

**NTUEE DCLAB**

---

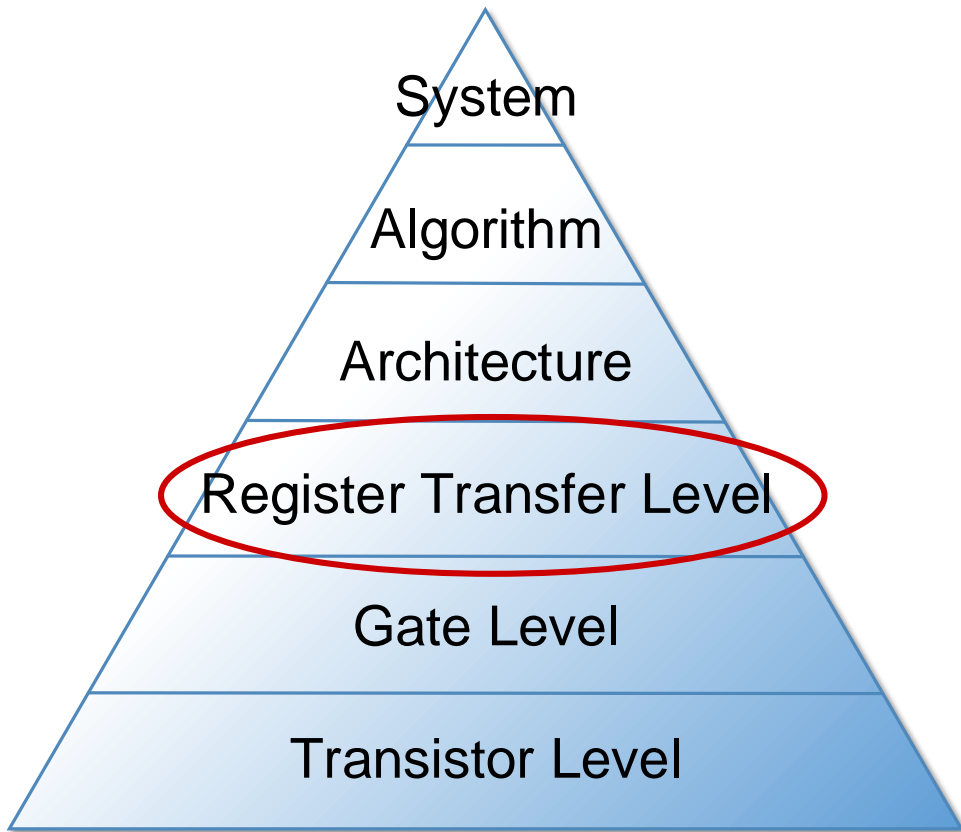
# **Introduction to Verilog/System Verilog**

---

**Dept. of Electrical Engineering, NTU  
National Taiwan University**

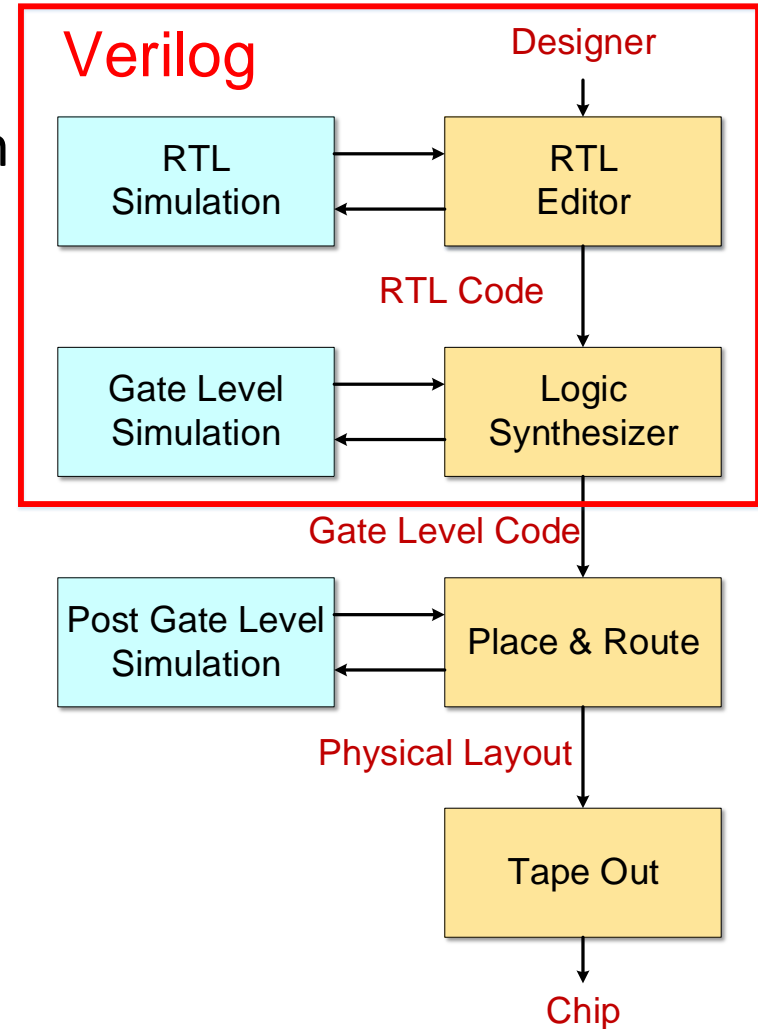
# What is Verilog HDL (Hardware Description Language)?

- ◆ Why using Hardware Description Language?
  - Hardware modeling
  - Reduce cost and time to design hardware
- ◆ Three popular HDLs
  - VHDL
  - Verilog
  - System Verilog



# What is Verilog?

- ◆ Verilog is a Hardware Description Language
  - Describes the flow of data between registers and how a design process these data
  - Model the timing
  - Express the *concurrency* of the system operation
  - Test the system

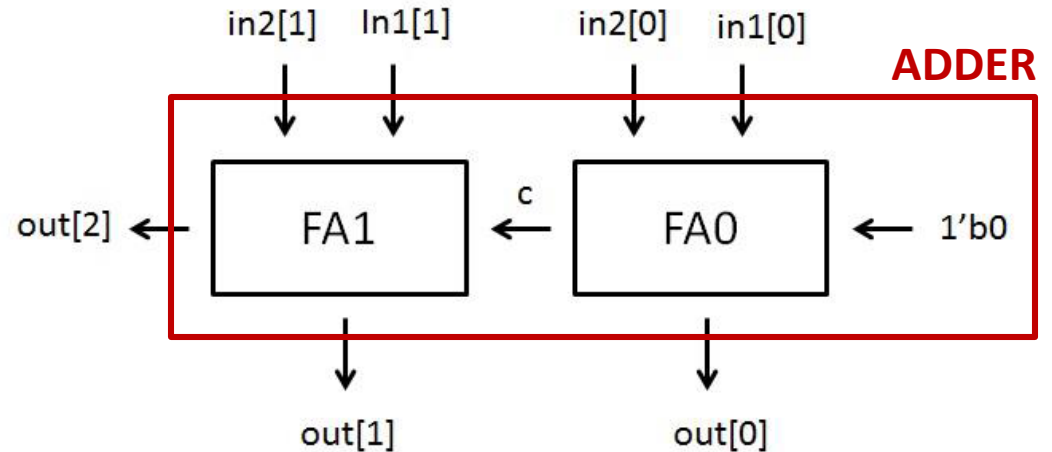


# RTL vs. Gate Level

## ◆ 2-bit Adder

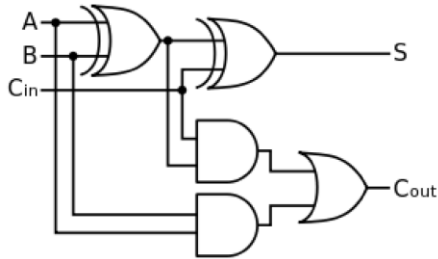
### RTL

```
module ADDER (out, in1, in2);  
    output [2:0] out;  
    input [1:0] in1, in2;  
  
    assign out = in1 + in2;  
  
endmodule
```



# RTL vs. Gate Level

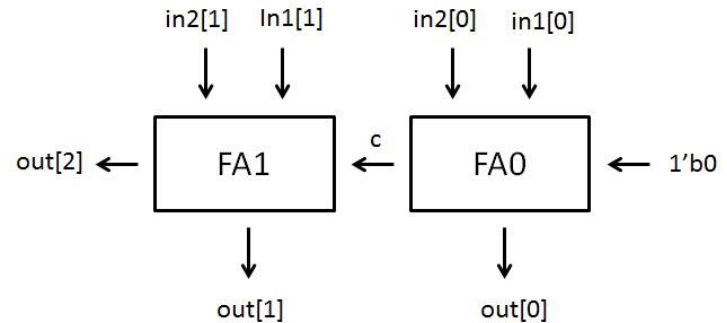
## Gate Level



```
module FA1 (Cout, S, A, B, Cin);  
    output Cout, S ;  
    input A, B, Cin;  
    wire N1, N2 , N3;
```

```
    EO XOR1(N1,A,B);  
    EO XOR2(s,N1,Cin);  
    AN2 AN2_1(N2,N1,Cin);  
    AN2 AN2_2(N3,A,B);  
    OR2 OR(Cout,N2,N3);
```

```
endmodule
```



```
module ADDER (out, in1, in2);  
    output [2:0] out;  
    input [1:0] in1, in2;  
    wire c;
```

```
    FA1 fa0(c, out[0], in1[0], in2[0], 1'b0);  
    FA1 fa1(out[2], out[1], in1[1], in2[1], c);
```

```
endmodule
```

# RTL Example

```
1. module speed_machine (
2     clock,      // system clock
3     reset,      // high-active asynchronous reset
4     accelerator, // input: accelerator signal
5     brake,       // input: brake signal
6     speed        // output: current speed
7 );
8
9 //==== PARAMETER DEFINITION =====
10 // using sequential code for state encoding
11 parameter stopped = 2'b00;
12 parameter s_low   = 2'b01;
13 parameter s_medium = 2'b10;
14 parameter s_high  = 2'b11;
15
16 //==== IN/OUT DECLARATION =====
17 2. input      clock, reset;
18    input      accelerator, brake;
19    output [1:0] speed;
20
21 //==== REG/WIRE DECLARATION =====
22 // wires ---
23 3. reg [1:0] next_state;
24    wire [1:0] next_speed;
25
26 //--- Flip-Flops ---
27 reg [1:0] state; // memory for current state
28 reg [1:0] speed; // memory for current output
29
30 //==== COMBINATIONAL CIRCUIT =====
31 // next-output logic (OL) ---
32 4. assign next_speed = state;
33
34 // next-state logic (NL) ---
35 4. always@ ( state or accelerator or brake ) begin
36     if( brake ) begin
37         case( state )
38             stopped: next_state = stopped;
39             s_low:   next_state = stopped;
40             s_medium: next_state = s_low;
41             s_high:  next_state = s_medium;
42             default: next_state = stopped;
43         endcase
44     end
45     else if( accelerator ) begin
46         case( state )
47             stopped: next_state = s_low;
48             s_low:   next_state = s_medium;
49             s_medium: next_state = s_high;
50             s_high:  next_state = s_high;
51             default: next_state = stopped;
52         endcase
53     end
54     else next_state = state;
55 end
56
57 //==== SEQUENTIAL CIRCUIT =====
58 // memory elements ---
59 4. always@ ( posedge clock or posedge reset ) begin
60     if( reset ) begin
61         state <= 2'd0;
62         speed <= 2'd0;
63     end
64     else begin
65         state <= next_state;
66         speed <= next_speed;
67     end
68 1. end
69 endmodule
```

source: CVSD "Behavior\_Modeling"

# Lexical Convention: Identifier and Comment

- ◆ Verilog is a **case sensitive** language
- ◆ Terminate lines with **semicolon ;**
- ◆ Identifiers
  - Starts *only* with a letter or an `_`(underscore), can be any sequence of letters, digits, `$`, `_`
  - Case-sensitive
    - E.g. `shiftreg_a`  
`_bus3`  
`n$657`  
`12_reg` ➡ **illegal**
- ◆ Comments
  - Single line: `//`
  - Multiple line: `/*...*/`

# Lexical Convention: Naming Conventions

- ◆ **Consistent naming** convention for the design
- ◆ Lowercase letters for signal names
- ◆ Uppercase letters for constants
- ◆ *clk* sub-string for clocks
- ◆ *rst* sub-string for resets
- ◆ Suffix
  - *\_n* for active-low, *\_z* for tri-state, *\_a* for async , ...
  - *\_r*, (*\_cs*) for current state, *\_w*, (*\_ns*) for next state
- ◆ Identical (similar) names for connected signals and ports
- ◆ Consistency within group, division and corporation



# Lexical Convention: Value Logic System

- ◆ 4-value logic system in Verilog: 0, 1, x or X, z or Z
  - 0
    - Zero, false, low
  - 1
    - One, true, high
  - X or x
    - unknown, occurs at un-initialized storage elements or un-resolvable logic conflicts
  - Z or z
    - High impedance, float

# Lexical Convention: Number

## ◆ Number specification: <size>'<base><value>

- Size: the size **in bits**
- Base:
  - b (binary), o (octal), d (decimal) or h(hexadecimal)
- Value: any legal number in selected base, including “x” and “z”

When <size> is *smaller than* <value>: *left-most bits of* <value> are truncated

When <size> is *larger than* <value>, then *left-most bits* are filled based on the value of the left-most bit in <value>

Left most '0' or '1' are filled with '0', 'Z' are filled with 'Z' and 'X' with 'X'

- Default size is **32-bits decimal** number

# Value and Number Examples

4'd10: 4-bit, 10, decimal

6'hca: 6-bit, store as 6'b001010 (truncated, not 11001010)

6'ha: 6-bit, store as 6'b001010 (filled with 2-bit '0' on left)

## ◆ Negative: : <size>'<base><value>

— E.g. -8'd3

8'd-3 //illegal

# Operator: Arithmetic and Bit-wise

Operator Type	Operator Symbol	Meaning
Arithmetic	+	Add
	-	Subtract
	*	Multiply
	/	Division
	%	Modulus (some synthesis tools don't support this operator)
Bit-wise Operator	~	NOT
	&	AND
		OR
	^	XOR

# Operator: Logical, Relational and Equality

Operator Type	Operator Symbol	Meaning
Logical: <b>return 1-bit</b>	!	Logic NOT
	&&	Logic AND
		Logic OR
Relational (conditional)	<=	Less than or equal
	<	Less than
	>=	Greater than and equal
	>	Greater than
Equality (conditional)	==	Equality
Equality (conditional)	!=	Inequality

# Operator: Logical vs. Bit-wise Operator

## ◆ Logical Operator: return **1-bit** true/false

— E.g. `a || b`, `a && b`

- `4'b1001 || 4'b1100 -> true`, `1'b1`
- `if( (a<=b)&&(c==d) || (e>f) )`

## ◆ Bit-wise Operator

— E.g. `a | b`

- `4'b1001 | 4'b1100 -> 4'b1101`

# Operator: Conditional Description

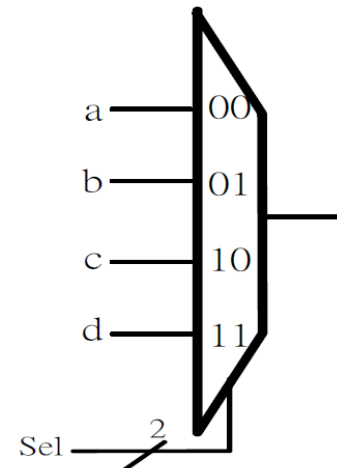
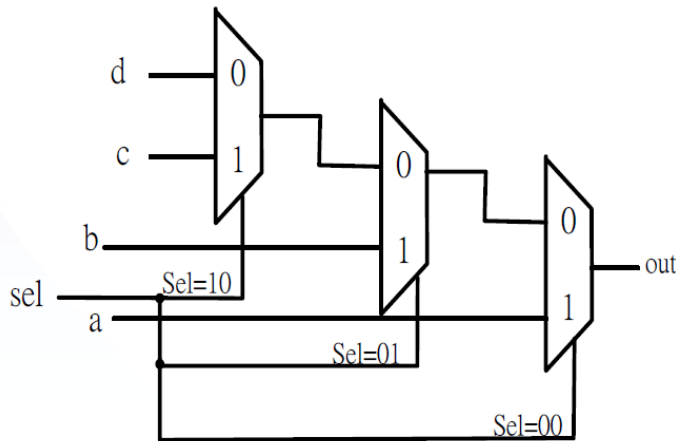
- ◆ **if elseif else**
- ◆ **? :**
  - $c = \text{sel} ? a : b;$
  - $d = \text{sel} ? e : f;$
- ◆ **case endcase**

```
if (sel == 1'b1)
begin
    c = a;
    d = e;
end
else begin
    c = b;
    d = f;
end
end
```

```
case (sel)
    1'b0: begin
        c = b;
        d = e;
    end
    1'b1: begin
        c = a;
        d = f;
    end
endcase
```

# Operator: if-then-else vs. Case

- ◆ ***if-then-else*** often infers a cascaded encoder
  - Inputs signals with different arrival time
- ◆ ***case*** infers a single-level mux
  - *case* is better if priority encoding is not required
  - *case* is generally simulated faster than *if-then-else*
- ◆ ***conditional assignment* (?) :**
  - Infers a mux with slower simulation performance





# Operator: Shift Operator

## ◆ Shift Operator (bit-wise)

—  $A = B \gg 2;$      $\rightarrow$  shift right B by 2-bit

•  $B=4'b1000;$   $A=B\gg 2;$   $A=4'b0010$

—  $A = B \ll 2;$      $\rightarrow$  shift left B by 2-bit

•  $B=4'b0100;$   $A=B\ll 2;$   $A=4'b0000$

## ◆ Shift Operator (arithmetic)

—  $A = B \ggg 2;$      $\rightarrow$  shift right B by 2-bit

•  $B=4'b1000;$   $A=B\ggg 2;$   $A=4'b1110$

•  $B=4'b0100;$   $A=B\ggg 2;$   $A=4'b0001$

—  $A = B \lll 2;$      $\rightarrow$  shift left B by 2-bit

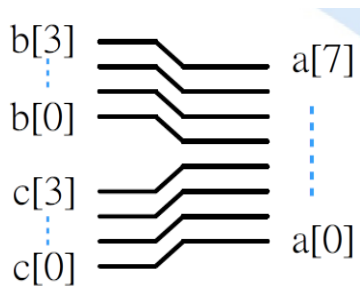
•  $B=4'b0100;$   $A=B\lll 2;$   $A=4'b0000$

# Operator: Concatenation

◆  $\{\}$   $\rightarrow a = \{b, c\}$

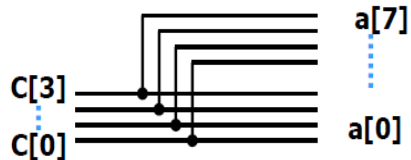
— E.g.  $a = \{b[3:0], c[3:0]\}$

$a = \{b[3], b[2], b[1], b[0], c[3], c[2], c[1], c[0]\}$



◆  $\{\{\}\}$   $\rightarrow a = \{2\{c\}\}$

—  $a = \{c, c\}$



◆  $a[4:0] = \{b[3:0], 1'b0\}; \rightarrow a = b \ll 1;$

# RTL Example: Revisit

```
1. module speed_machine (
2     clock,      // system clock
3     reset,      // high-active asynchronous reset
4     accelerator, // input: accelerator signal
5     brake,       // input: brake signal
6     speed        // output: current speed
7 );
8
9 //==== PARAMETER DEFINITION =====
10 // using sequential code for state encoding
11 parameter stopped = 2'b00;
12 parameter s_low   = 2'b01;
13 parameter s_medium = 2'b10;
14 parameter s_high  = 2'b11;
15
16 //==== IN/OUT DECLARATION =====
17 2. input      clock, reset;
18    input      accelerator, brake;
19    output [1:0] speed;
20
21 //==== REG/WIRE DECLARATION =====
22 // wires ---
23 3. reg [1:0] next_state;
24    wire [1:0] next_speed;
25
26 //--- Flip-Flops ---
27 reg [1:0] state; // memory for current state
28 reg [1:0] speed; // memory for current output
29
30 //==== COMBINATIONAL CIRCUIT =====
31 // next-output logic (OL) ---
32 4. assign next_speed = state;
33
34 // next-state logic (NL) ---
35 4. always@ ( state or accelerator or brake ) begin
36     if( brake ) begin
37         case( state )
38             stopped: next_state = stopped;
39             s_low:   next_state = stopped;
40             s_medium: next_state = s_low;
41             s_high:  next_state = s_medium;
42             default: next_state = stopped;
43         endcase
44     end
45     else if( accelerator ) begin
46         case( state )
47             stopped: next_state = s_low;
48             s_low:   next_state = s_medium;
49             s_medium: next_state = s_high;
50             s_high:  next_state = s_high;
51             default: next_state = stopped;
52         endcase
53     end
54     else next_state = state;
55 end
56
57 //==== SEQUENTIAL CIRCUIT =====
58 // memory elements ---
59 4. always@ ( posedge clock or posedge reset ) begin
60     if( reset ) begin
61         state <= 2'd0;
62         speed <= 2'd0;
63     end
64     else begin
65         state <= next_state;
66         speed <= next_speed;
67     end
68 1. end
69 endmodule
```

source: CVSD "Behavior\_Modeling"

# 1. Module

- ◆ Basic building blocks
- ◆ Begin with **module**, end with **endmodule**
- ◆ All modules run **concurrently**

```
module module_name(port_list);
```

```
//port declaration
```

```
//data type declaration
```

```
//task & function declaration
```

```
//module functionality or structure
```

```
endmodule
```

```
module test(Q, S, clk);  
input S,clk;  
output Q;  
reg Q;  
  
always@(*)  
    Q = (S & clk);  
  
endmodule
```

## 2. Module Ports

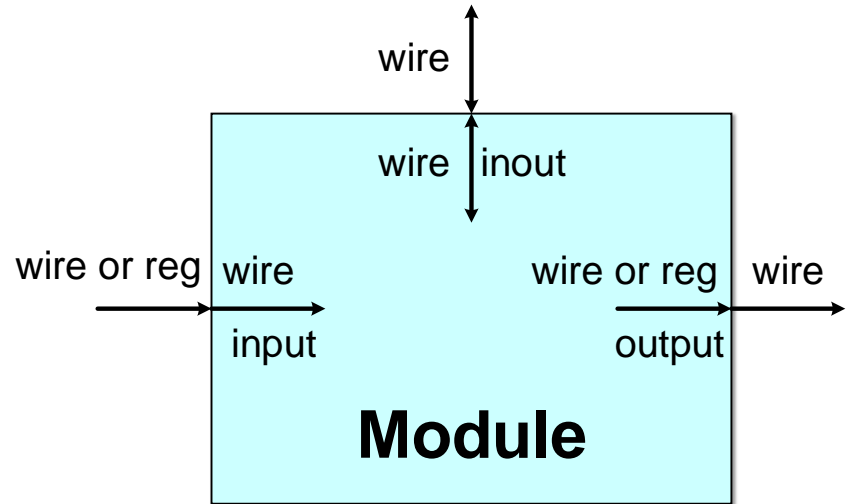
- ◆ Interface is defined by ports

- ◆ Port declaration

- input: input port
- output: output port
- inout: bidirectional port

- ◆ Port connection

- input: only wire can be assigned to represent this port **in** the module
- output: only wire can be assigned to represent this port **out** of module
- inout: **register assignment is forbidden** neither in module nor out of module *[Tri-state]*



## 2. Module Ports: Module Connection

- ◆ **Port ordering**

- One port per line with appropriate comments
- inputs first then outputs

- ◆ **Modules connected by port order (implicit)**

- Here order shall match correctly. Normally, it is not a good idea to connect ports implicitly. It could cause problem in debugging when any new port is added or deleted.
- e.g. FA U01( A, B, CIN, SUM, COUT );

- ◆ **Modules connect by name (explicit)**

**Use this**

- Use named mapping instead of positional mapping
- **Name shall match correctly**
- e.g. FA U01( .a(A), .b(B), .cin(CIN), .sum(SUM), .cout(COUT) );

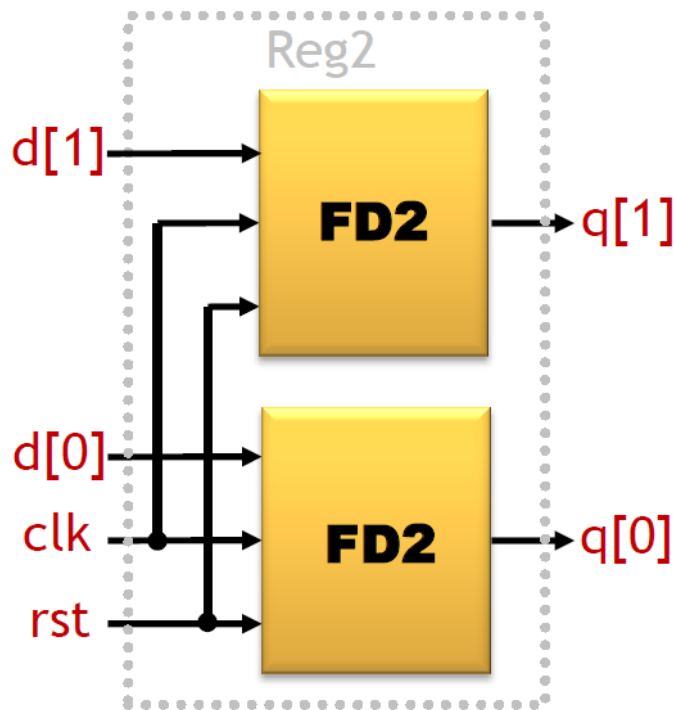
## 2. Module Ports: Examples

```
module MUX(out, a, b, sel, clk, rst);  
input sel, clk, rst;  
reg a, b; //incorrect define  
input a, b;  
output out;  
wire c;  
reg out;  
  
//Continuous assignment  
assign c = (sel==1'b0)? a:b;  
  
//Procedural assignment  
//only reg data type can be assigned value  
always@(posedge rst or posedge clk)  
begin  
    if(rst==1'b1) out <= 0;  
    else out <= c;  
end  
  
endmodule
```

```
`include "mux.v"  
module test  
reg a, b;  
reg clk, sel, rst;  
reg out; //incorrect define  
wire out;  
  
// 1. connect port by ordering  
MUX mux_1(out, a, b, sel, clk, reset);  
  
// 2. connect port by name  
MUX mux_2(.clk(clk), .rst(rst), .sel(sel),  
    .a(a), .(b)b, .out(out));  
  
...  
  
endmodule
```

## 2. Module Instance: Examples

- ◆ Create a higher-level system by connecting lower-level components



```
module Reg2(q, d, clk, rst);  
  input clk,rst;  
  input [1:0] d;  
  output [1:0] q;  
  
  FD2 f0(.Q(q[1]), .D(d[1]), .clk(clk),  
    .rst(rst));  
  FD2 f1(q[0], d[0], clk, rst);  
  
endmodule
```



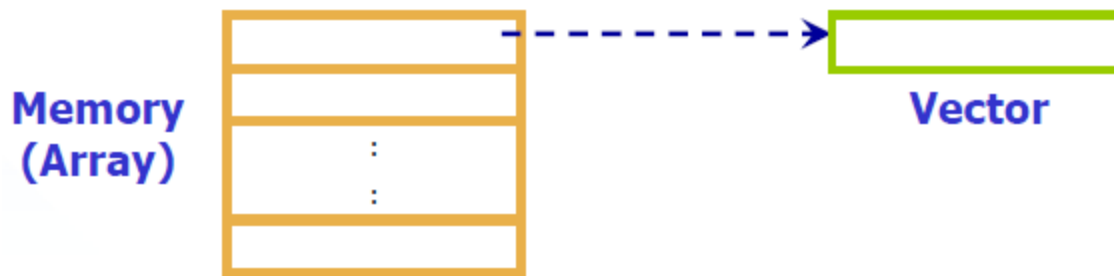
# 3. Data Type

- ◆ **Declaration Syntax** `<data_type>[MSB:LSB]<list_of_identifier>`
  - Nets: represent physical connections between devices (default=z)
    - `wire [MSB:LSB] variables;`
    - Represent connections between things
    - Cannot be assigned in an initial or always block
  - Register: represent abstract data storage element(default=x)
    - `reg [MSB:LSB] variables;`
    - Hold their value until explicitly assigned in an initial or always block

Register Type	Attribute
reg	Unsigned value with varying bit width
integer	32-bit <b>signed</b> (2's complement)
time	64-bit unsigned
real	Real number

### 3. Data Type: Vector and Array

- ◆ **Vectors: the wire and register can be represented as a vector**
  - wire [7:0] vec; → 8-bit bus
  - reg [0:7] vec; → vec[0] is the MSB
- ◆ **Arrays: <array\_name>[<subscript>]**
  - It isn't well for the backend verifications
  - integer mem[0:7] → (8x32)-bit mem
  - reg [7:0] mem[0:1023]
- ◆ **Difference between Vector and Array**
  - Vector: single-element with multiple-bit
  - Array: multiple-element with multiple-bit



## 4. Data Assignment

### ◆ Continuous Assignment for wire assignment

- Implies that whenever any change on the RHS of the assignment occurs, it is evaluated and assigned to the LHS.

- E.g. wire[3:0] a;

assign a = b + c;     // continuous assignment

### ◆ Procedural Assignment for reg assignment

- Assignment to “register” data types may occur within *always*, *initial*, *task* and *function*. These expressions are controlled by **triggers** which cause the assignment to evaluate

- E.g. reg a, b;

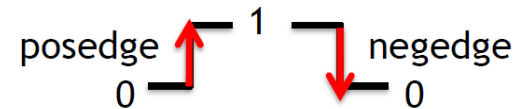
always@(b)                     //procedural assignment with triggers

a = ~b;

## 4. Case Study: Behavioral Modeling

### ◆ Description in Verilog

- assign
- always
  - Used in event-driven expression
  - Must contain a sensitive list
    - posedge, negedge



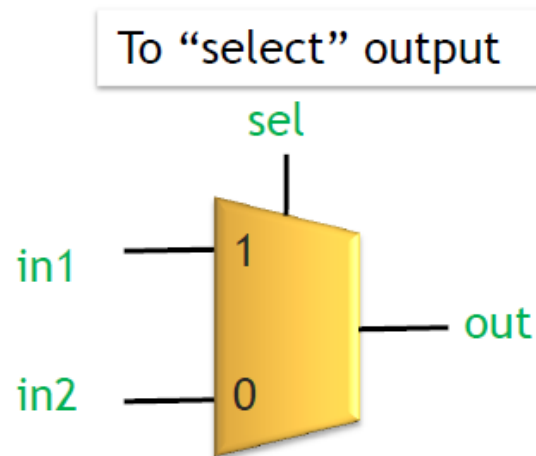
```
always @(a or b or c)
begin
    output1 = a & b & c;
end
```

When a/b/c changes, (0->1/1->0)  
output1 will be updated

```
always @(posedge clk or negedge rst)
begin
    if (!rst) output1 <= 1'b0;
    else      output1 <= next_output;
end
```

Output1 will change when at posedge of  
clock or negedge of reset

## 4. Case Study: Mux



```
if(sel==1)
    out = in1;
else
    out = in2;
```

```
out = (sel&in1)+(sel'&in2)
```

```
module mux2(out,in1,in2,sel);
input in1,in2,sel;
output out;

assign out = sel?in1:in2;

endmodule
```

```
module mux2(out,in1,in2,sel);
input in1,in2,sel;
output out;
reg out

always@(in1 or in2 or sel)
begin
    if(sel) out = in1;
    else    out = in2;
end

endmodule
```

# Difference in System Verilog

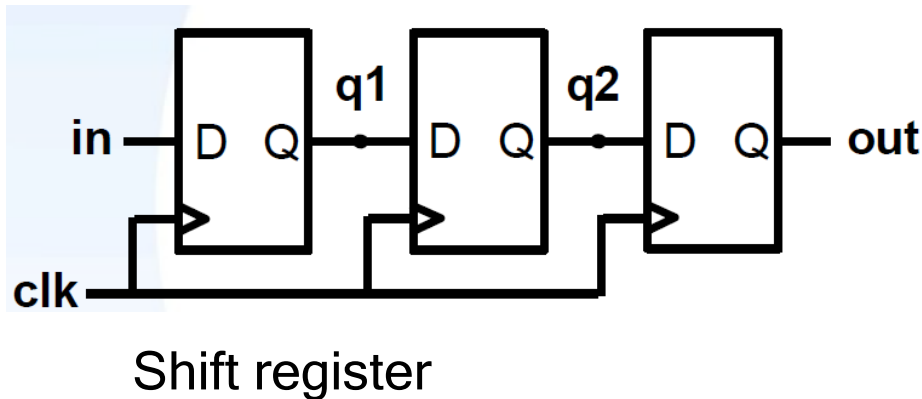
- ◆ In System Verilog, **logic** is used to replace **reg** and **wire** in Verilog
- ◆ In System Verilog
  - Combinational circuit
    - **always\_comb** is used to replace **always@(\*)**
  - Sequential circuit
    - **always\_ff@(posedge clk)** is used to replace **always@(posedge clk)**

# 5. Blocking and Non-Blocking

## ◆ Non-Blocking

```
always@(posedge clk)
begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
end
```

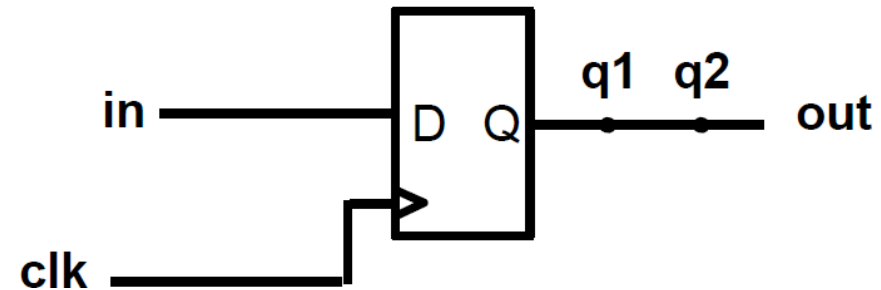
Concurrent execution



## ◆ Blocking

```
always@(posedge clk)
begin
    q1 = in;
    q2 = q1;
    out = q2;
end
```

Sequential execution



# References

---

- ◆ James M. Lee, “Verilog® Quickstart A Practical Guide to Simulation and Synthesis in Verilog,” Springer, 2005.
- ◆ S. Churiwala and S. Garg, “Principles of VLSI RTL Design,” Springer, 2011.
- ◆ M. Keating, “The Simple Art of SoC Design - Closing the Gap between RTL and ESL,” Springer, 2011.
- ◆ NTU VLSI Lab course material, Lec 1, 2014
- ◆ NCTU IC Lab course material, Lec 1-3, 2014
- ◆ CIC Training Manual