor** (**\$time**, "a = %b, b = %h", a, b);

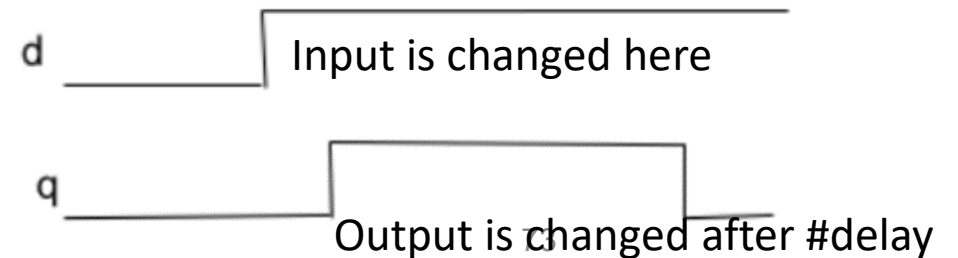
Special Language Tokens:

Delay Specification

The pound sign (#) character denotes the **delay** specification for procedural statements, and for gate instances but not module instances.

- The # delay for gate instances is referred to by many names, including gate delay, propagation delay, intrinsic delay, and intra-object delay.

```
module MUX2_1 (out, a, b, sel);
  output out;
  input  a, b, sel;
  not #1 not1(sel,sel);
  and #2 and1(a1,a,sel_);
  and #2 and2(b1,b,sel);
  or  #1 or1(out,a1,b1);
endmodule
```



Compiler directives

- The directives start with a grave accent (`) followed by some keyword

``define`

Text-macro substitution

``ifdef, `ifndef, `else,
`endif`

Conditional compilation

``include`

File inclusion

```
`include "file1.v"  
// Used as `WORD_SIZE in code  
`define WORD_SIZE 32  
  
module test ();  
`ifdef TEST  
// A implementation  
`else  
// B implementation  
`endif  
assign out = `WORD_SIZE{1'b1};  
endmodule
```

Summary of basic lexical conventions in Verilog

- Comments
 - Single line: `//`
 - Multi line: `/* */`
- Case sensitive
 - Verilog keywords: lower case
 - Others: lower case != upper case 和C一樣，大小寫有差
 - `AaBb != aabb`
- 建議
 - 變數都用小寫
 - 常數都用大寫
 - `clk` sub-string for clocks
 - `rst` sub-string for resets
 - `_n` for active-low

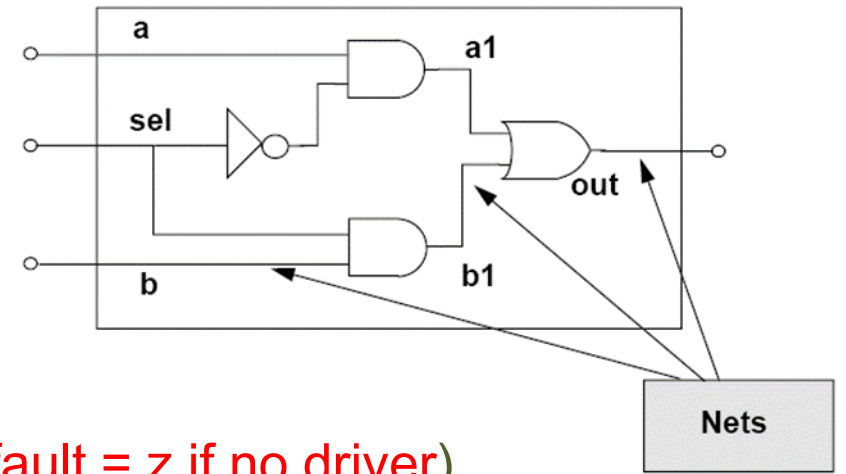
VERILOG DATA TYPES AND LOGIC SYSTEM: NETS/REGISTERS/LOGIC

Outline

- wire vs. reg
- Vector and array in Verilog

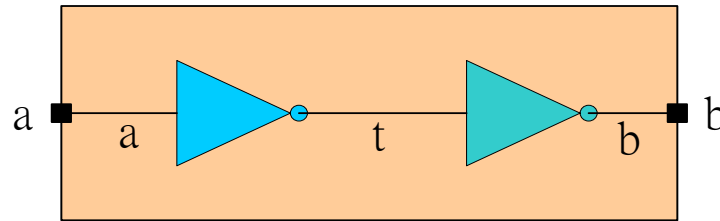
Major Data Types

- Two major data type classed in Verilog
 - Nets “**wire**”: structural connectivity
 - Represent physical connection between devices (**default = z if no driver**)
 - Default values if not declared
 - Registers “**reg**”: abstraction of storage (**default = x**)
 - (may or may not be physical storage)
 - Both nets and registers are informally called signals, and may be either scalar or vector
- SystemVerilog
 - Use **logic** to replace both



Explicitly vs. Implicitly

- **Explicitly** declared with a declaration in your Verilog code.
- **Implicitly** declared with no declaration when used to connect structural building blocks in your code. Implicit declaration is always a net type "wire" and is one bit wide.



Explicit declarations :

```
module buf2 (a, b);
  output b;
  input a;

  wire a, b, t;

  not (t, a);
  not (b, t);
endmodule
```

明确宣告

Implicit declarations :

```
module buf2 (a, b);
  output b;
  input a;

  not (t, a);
  not (b, t);

endmodule
```

隐含 a, b, t 是 wire

reg

- In Verilog, the term register (reg) simply means a variable that can hold a value.
 - ◆ reg表示資料儲存，除非給定新的數值，否則數值會一直維持。
- They are used only in functions and procedural blocks (behavioral modeling).
- The default initialization value for a register data type shall be the unknown value x.

Note: 別把Verilog的reg與硬體電路中用信號緣觸發之正反器DFF(暫存器)互相混淆。就是一個可以保持(hold)數值的變數。

VLSI Signal Processing Lab.

- logic



Data type (Optional)

- Net

- **Structural connection** between components

wire , tri	Interconnecting wire - no special resolution function
wor, trior	Wired outputs OR together (models ECL)
wand, triand	Wired outputs AND together (models open-collector)
tri0, tri1	Net pulls-down or pulls-up when not driven
supply0, supply1	Net has a constant logic 0 or logic 1 (supply strength)
triereg	Retains last value, when driven by z (tristate).

- Register

- **Variable** to store data

reg	Unsigned variable
integer	Signed variable - 32 bits
time	Unsigned integer - 64 bits
real	Double precision floating point variable
string	e.g. "internal error", \n, \t

Verilog vectors

Known as BUS in hardware

- Declare by a range following the type

`<data type> [left range : right range] <Variable name>`

- Single element that is n-bits wide

`reg [0:7] a, b; //Two 8-bit reg with MSB as the 0th bit`

`wire [3:0] data; //4-bit wide wire MSB as the 4th bit`



- Vector part select (access)

`a[5] // bit # 5 of vector a`

`data[2:0] // Three LSB of vector Data`

Bit extraction and combination

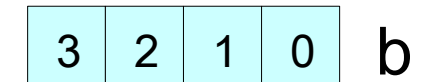
- Bit-selecting

$a = b[0];$



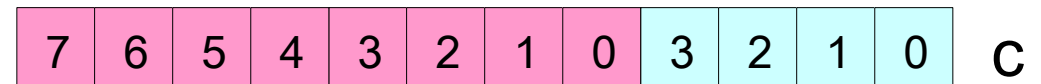
- Part-selecting

$a = b[3:0];$



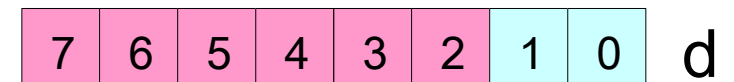
$c = b[7:4];$

- Concatenation

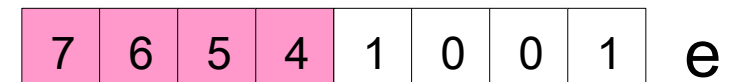


$c = \{ a, b \};$

$d = \{ a[7:2], b[1:0] \};$



$e = \{ a[7:4], 4'b1001 \};$



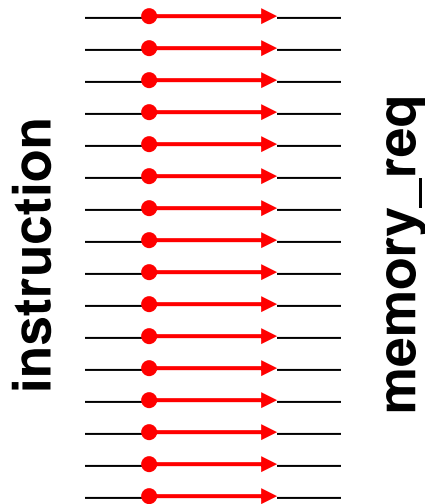
$f = \{ 2\{a[7:4] \} \};$



The Verilog keyword **wire** is used to denote a standard hardware net

```
wire [15:0] instruction;  
wire [15:0] memory_req;  
wire [ 7:0] small_net;
```

```
assign memory_req = instruction;
```



**Absolutely no type safety
when connecting nets!**

```
assign small_net = instruction;
```



Only the lowest instruction[7:0] is assigned.

Verilog arrays

- Array: range follows the name

```
<datatype> <array name> [<array indices>]  
reg B [15:0]; // array of 16 reg elements
```

- Array of vectors

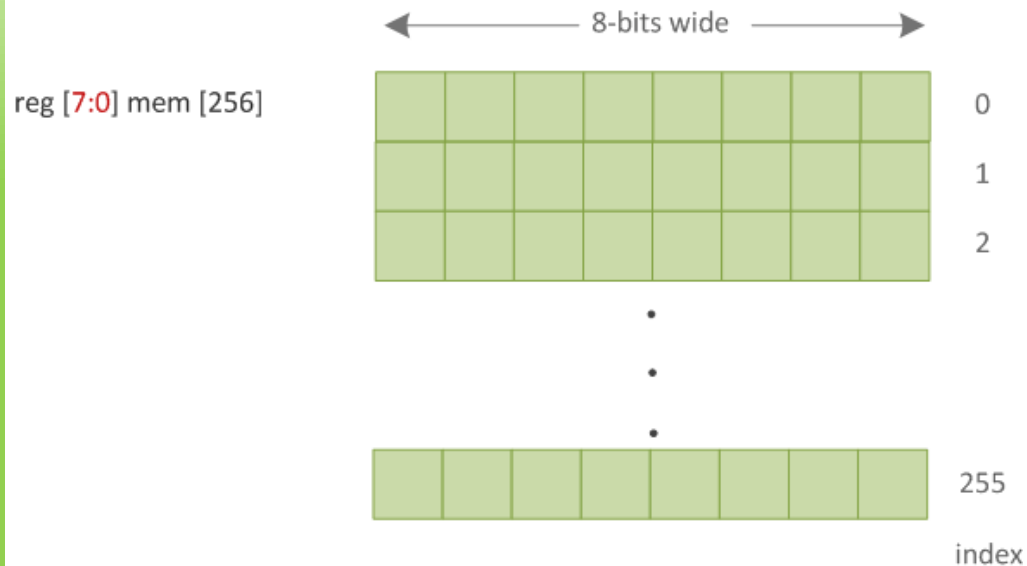
```
<data type> [<vector indices>]<array name> [<array indices>]  
reg [15:0] C [1023:0]; // vector with 1024 elements, each 16b
```

- Memory access

```
<var name> [<array indices>] [<vector indices>]
```

Data storage and Verilog arrays

- Simple RAM model



```

module RAM (output [7:0] Obus,
             input  [7:0] Ibus,
             input  [3:0] Adr,
             input          Clk, Read
             );
    reg [7:0] mem[255:0];
    reg [7:0] ObusReg;

    assign Obus = ObusReg;

    always @(posedge Clk)
        if (Read==1'b0) mem[Adr]      <= Ibus;
        else              ObusReg <= mem[Adr];

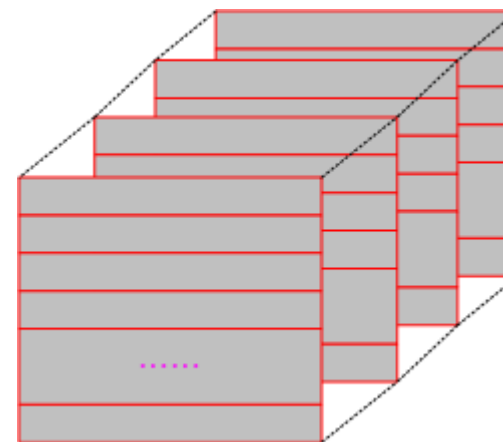
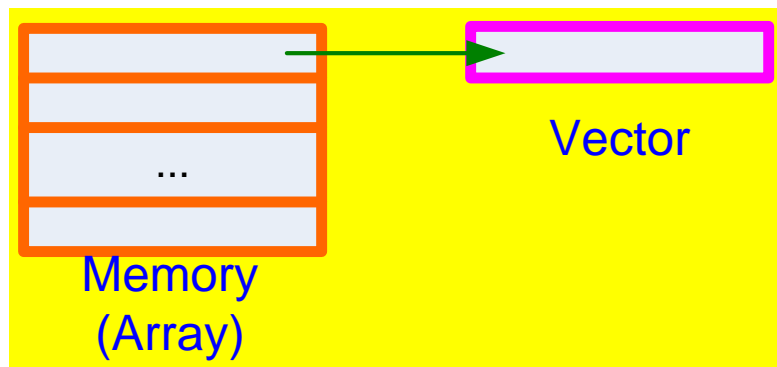
endmodule

```

More Than 2-D Arrays

- Multi-dimensional array support in Verilog-2001

```
reg [7:0] ram [0:127] [0:3] ; // 4 Bank RAM  
$display("ram[124][3] =%3h", ram[124][3]); //ram[124][3] = 05a
```



Memory bank

VERILOG EXPRESSIONS AND OPERATORS

Operators Precedence

`a < b-1 && c != d || e == f`
`(a < b-1) && (c != d) || (e == f)`

worse
better

Operator		Property
Arithmetic	<code>+, -, *</code> <code>/, %</code>	
Relational	<code><, >, <=, >=</code>	
Equality	<code>==, !=</code> <code>==, !=</code>	Including x, z comparison, for test bench only
Logic	<code>!, &&, </code>	True or false (1 or 0) (1bit)
Bitwise	<code>~, &, , ^, ~^, ^~</code>	Multi-bit (>=1 bit)
Reduction	<code>&, ~&, , ~ , ^, ~^, ^~</code>	Multiple bit to 1 bit
Shift	<code><<, >></code> <code><<<, >>></code>	Can be replaced by connection With signed extension, not used
Replication	<code>{ n { m } }</code>	Duplicate m by n times
Conditional	<code>A ? B : C;</code>	

Boolean operators

- **Bitwise operators** perform bit-oriented operations on vectors
 $\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = 4'b1010$
 $4'b0101 \& 4'b0011 = \{0\&0, 1\&0, 0\&1, 1\&1\} = 4'b0001$
- **Reduction operators** act on each bit of a single input vector
 $\&(4'b0101) = 0 \& 1 \& 0 \& 1 = 1'b0$
- **Logical operators** return one-bit (**true/false**) results
 $!(4'b0101) = 1'b0$

Bitwise

$\sim a$	NOT
$a \& b$	AND
$a b$	OR
$a \wedge b$	XOR
$a \sim \wedge b$	XNOR

Logical

$!a$	NOT
$a \&\& b$	AND
$a b$	OR

Note distinction between $\sim a$ and $!a$

Reduction

$\&a$	AND
$\sim \&$	NAND
$ $	OR
$\sim $	NOR
\wedge	XOR

Operator Examples

```

wire [3:0] A, X,Y,R,Z;
wire [7:0] P;
wire r, a, cout, cin;

assign R = X | (Y & ~Z);
assign r = &X;
assign R = (a == 1'b0) ? X : Y;
assign P = 8'hff;
assign P = X * Y;
assign P[7:0] = {4{X[3]}, X[3:0]};
assign {cout, R} = X + Y + cin;
assign Y = A << 2;
assign Y = {A[1], A[0], 1'b0, 1'b0};

```

use of bit-wise Boolean operators
 example reduction operator
 conditional operator
 example constants
 arithmetic operators (use with care!)
 (ex: sign-extension)
 bit field concatenation
 bit shift operator
 equivalent bit shift

Operator Examples

```

ans = 2**3           // ans = 8
numa = rega << 2 ;   // rega      =1 0 1 1
                    // numa      =1 0 1 1 0 0
numb = rega >> 3;     // rega      =1 0 1 1
                    // numb      = 0 0 0 0 0 1
numc = regb <<< 1;    // regb      =   1 1 0 1
                    // numc      = 1 1 1 0 1 0
numd = regb >>> 2;    // regb      =   1 1 0 1
                    // numd      = 1 1 1 1 1 1
nume = regb >> 2;     // regb      =   1 1 0 1
                    // nume      = 0 0 1 1 1 1
-7 % 2;              //evaluates to -1, takes sign of first operand
7 % -2;              //evaluates to 1, takes sign of first operand

```


Logic Values

- Logic with multilevel (0,1,X,Z) logic values
 - NAND anything with 0 is 1
 - NAND with X/Z get an X
- True tables define the how outputs are compute
 - Z is treated as X

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

	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

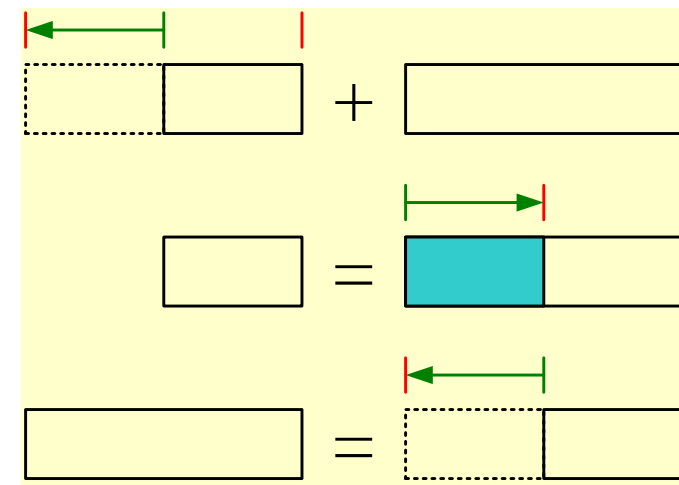
Sizing in Verilog

- Verilog automatically **resizes values** in an expression according to the sizes of variables in the expression.
- Verilog automatically **truncates or extends** the right-hand-side value in an assignment to fit the left-hand-side variable.

```

module sign_size;
reg [3:0] a, b;
reg [15:0] c;
initial
begin
  a = -1; // a is unsigned, so it stores "1111"
  b = 8;
  c = 8; // b = c = 1000
  #10 b = b + a; // result 10111 is truncated. b = 0111
  #10 c = c + a; // c = 00000000000010111
end
endmodule

```



Sizing in Verilog

- all operands in an expression are first expanded to the size of the largest vector in the statement (including both sides of an assignment statement).
 - Concatenate and replicate operations are evaluated before the expansion, and represent a new vector size.
 - Unsigned operands are expanded by left-extending with zero.
 - Signed operands are expanded by left-extending with the value of the most significant bit (the sign bit).
- 會把運算元先擴展到計算式中bit數最多的那個 (包含等號兩邊一起看)
 - Unsigned 補0
 - Signed 補sign bit

MODEL COMBINATIONAL LOGIC

Three Ways to Model Combinational Logic

- Structural Verilog
- `assign a =`
 - Continuous assignment
 - 一行寫完的logic
- `always@*`
 - Procedural assignment
 - 要多行寫完
 - `if else, case,`
 - `for loop`

assign

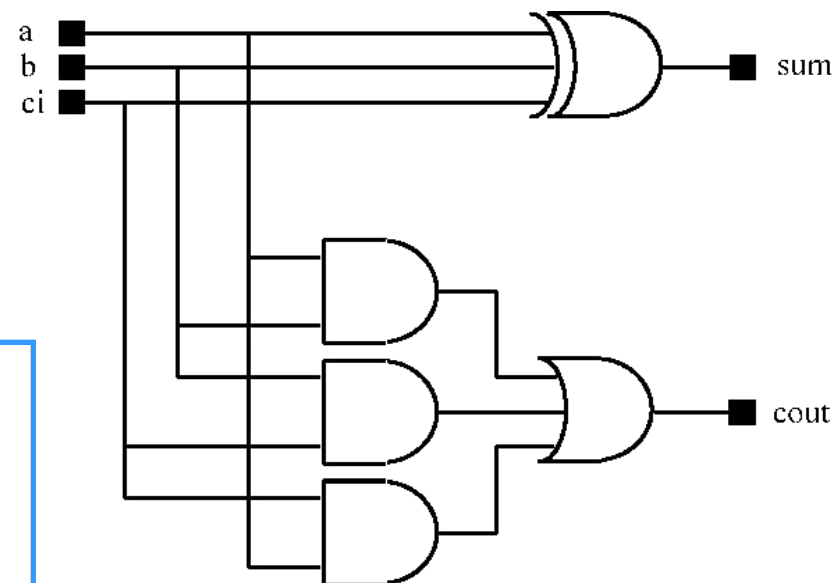
COMBINATIONAL LOGIC

使用CONTINUOUS ASSIGNMENT/ STRUCTURAL CONNECTION

A Full Adder Example

```
module fadder (sum,cout,a,b,ci);
  //port declaration
  output sum, cout;
  input  a, b, ci;
```

```
  //netlist declaration
  xor u0 (sum, a, b, ci);
  and u1 (net1, a, b);
  and u2 (net2, b, ci);
  and u3 (net3, ci, a);
  or  u4 (cout, net1, net2, net3);
endmodule
```



assign {cout, sum} = a + b + ci;

直接使用gate 去組合出combinational logic，通常用在netlist，一般RTL 不建議使用

Predefined Primitives

- Combinational logic gates
 - and, nand
 - or, nor
 - xor, xnor
 - buf, not

Continuous Assignments (4/4)

<assign> [#delay] [strength] <net_name> = <expression>

- LHS (left hand side)
 - To any **net** type
 - To bit- or part-selects of vectored nets // assign o2 [7:4] = 4'hc;
 - To several nets at once in a concatenation // assign {COUT, SUM} = A + B + CIN ;
- RHS (right hand side)
 - From any expression (**composed of nets or registers or both**), including a constant
// assign w = ({a, b} & c) | r ; assign w = 1'b1;
 - With a propagation delay and strengths // wire [7:0] #(3) o1 = in; //not recommended
 - From the return values of user-defined functions // assign w = f(...)

Continuous Assignments (2/4)

- The assignment is **always active** (continuous assignment)
 - Whenever any change on the RHS of the assignment occurs, it is evaluated and assigned to the LHS.

- You can make continuous assignments explicit or implicit:

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

```
assign a = b + c; // explicit
```

```
wire [ 3:0 ] a = b + c; // implicit
```

- It's not allowed which required a concatenation on the LHS.

```
wire [ 7:0 ] {co, sum} = a + b + ci;      → Error !!
```

```
assign {co, sum} = a + b + ci;            → OK !!
```

◆ **<assign> [#delay] [strength] <net_name> = <expression>**

◆ `wire [7:0] (strong1, weak0) #(3,5,2) o1 = in;` // strength and delays

以下語法，請在

`always @*`

`begin`

`.. = ...;`

`end`

COMBINATIONAL LOGIC 使用PROCEDURAL ASSIGNMENT

需要用多行連續表示的，就用procedural assignment

Outline

- Conditional statements
 - `if-else`
 - `case/casez/casex`
 - Common gotchas of conditional statements
- Looping statements
 - `for` loop
 - Other loops like `while` and `repeat` and `forever` are not synthesizable
 - Only allowed in the testbench

```
logic A;  
always @ (B or C)  
begin  
    A = ~(B & C);  
end
```

Isn't this just

```
logic A;  
assign A = ~(B & C);
```

Why bother?

Always blocks can contain more advanced control constructs

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );
```

```
    reg out;
```

```
    always @( * )
```

```
    begin
```

```
        if ( sel == 2'd0 )
```

```
            out = a;
```

```
        else if ( sel == 2'd1 )
```

```
            out = b;
```

```
        else if ( sel == 2'd2 )
```

```
            out = c;
```

```
        else
```

```
            out = d;
```

```
    end
```

```
endmodule
```

Nested if-else

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );
```

```
    reg out;
```

```
    always @( * )
```

```
    begin
```

```
        case ( sel )
```

```
            2'd0 : out = a;
```

```
            2'd1 : out = b;
```

```
            2'd2 : out = c;
```

```
            2'd3 : out = d;
```

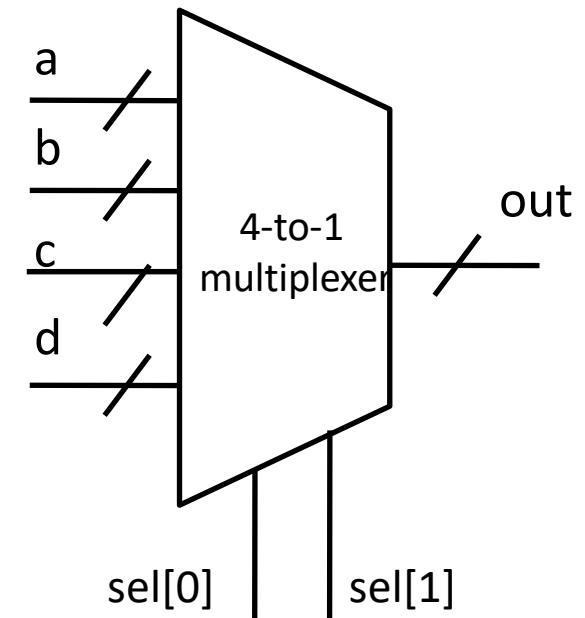
```
            default : out = a;
```

```
        endcase
```

```
    end
```

```
endmodule
```

case

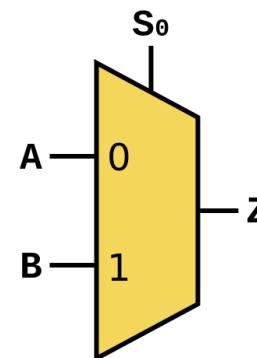


if...else

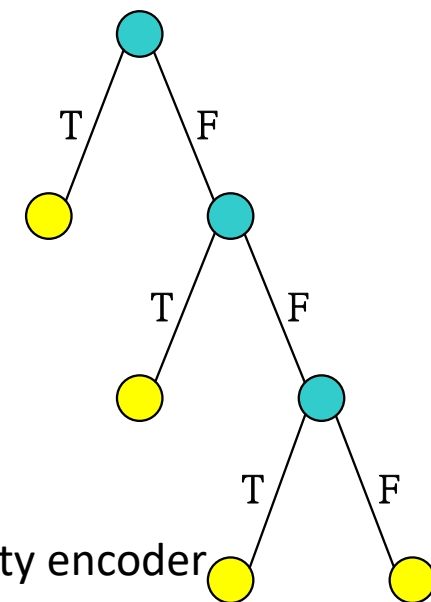
- **else** is paired with nearest **if** when ambiguous (use **begin...end** in nesting to clarify)
 - Beware of **no-else** condition => **latch** inference

```
if (a < b) c = d + 1;
if (a < b);
if (k == 1)
    begin : A_block
        sum_out = sum_reg;
        c_out = c_reg;
    end
```

```
if (a < b)
    sum = sum + 1;
else
    sum = sum + 2;
```



```
always_comb
    if (index < stage1)
        result = a + b;
    else if (index < stage2)
        result = a - b;
    else
        result = a;           Nesterov
```



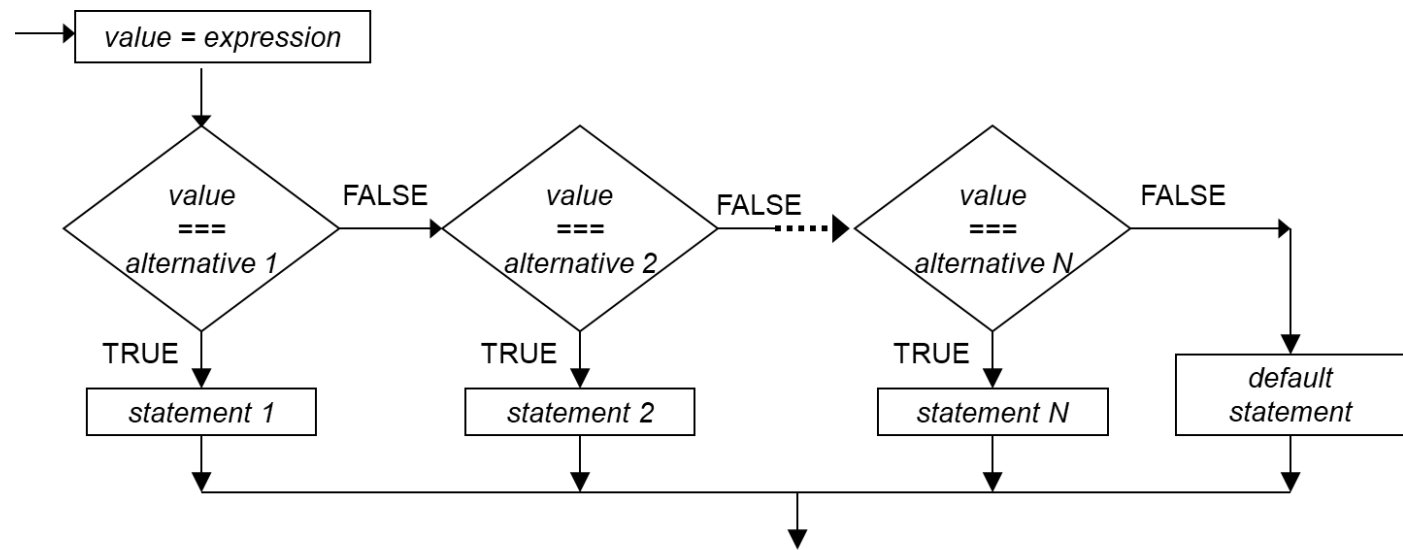
Nested if \Rightarrow priority encoder

default

case

- bit-by-bit comparison for an **exact match** (including x and z)
- **default** statement is optional

```
...  
always @ *  
begin  
  case (select)  
    0: y = a;  
    1: y = b;  
    2: y = c;  
    3: y = d;  
    default: y = 1'bx;  
  endcase  
end
```



Conditional statements (**case**)

- **case**, **casex**, **casez**: case statements are used for switching between multiple selections
 - If there are multiple matches only the first is evaluated
 - Breaks automatically
- **casez** treats Z as don't care
- **casex** treats Z and X as don't care

Help logic simplification

```
case (<expression>)  
    <alternative 1> : <statement 1>;  
    <alternative 2> : <statement 2>;  
    default         : <default statement>;  
endcase
```

case

- **case** is easier to read than a long string of if...else statements

```
module mux_2_to_1(a, b, out, sel);  
input a, b, sel;  
output out;  
reg out;
```

```
always @ (a or b or sel)
```

```
begin
```

```
    if (sel) out = a;  
    else out = b;
```

```
end
```

```
endmodule
```

=

```
case (sel)
```

```
    1'b1: out = a;
```

```
    1'b0: out = b;
```

```
endcase
```

case

```
module compute (result, rega, regb, opcode);
  input [7:0] rega, regb;
  input [2:0] opcode;
  output [7:0] result;
  reg [7:0] result;
  always @(rega or regb or opcode)
```

```
    case (opcode)
      3'b000 : result = rega + regb;
      3'b001 : result = rega - regb;
      ...
      3'b010 , //specify multiple cases with the same result(用","表示)
      3'b100 : result = rega / regb;
      default : begin
        result = 8'b0;
        $display ("no match");
      end
    endcase
```

```
endmodule
```

case, casez, casex

```
always @(s, a, b, c, d)
```

```
case (s)
```

```
    2'b00: out = a;
```

```
    2'b01: out = b;
```

```
    2'b10: out = c;
```

```
    2'b11: out = d;
```

```
endcase
```

```
always @*
```

```
casez (state)
```

```
// 3'b11z, 3'b1zz,...match 3'b1??
```

```
3'b1??: fsm = 0;
```

```
3'b01?: fsm = 1;
```

```
default: fsm = 1;
```

```
endcase
```

```
always @*
```

```
casex (state)
```

```
/*
```

```
during comparison : 3'b01z,
```

```
3'b01x, 3'b'011 ... match case
```

```
3'b01x
```

```
*/
```

```
3'b01x: fsm = 0 ;
```

```
3'b0xx: fsm = 1 ;
```

```
default: fsm = 1 ;
```

```
endcase
```

case, casez, casex

- case does not allow don't-care values (case: exact match (including x and z))
- casez allow both “z” and “?” values to be treated as don't-care values
- casex allows “z”, “x” and “?” to be treated as don't-care values

case \ input	x	z	?	1	0
	1	unmatch	unmatch	match	unmatch
0	unmatch	unmatch	unmatch	unmatch	match

match
unmatch

casez \ input	x	z	?	1	0
	x	match	match	unmatch	unmatch
z	match	match	match	match	match
?	match	match	match	match	match
1	unmatch	match	match	match	unmatch
0	unmatch	match	match	unmatch	match

casex \ input	x	z	?	1	0
	x	match	match	match	match
z	match	match	match	match	match
?	match	match	match	match	match
1	match	match	match	match	unmatch
0	match	match	match	unmatch	match

Comma Separated **Sensitivity List**

- @():
 - event control, enable the following expression when @(..) condition meets

```
always @(a or b or ci)
    sum = a + b + ci;

always @(posedge clk
        or negedge rst_n)
    if(rst_n = 1'b0)
        q <= 0;
    else
        q <= d;
```

Verilog-1995

```
always @(a, b, ci)
    sum = a + b + ci;

always @(posedge clk,
        negedge rst_n)
    if(rst_n = 1'b0)
        q <= 0;
    else
        q <= d;
```

Verilog-2001

Combinational Logic Sensitivity List

```
always @(a or b or sel)
  case(sel)
    1'b0: y = a;
    1'b1: y = b;
  endcase
```

Verilog-1995

```
always @* // @(*) is also OK
  case(sel)
    1'b0: y = a;
    1'b1: y = b;
  endcase
```

Verilog-2001

Gotcha: (if ... else)

- The statement occurs if the expressions controlling the if statement evaluates to true
 - True: 1 or non-zero value
 - False: 0 or ambiguous (X)
- Explicit priority

```
if (<expression>)  
// statement1  
else if (<expression>)  
// statement2  
else  
// statement3
```

```
always_comb  
begin  
    if (!WRITE)  
        begin  
            out = oldvalue;  
        end  
    else if (!STATUS)  
        begin  
            q = newstatus;  
        end  
end
```

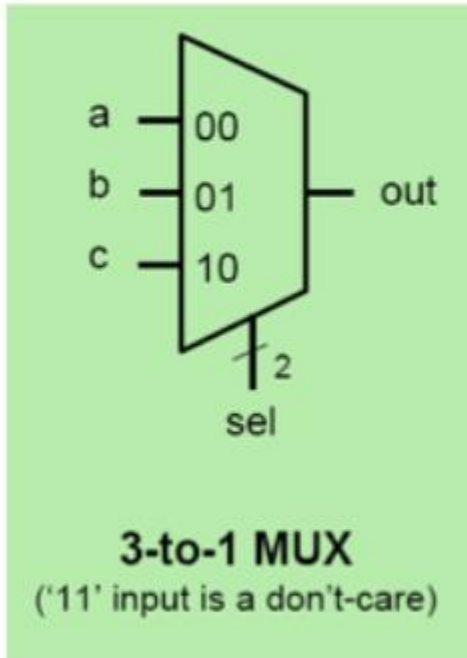


Something wrong here

What happens if the case statement is not complete?



Intention



What you may write, but wrong!

```
module maybe_mux_3to1(a, b, c,
                      sel, out);

    input [1:0] sel;
    input a,b,c;
    output out;
    reg out;

    always @(a or b or c or sel)
    begin
        case (sel)
            2'b00: out = a;
            2'b01: out = b;
            2'b10: out = c;
        endcase
    end
endmodule
```

**If sel = 3, mux will output the previous value.
What have we created?**

Incomplete Specification:

Added unwanted latch

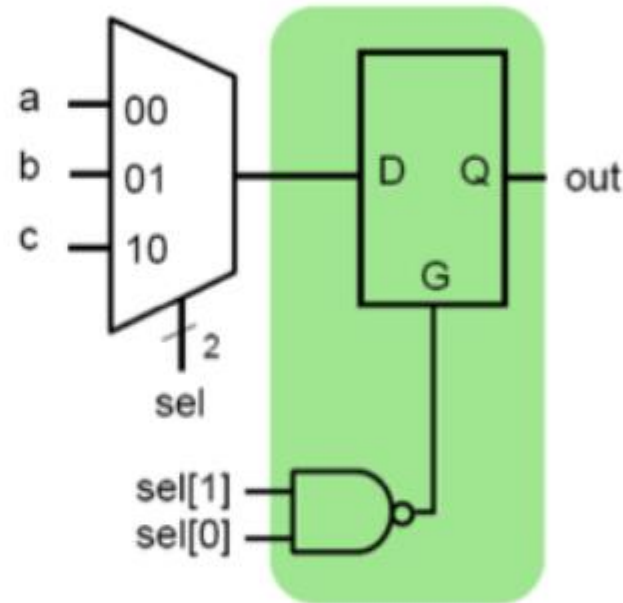


if out is not assigned during any pass through the always block, then **the previous value must be retained!**

```
module maybe_mux_3to1(a, b, c,
                      sel, out);
    input [1:0] sel;
    input a,b,c;
    output out;
    reg out;

    always @(a or b or c or sel)
    begin
        case (sel)
            2'b00: out = a;
            2'b01: out = b;
            2'b10: out = c;
        endcase
    end
endmodule
```

Synthesized Circuit



Unintentional latch
Similar for if-else

- ◆ When $sel = 2'b11$, $G = 0$, therefore the latch stores the previous output value as required by Verilog in this situation.

Always Avoid Incomplete Specification

Solution 1: precede all condition with a default statement for all signals

Solution 2: fully specified if-else, add default statement

```

module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

    reg out;

    always @( * )
    begin
        out = d; //default values, solution 1
        if ( sel == 2'd0 )
            out = a;
        else if ( sel == 2'd1 )
            out = b;
        else if ( sel == 2'd2 )
            out = c;
        //else out = d; //solution 2
    end
endmodule

```

```

module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

    reg out;

    always @( * )
    begin
        out = d; //default values, solution 1
        case ( sel )
            2'd0 : out = a;
            2'd1 : out = b;
            2'd2 : out = c;
            //default: out = d; //solution 2
        endcase
    end
endmodule

```

COMBINATIONAL LOGIC EXAMPLES

Code Converter (1/3)

- A code converter transforms one representation of data to another

.Ex: A BCD to excess-3 code converter

- BCD: Binary Coded Decimal
- Excess-3 code: the decimal digit plus 3

Truth Table for Code Converter Example

Input BCD				Output Excess-3			
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Code Converter (2/3)

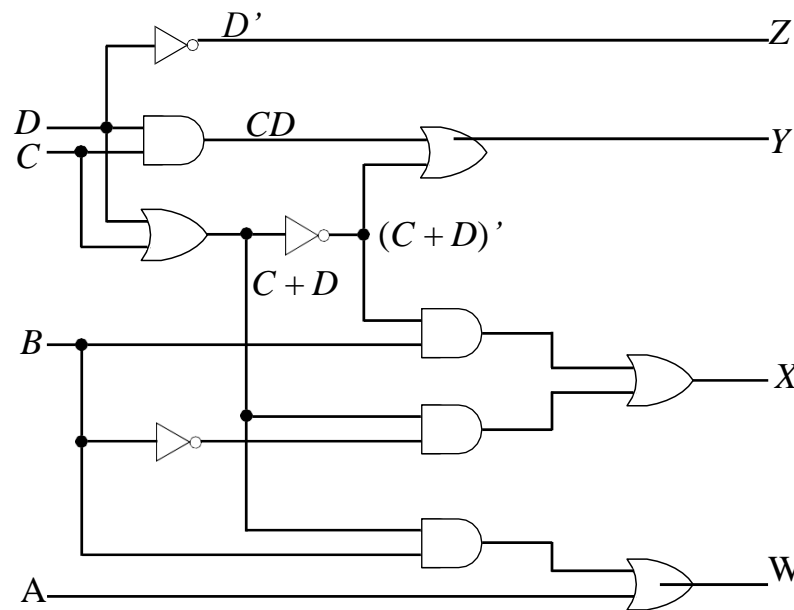
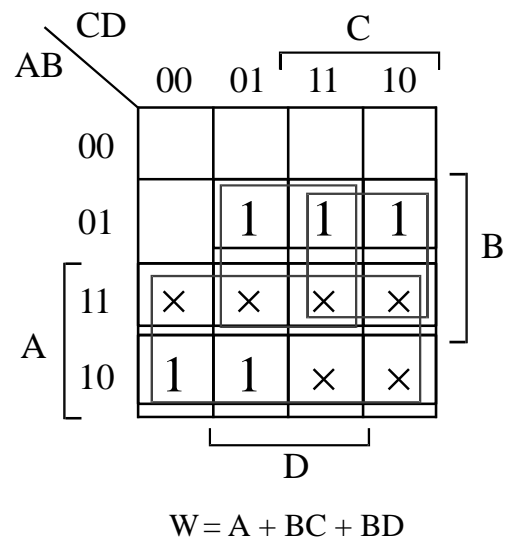
- Equations: (share terms to minimize cost)

$$W = A + BC + BD = A + B(C + D)$$

$$X = \overline{B}C + \overline{B}D + B\overline{C}\overline{D} = \overline{B}(C + D) + B\overline{C}\overline{D}$$

$$Y = CD + \overline{C}\overline{D} = \overline{C \oplus D}$$

$$Z = \overline{D}$$



Code Converter (3/3)

- RTL style

```
assign W = A|(B&(C|D));
assign X = ~B&(C|D)|(B&~C&~D);
assign Y = ~(C^D);
assign Z = ~D;
```

$$W = A + BC + BD = A + B(C + D)$$

$$X = \overline{B}C + \overline{B}D + B\overline{C}\overline{D} = \overline{B}(C + D) + B\overline{C}\overline{D}$$

$$Y = CD + \overline{C}\overline{D} = \overline{C} \oplus \overline{D}$$

$$Z = \overline{D}$$

- Behavior style

```
assign ROM_in = {A, B, C, D};
assign {W, X, Y, Z} = ROM_out;
always @(ROM_in) begin
    case (ROM_in)
        4'b0000: ROM_out = 4'b0011;
        4'b0001: ROM_out = 4'b0100;
        ...
        4'b1001: ROM_out = 4'b1100;
        default: ROM_out = 4'b0000;
    endcase
end
```

Truth Table for Code Converter Example

Input BCD				Output Excess-3			
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

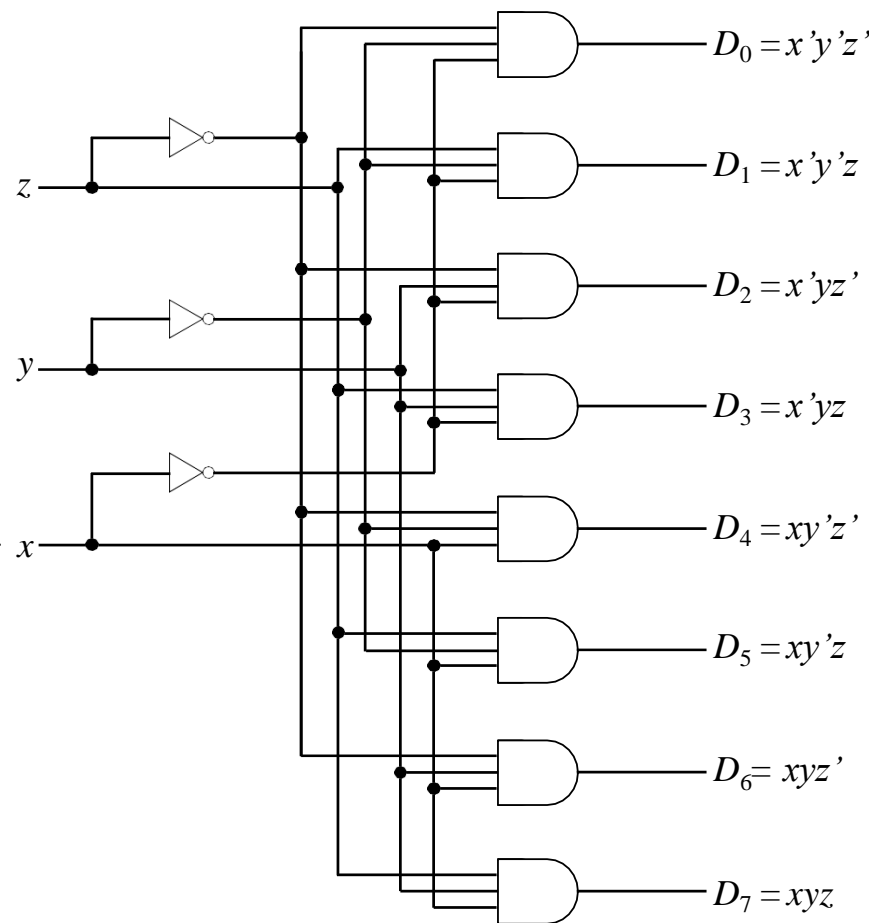
RTL: register transfer level

Decoder (1/2)

- A decoder is to generate the 2^n (or fewer) minterms of n input variables

Ex: a 3-to-8 line decoder

Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



Decoder (2/2)

- Behavior style 1

```
input x, y, z;
output [7:0] D;
reg [7:0] D;
always @(x or y or z) begin
    case ({x, y, z})
        3'b000: D = 8'b00000001;
        3'b001: D = 8'b00000010;
        ...
        3'b111: D = 8'b10000000;
        default: D = 8'b0;
    endcase
end
```

- Behavior style 2

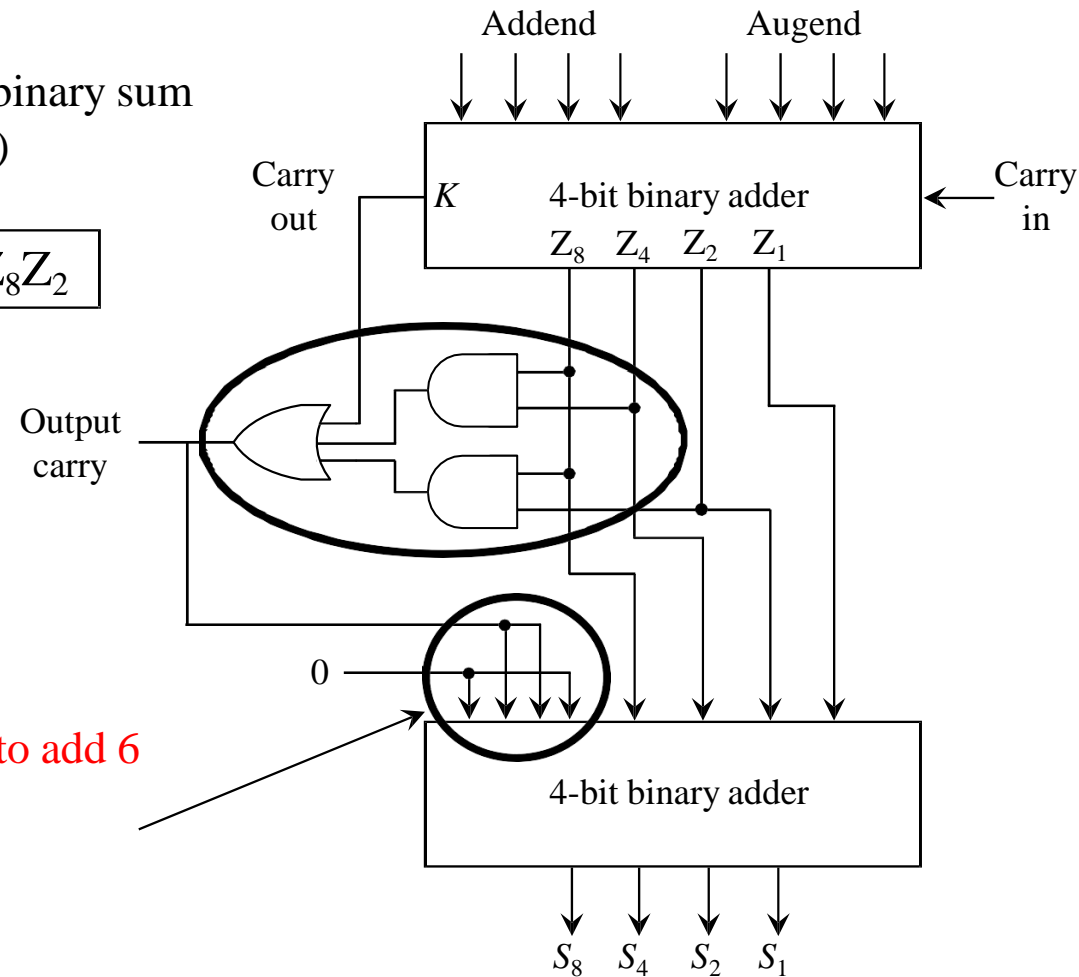
```
input x, y, z;
output [7:0] D;
wire [2:0] addr;
reg [7:0] D;
assign addr = {x, y, z};
always @(addr) begin
    D = 8'b0;
    D[addr] = 1;
end
```



BCD Adder (1/2)

C detects whether the binary sum is **greater than 9** (1001)

$$C = K + Z_8Z_4 + Z_8Z_2$$



If $C=1$, it is necessary to add 6 (0110) to binary sum

BCD Adder (2/2)

```
module bcd_add (S, Cout, A, B, Cin);  
    input [3:0] A, B;  
    input Cin;  
    output [3:0] S;  
    output Cout;  
    reg [3:0] S;  
    reg Cout;  
  
    always @(A or B or Cin) begin  
        {Cout, S} = A + B + Cin;  
        if (Cout != 0 || S > 9) begin  
            S = S + 6;  
            Cout = 1;  
        end  
    end  
end  
endmodule
```

以下語法，除latch 外，請在
always @(posedge clock or negedge reset_n)
begin
 .. <=
end

MODEL SEQUENTIAL LOGIC

請只拿來MODEL DFFS (建議，作業規定)

請用NON-BLOCKING ASSIGNMENT <=

Latch Inference (不建議使用，作業禁止)

- Incompletely specified wire in the synchronous section
- D latch

```
always @(enable or data)
    if (enable)
        q <= data;
```

DFF

• DFF with synchronous reset

```
module dff_sync_reset (DATA, CLK,
RESET, Q);
input DATA, CLK, RESET;
output Q;
reg Q;
```

```
always @(posedge CLK)
```

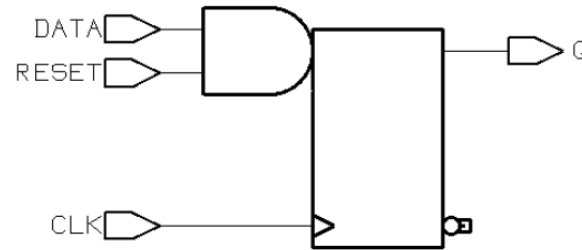
```
if (~RESET)
```

```
    Q <= 1'b0;
```

```
else
```

```
    Q <= DATA;
```

```
endmodule
```



• DFF with asynchronous reset

```
module dff_async_reset (DATA, CLK,
RESET, Q);
input DATA, CLK, RESET;
output Q;
reg Q;
always @(posedge CLK or negedge RESET)
```

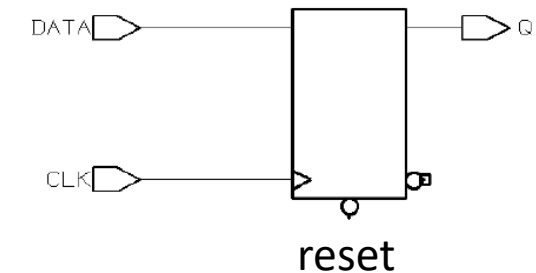
```
if (~RESET)
```

```
    Q <= 1'b0;
```

```
else
```

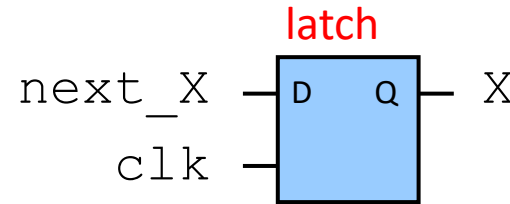
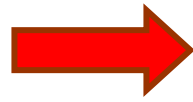
```
    Q <= DATA;
```

```
endmodule
```



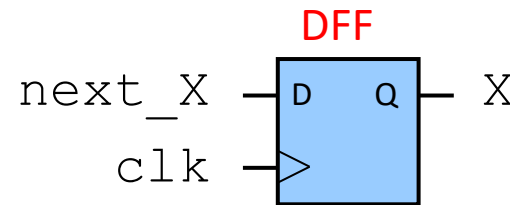
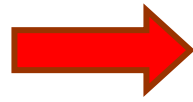
Common patterns for latch and flip-flop inference

```
always @( clk )
begin
  if ( clk )
    D <= Q;
end
```

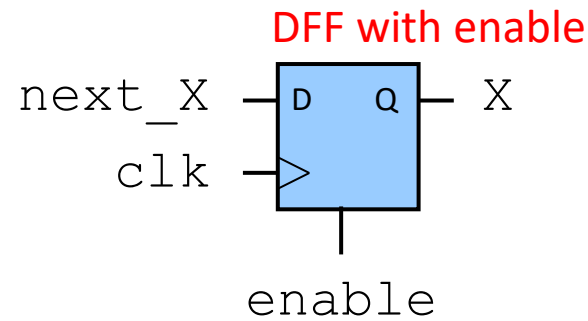
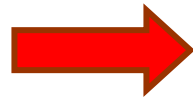


- Verilog uses **idioms** to describe latches, flip-flops and FSMs
- Other coding styles may simulate correctly but produce incorrect hardware

```
always @( posedge clk )
begin
  D <= Q;
end
```



```
always @( posedge clk )
begin
  if ( enable )
    D <= Q;
end
```



The Simplest Shift Register

```
always@(posedge clk)
begin
    outa<=in;
    outb<=outa;
    outc<=outb;
end
```

=

```
always@(posedge clk)
    outa<=in;
always@(posedge clk)
    outb<=outa;
always@(posedge clk)
    outc<=outb;
```



Better avoid this

