

Computer-Aided VLSI System Design

Chap.2-1 Logic Design at Register-Transfer Level

Lecturer: 張惇宥

Graduate Institute of Electronics Engineering, National Taiwan University



NTU GIEE



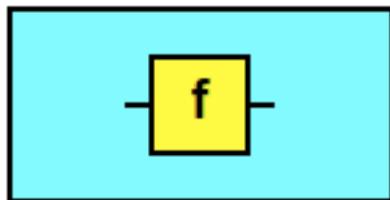
Outline

- ❖ Definition of Register-Transfer Level (RTL)
- ❖ Verilog Syntax of RTL
 - Operands
 - Operators
- ❖ Continuous Assignments for Combinational Circuits
- ❖ Signing and Sizing in Verilog



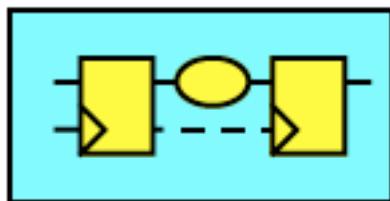
Brief Review of Modeling Levels

Behavioral Level



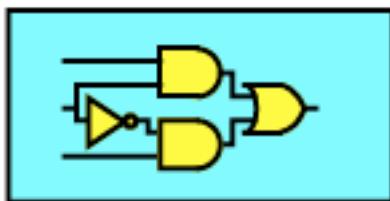
Most used modeling level, easy for designer, can be simulated and synthesized to gate-level by EDA tools

Register Transfer Level (RTL)



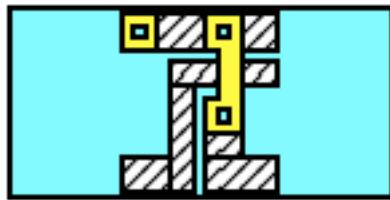
Common used modeling level for small sub-modules, can be simulated and synthesized to gate-level

Structural/Gate Level



Usually generated by synthesis tool by using a front-end cell library, can be simulated by EDA tools. A gate is mapped to a cell in library

Transistor/Physical Level



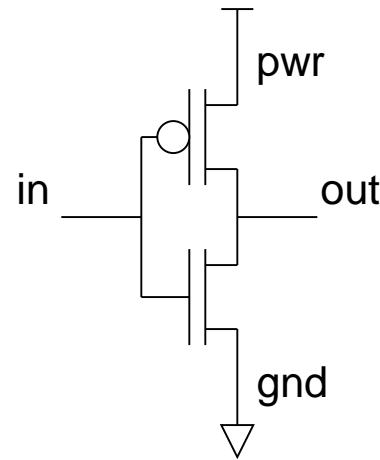
Usually generated by synthesis tool by using a back-end cell library, can be simulated by SPICE



Various Abstraction of Verilog (1/4)

- ❖ Transistor level Verilog description of a inverter

```
module inv(out, in);
// port declaration
output    out;
input     in;
// declare power and ground
supply1  pwr;
supply0  gnd;
// Switch level description
pmos   S0(out, pwr, in);
nmos   S1(out, gnd, in);
endmodule
```

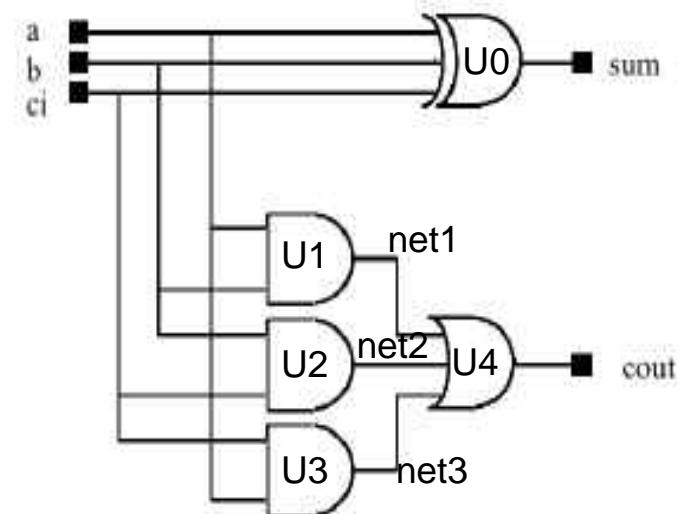




Various Abstraction of Verilog (2/4)

- ❖ Gate level Verilog description of a full-adder

```
module fadder(cout, sum, a, b, cin);
// port declaration
    output cout, sum;
    input a, b, cin;
    wire net1, net2, net3;
// Netlist description
    xor U0(sum,a,b,cin);
    and U1(net1, a, b);
    and U2(net2, b, cin);
    and U3(net3, cin, a);
    or U4(cout, net1, net2, net3);
endmodule
```





Various Abstraction of Verilog (3/4)

- ❖ RT-Level (RTL) Verilog description of a full adder

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

// RTL description
    assign sum = a^b^cin;
    assign cout = (a&b)|(b&cin)|(cin&a);

endmodule
```

Whenever a or b or c changes its logic state, evaluate sum and $cout$ by using the equation

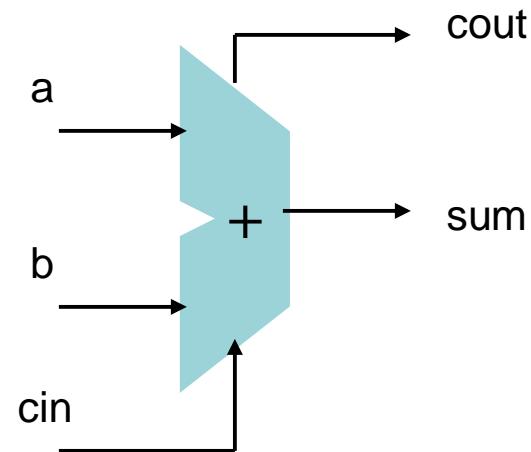
$$sum = a \oplus b \oplus ci$$
$$cout = ab + bc + ca$$



Various Abstraction of Verilog (4/4)

- ❖ Behavioral level Verilog description of a full adder

```
module fadder(cout, sum, a, b, cin);
// port declaration
  output cout, sum;
  input a, b, cin;
  reg cout, sum;
// behavior description
  always @(a or b or cin)
    begin
      {cout,sum} = a + b + cin;
    end
endmodule
```





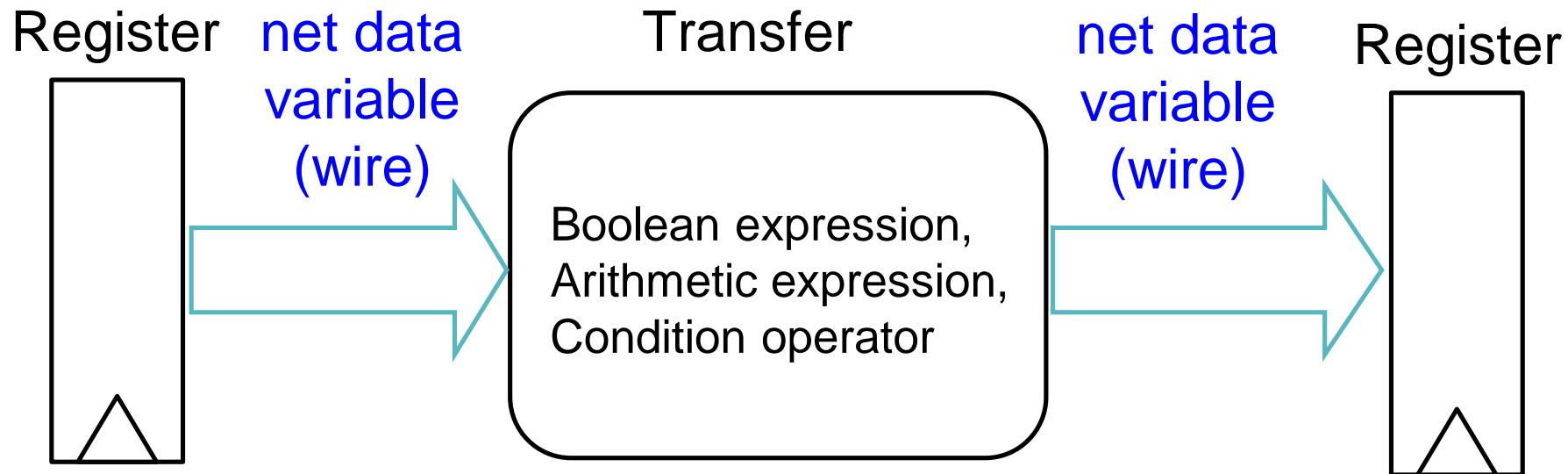
What is Register Transfer Level?

- ❖ In integrated circuit design, Register Transfer Level (RTL) description is a way of describing the operation of a **synchronous digital circuit**.
- ❖ In RTL design, a circuit's behavior is defined in terms of the flow of signals (or transfer of data) between synchronous registers and the logical operations performed on those signals.
- ❖ Modern RTL code: “**Any code that is synthesizable is called RTL code**”



Description at RT-Level

- ❖ Register (memory, usually D Flip-Flops)
- ❖ Transfer (operation, **combinational**)





Procedures to Design at RT-Level

❖ Design partition

- Sequential circuit (register part)
- Combinational circuit (transfer part)

❖ Net declaration

- I/O ports
- Net variable

❖ Transfer Circuit design

- Logical operator
- Arithmetical operator
- Conditional operator



Example

```
wire [7:0] in1;
wire [7:0] in2;
wire [7:0] in3;                                //input signals of transfer part

wire [7:0] out1;
wire [7:0] out2;
wire [7:0] out3;                                //output signals of transfer part

assign out1 = in1 & in2;
assign out2 = in1 + in3;
assign out3 = in1[0]==1'b1 ? in2 : in3;      //transfer description
```



Outline

- ❖ Definition of Register-Transfer Level (RTL)
- ❖ Verilog Syntax of RTL
 - Operands
 - Operators
- ❖ Continuous Assignments for Combinational Circuits
- ❖ Signing and Sizing in Verilog



Operands

- ❖ Data types **wire** & **reg** are used for operands in RTL/Behavioral Level description
- ❖ Since all metal wires in circuits only represent 0 or 1, we should consider all the variables in binary
 - Unsigned: Ordinary binary
 - Signed: **2's complement** binary
 - Since there's only 0/1 for each bit, **whether the variable is unsigned or signed is up to your recognition**
 - Simulators do all operations in binary (i.e., 0/1). If you check all the operations in binary, you'll never get wrong arithmetic/logic results

```
reg [4:0] a;  
reg signed [4:0] a;
```



Operators

Concatenation and replications	{,}
Negation	!, ~
Unary reduction	&, , ^, ^~ ...
Arithmetic	+ , - , * , / , %
Shift	>> , <<
Relational	< , <= , > , >=
Equality	== , != , ==== , !==
Bitwise	~, &, , ^
Logical	&& ,
Conditional	? :



Concatenation & Replication

❖ Concatenation and Replication Operator ({}))

Concatenation operator in LHS

```
module add_32 (co, sum, a, b, ci);  
    output co;  
    output [31:0] sum;  
    input  [31:0] a, b;  
    input  ci;  
    assign #100 {co, sum} = a + b + ci;  
endmodule
```

Bit replication to produce *01010101*

```
assign byte = {4{2'b01}};
```

Sign Extension

```
assign word = {{8{byte[7]}}, byte};
```



Negation Operators

- ❖ The logical negation operator (!)
 - produces a 0, 1, or X scalar value
- ❖ The bitwise negation operator (~)
 - inverts each individual bit of the operand

```
module negation;
    initial begin
        $displayb( !4'b0100 ); // 0
        $displayb( !4'b0000 ); // 1
        $displayb( !4'b00z0 ); // x
        $displayb( !4'b000x ); // x
        $displayb( ~4'b01zx ); // 10xx
    end
endmodule
```



Unary Reduction Operators

- ❖ The unary reduction operators (**&**, **|**, **^**, **^~**) produce a 0, 1, or X scalar value.

```
module reduction;
    initial begin
        $displayb( &4'b1110 ); // 0
        $displayb( &4'b1111 ); // 1
        $displayb( &4'b111z ); // x
        $displayb( &4'b111x ); // x
        $displayb( |4'b0000 ); // 0
        $displayb( |4'b0001 ); // 1
        $displayb( |4'b000z ); // x
        $displayb( |4'b000x ); // x
        $displayb( ^4'b1111 ); // 0
        $displayb( ^4'b1110 ); // 1
        $displayb( ^4'b111z ); // x
        $displayb( ^4'b111x ); // x
        $displayb( ^~4'b1111 ); // 1
        $displayb( ^~4'b1110 ); // 0
        $displayb( ^~4'b111z ); // x
        $displayb( ^~4'b111x ); // x
    end
endmodule
```



Arithmetic Operators

- ❖ The arithmetic operators (**, *, /, %, +, -)
 - Produce numerical or unknown results
 - Integer division discards any remainder
 - An unknown operand produces an unknown result
 - Assignment of a signed value to an unsigned register is 2's-complement

```
module arithmetic;
    initial begin
        $display( -3 * 5 ); // -15
        $display( -3 / 2 ); // -1
        $display( -3 % 2 ); // -1
        $display( -3 + 2 ); // -1
        $display( 2 - 3 ); // -1
        $displayh( 32'hfffffd / 2 ); // 7fffffe
        $displayb( 2 * 1'bx); // xx...
    end
endmodule
```

Treat the all signal as binary signal !!!

How to deal with fraction number???

→ Fixed point representation



Example (Arithmetic)

```
module DUT (sum,diff1,diff2,negA,A,B);
    output [4:0] sum,diff1,diff2,negA;
    input  [3:0] A , B

    assign sum = A + B;
    assign diff1 = A - B;
    assign diff2 = B - A;
    assign neg = -A ;
endmodule
```

t_sim	A	B	sum	diff1	diff2	negA
5	5	2	7	3	29	27
15	0101	0010	00111	00011	11101	11011



Shift Operators

❖ Shift operator

- “**>>**” logical shift right
- “**<<**” logical shift left
- “**>>>**” arithmetic shift right (fills in signed bits if the signal is signed)
- “**<<<**” arithmetic shift left

❖ Treat the right operand as unsigned

```
module shift;
    initial begin
        $displayb ( 8'b000011000 << 2      ) ; // 01100000
        $displayb ( 8'b000011000 >> 2      ) ; // 00000110
        $displayb ( 8'b000011000 >> -2     ) ; // 00000000
        $displayb ( 8'b000011000 >> 1'bx ) ; // xxxxxxxx
    end
endmodule
$displayb (8'sb11001100 >>> 2); // 11110011
$displayb (8'sb11001100 <<< 2); // 00110000
```

-2 = 1110 (2's comp)
→ Take it as unsigned
→ 1110 = 14



Relational Operators

- ❖ Relational operators (< , <= , >= , >)

```
module relational;
    initial begin
        $displayb ( 4'b1010 < 4'b0110 ) ; // 0
        $displayb ( 4'b0010 <= 4'b0010 ) ; // 1
        $displayb ( 4'b1010 < 4'b0x10 ) ; // x
        $displayb ( 4'b0010 <= 4'b0x10 ) ; // x
        $displayb (| 4'b1010 >= 4'b1x10 ) ; // x
        $displayb ( 4'b1x10 > 4'b1x10 ) ; // x
        $displayb ( 4'b1z10 > 4'b1z10 ) ; // x
    end
endmodule
```



Equality Operators

❖ Equality operators

- “`==`” , “`!=`” don't perform a definitive match for Z or X.
- “`====`” , “`!==`” does perform a definitive match for Z or X.

```
module equality;
    initial begin
        $displayb ( 4'b0011 == 4'b1010 ); // 0
        $displayb ( 4'b0011 != 4'b1x10 ); // 1
        $displayb ( 4'b1010 == 4'b1x10 ); // x
        $displayb ( 4'b1x10 == 4'b1x10 ); // x
        $displayb ( 4'b1z10 == 4'b1z10 ); // x
    end
endmodule
```

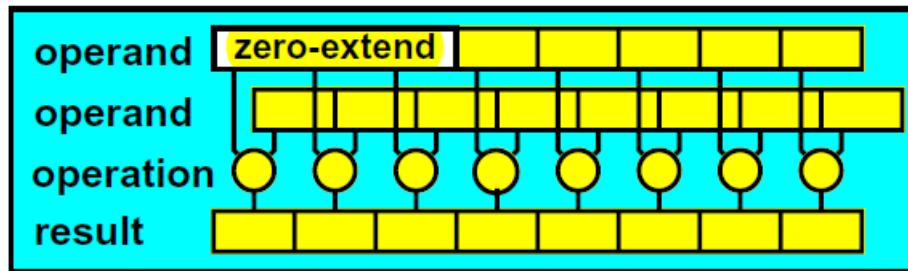
```
module identity();
    initial begin
        $displayb ( 4'b01zx === 4'b01zx ); // 1
        $displayb ( 4'b01zx !== 4'b01zx ); // 0
        $displayb ( 4'b01zx === 4'b00zx ); // 0
        $displayb ( 4'b01zx !== 4'b11zx ); // 1
    end
endmodule
```

avoid using in DUT



Bit-Wise Operators

- ❖ Bit-wise operators ($\&$, $|$, $^$, $^~$)
 - operate on each individual bit of a vector



```
module bit_wise;
    initial begin
        $displayb ( 4'b01zx & 4'b0000 ) ; // 0000
        $displayb ( 4'b01zx & 4'b1100 ) ; // 0100
        $displayb ( 4'b01zx & 4'b1111 ) ; // 01xx
        $displayb ( 4'b01zx | 4'b1111 ) ; // 1111
        $displayb ( 4'b01zx | 4'b0011 ) ; // 0111
        $displayb ( 4'b01zx | 4'b0000 ) ; // 01xx
        $displayb ( 4'b01zx ^ 4'b1111 ) ; // 10xx
        $displayb ( 4'b01zx ^~ 4'b0000 ) ; // 10xx
    end
endmodule
```



Logical Operators

❖ Logical operators (&&, ||)

- An operand is logically false if **all** of its bits are 0
- An operand is logically true if **any** of its bits are 1

```
module logical;
    initial begin
        $displayb ( 2'b00 && 2'b10 ) ; // 0
        $displayb ( 2'b01 && 2'b10 ) ; // 1
        $displayb ( 2'b0z && 2'b10 ) ; // x
        $displayb ( 2'b0x && 2'b10 ) ; // x
        $displayb ( 2'b1x && 2'b1z ) ; // 1
        $displayb ( 2'b00 || 2'b00 ) ; // 0
        $displayb ( 2'b01 || 2'b00 ) ; // 1
        $displayb ( 2'b0z || 2'b00 ) ; // x
        $displayb ( 2'b0x || 2'b00 ) ; // x
        $displayb ( 2'b0x || 2'b0z ) ; // x
    end
endmodule
```

Usually connect two Boolean value
(e.g. (a > b) && (a > 0))



Conditional Operators

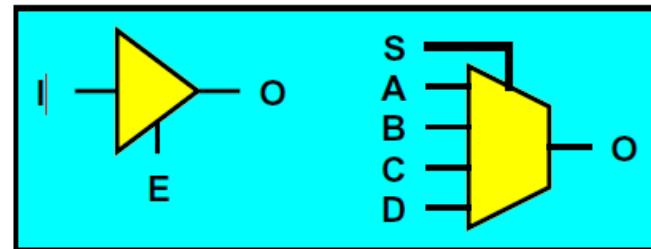
❖ Conditional Operator

- Usage:

conditional_expression ? true_expression: false_expression;

```
module driver(O,I,E);
output O; input I,E;
  assign O = E ? I : 'bz;
endmodule

module mux41(O,S,A,B,C,D);
output O;
input A,B,C,D; input [1:0] S;
  assign O = (S == 2'h0) ? A :
              (S == 2'h1) ? B :
              (S == 2'h2) ? C : D;
endmodule
```





Operator Precedence

Type of Operators	Symbols					
Concatenate & replicate	{ }	$\{{\ }}$				
Unary	!	\sim	&	\wedge	$\wedge\sim$	
Arithmetic	*	/	%			
	+	-				
Logical shift	$<<$	$>>$				
Relational	<	\leq	>	\geq		
Equality	\equiv	\neq	$\equiv\equiv$	$\neq\equiv$		
Binary bit-wise	&	\wedge	$\wedge\sim$			
Binary logical	$\&\&$	\parallel				
Conditional	? :					

Use parentheses (i.e. “()”) to explicitly define the operation order



Outline

- ❖ Definition of Register-Transfer Level (RTL)
- ❖ Signing and Sizing in Verilog
- ❖ Verilog Syntax of RTL
 - Operands
 - Operators
- ❖ Continuous Assignments for Combinational Circuits
- ❖ Signing and Sizing in Verilog



Procedures

- ❖ **Procedure block:** a group of statements that appear between a begin and an end
 - **initial, always, task, function**
 - considered as a statement – can be nested
- ❖ Procedures execute **concurrently** with other procedures



Assignments (1/2)

- ❖ Assignment: Drive value onto nets and registers
- ❖ There are two basic forms of assignment
 - **continuous assignment**, which assigns values to nets
 - **procedural assignment**, which assigns values to registers
- ❖ Basic form

<left-hand side> = <right-hand side>

Statement type	Left-hand side (LHS)
Continuous assignment	Net* <code>wire</code> , <code>tri</code>
Procedural assignment	Register* <code>reg</code> , <code>integer</code> , <code>real</code>

- * Include scalar, vector, concatenation, and bit/part-select
- May synthesizable



Assignments (2/2)

❖ Continuous assignment

```
module holiday_1(sat, sun, weekend);
    input sat, sun; output weekend;
    assign weekend = sat | sun;      // outside a procedure
endmodule
```

❖ Procedural assignment

```
module holiday_2(sat, sun, weekend);
    input sat, sun; output weekend; reg weekend;
    always #1 weekend = sat | sun; // inside a procedure
endmodule
```

```
module assignments
    // continuous assignments go here
    always begin
        // procedural assignments go here
    end
endmodule
```



Continuous Assignments (1/5)

❖ Syntax of continuous assignment

```
assign [#delay] <net name> = <expression>
```

❖ The continuous assignment is **always active**

- Any changes in the RHS of the continuous assignment are evaluated and the LHS is updated

❖ The LHS shall be

- **Scalar or vector net**
- Bit/part-select of vector net
- Concatenation of any of the above type

```
assign {cout, sum[3:0]} = a[3:0] + b[3:0] + cin;
```

❖ The RHS can be registers or nets or function call



Continuous Assignments (2/5)

❖ Syntax of continuous assignment

```
assign [#delay] <net name> = <expression>
```

❖ The LHS is updated at any change in the RHS expression after a specified delay

- The delay is optional
- The continuous assignment **with delay is non-synthesizable**
- Continuous assignments can only contain simple, left-hand side delay (i.e., limited to a **# delay**); because of their continuous nature, @ timing control is unnecessary

❖ Example

```
assign #10 o = in1 + in2
```



Continuous Assignments (3/5)

- ❖ Drive a value onto a **wire**, **wand**, **wor**, or **tri**
 - Use an explicit continuous assignment statement after the declaration

```
wire [7:0] o;  
assign o = in1 + in2;
```
 - Specify the continuous assignment statement in the same line as the declaration for a wire

```
wire [7:0] o = in1 + in2;
```
- ❖ Using continuous assignments for datapath descriptions
- ❖ Using them outside of a procedural block
- ❖ Using them to model **combinational circuits**



Continuous Assignments (4/5)

- ❖ The continuous assignment is **always active**

```
module assignment_1();
wire pwr_good, pwr_on, pwr_stable; reg Ok, Fire;

assign pwr_stable = Ok & (!Fire);
assign pwr_on = 1;
assign pwr_good = pwr_on & pwr_stable;

initial begin Ok = 0; Fire = 0; #1 Ok = 1; #5 Fire = 1; end
initial begin $monitor("TIME=%0d",$time," ON=",pwr_on, " STABLE=",
    pwr_stable," OK=",Ok," FIRE=",Fire," GOOD=",pwr_good);
#10 $finish; end
endmodule
```

```
> TIME=0 ON=1 STABLE=0 OK=0 FIRE=0 GOOD=0
> TIME=1 ON=1 STABLE=1 OK=1 FIRE=0 GOOD=1
> TIME=6 ON=1 STABLE=0 OK=1 FIRE=1 GOOD=0
```



Continuous Assignments (5/5)

- ❖ Continuous assignments provide a way to model combinational logic

continuous assignment

```
module inv_array(out,in);
    output [31:0] out;
    input [31:0] in;
    assign out=~in;
endmodule
```

gate-level modeling

```
module inv_array(out,in);
    output [31:0] out;
    input [31:0] in;
    not U1(out[0],in[0]);
    not U2(out[1],in[1]);
    ...
    not U31(out[31],in[31]);
endmodule
```



Assignment Delays

❖ Regular Assignment Delay

```
wire out;  
assign #10 out = in1 & in2;  
// delay in a continuous assign
```

❖ Implicit Continuous Assignment Delay

```
wire #10 out = in1 & in2;
```

❖ Net Declaration Delay

```
wire # 10 out; // net delays  
assign out = in1 & in2;
```

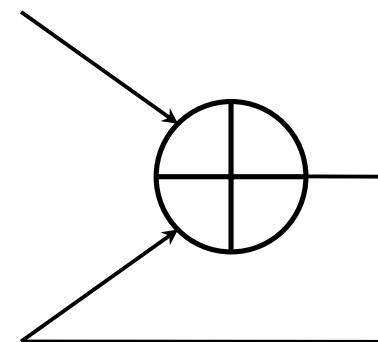
- Remind again, non-synthesizable! For Testbench/Behavior model usage



Avoiding Combinational Loops

- ❖ Avoid **combinational loops** (or logic loops)
 - HDL Compiler and Design Compiler will automatically open up asynchronous combinational loops
 - Without disabling the combinational feedback loop, the static timing analyzer can't resolve
 - Example

```
wire [3:0] a;  
wire [3:0] b;  
  
assign a = b + a;
```





Outline

- ❖ Definition of Register-Transfer Level (RTL)
- ❖ Signing and Sizing in Verilog
- ❖ Verilog Syntax of RTL
 - Operands
 - Operators
- ❖ Continuous Assignments for Combinational Circuits
- ❖ Signing and Sizing in Verilog



Sizing and Signing

- ❖ What are the results of the following expressions

```
reg [31:0] A, B, C, D, E;  
  
A = -12 / 3;                                // -4  
B = -'d12 / 3;                               // 1431655761  
C = -'sd12 / 3;                              // -4  
D = -'sd12 / $signed('d3);                  // -4  
E = -5'sd12 / 3'sd3;                         // -4
```

```
reg [7:0] o;  
reg [3:0] a, b;  
reg signed [3:0] c;  
  
a = 4'd15; b = 4'd8; c = -4'd1;  
o = (a + b) + c;                            // 38
```



Signing (1/3)

- ❖ Signals are “**regarded as**” signed in Verilog
 - In signed operations, the bit pattern is viewed as 2's complement
 - The bit pattern itself does not change
- ❖ The number without size and base format specified is a signed decimal
- ❖ The number with the base format should be viewed as unsigned unless the **s** designator is included

$$12 = (00\ldots001100)_2$$

$$-12 = (11\ldots110100)_2 = -12 \text{ (signed)}$$

$${}'\text{d}12 = (11\ldots110100)_2 = 4291967296 \text{ (unsigned)}$$

$${}'\text{s}\text{d}12 = (11\ldots110100)_2 = -12 \text{ (signed)}$$



Sign Conversion

- ❖ A built-in system functions to handle type casting
 - **`$signed()`**: Return value is signed
 - **`$unsigned()`**: Return value is unsigned
- ❖ Example

```
wire [7:0] a, b, c;  
assign a = $unsigned(-4)      // a = 8'b11111100  
assign b = $unsigned(-4'sd4)  // b = 8'b000001100  
assign c = $signed(4'b1100)   // c = 8'b11111100
```



Signed Signal (1/2)

- ❖ Signed addition and subtraction
 - Equivalent to unsigned signals
- ❖ Signed multiplication, division, and comparison

1. Using the **signed** declaration of regs and wires

```
wire signed [7:0] a;  
wire signed [7:0] b;  
wire cmp;  
assign cmp = a < b;
```

2. Or use **\$signed()** function to cast unsigned variables

```
wire [7:0] a, b;  
wire cmp;  
assign cmp = $signed(a) < $signed(b);
```

3. (constant) using the **s** designator

```
wire [7:0] a;  
wire cmp;  
assign cmp = $signed(a) < -8'sd2;
```



Signed Signal (2/2)

- ❖ Some expressions will convert the signed signal to unsigned
 - Bit-select results are unsigned
 - Part-select results are unsigned, even if the part-select specifies the entire vector
 - Concatenate results are unsigned
 - Comparison results (1, 0) are unsigned

```
reg signed [7:0] a, b;  
reg signed [16:0] c;  
  
c = a[7:0]; // {{8{1'b0}}, a}  
// b[7:0] is unsigned and therefore zero-extended  
  
c = {a, b}; // {1'b0, {a, b}}  
// {a, b} is unsigned and therefore zero-extended
```



Sizing and Signing Rules

- ❖ Verilog evaluating an expression by the following steps

- 1. Determine the sign of the right-hand side (RHS), then coerce all RHS to the result type**
 - If all operands are signed, the result is signed
 - If any operand is unsigned, the result is unsigned
 - The LHS should not be considered
- 2. Determine the expression size by choosing the largest operands size, including the size of the LHS**
- 3. Resize all RHS operands to the expression size**
 - Signed number is signed extension
 - Unsigned number is zero extension
- 4. Evaluate the RHS expression, producing a result of type and size found in step1-2**
- 5. Assign RHS value to LHS**



Sizing and Signing Examples (1/6)

```
wire [3:0] a;  
wire signed [3:0] b;  
wire signed [7:0] out;  
assign out = a + b;
```

```
> a = 5 (0101), b = -2 (1110)  
> a = 5 (0000_0101), b = -2 (0000_1110)  
> out = 19 (0001_0011)
```

- ❖ a is unsigned, b is signed \Rightarrow result is unsigned \Rightarrow **zero** extension

```
wire signed [3:0] a;  
wire signed [3:0] b;  
wire signed [7:0] out;  
assign out = a + b;
```

```
> a = 5 (0101), b = -2 (1110)  
> a = 5 (0000_0101), b = -2 (1111_1110)  
> out = 3 (0000_0011)
```

- ❖ a is signed, b is signed \Rightarrow result is signed \Rightarrow **sign** extension

```
wire signed [3:0] a;  
wire signed [3:0] b;  
wire [7:0] out;  
assign out = a + b;
```

```
> a = -3 (1101), b = -2 (1110)  
> a = -3 (1111_1101), b = -2 (1111_1110)  
> out = 251 (1111_1011)  
> $signed(out) = -5 (1111_1011)
```



Sizing and Signing Examples (2/6)

- ❖ Relational and equality operator
- ❖ **The result of part-select is unsigned**

```
reg signed [3:0] a;
reg [3:0] b;
reg signed [2:0] c;
reg cmp;

a = -1; b = 15; c = 7;

cmp = a > 3;                      // 0, signed
cmp = a > 'd3;                    // 1, unsigned 1111 > 0011
cmp = a > 4'sd3;                  // 0, signed
cmp = a[3:0] > 4'sd3;              // 1, unsigned 1111 > 0011
cmp = (a == b);                   // 1
cmp = (a == $signed(b));          // 1
cmp = (a == c);                   // 1, signed extension
```



Sizing and Signing Examples (3/6)

- ❖ Average of two number

```
reg [3:0] a, b, avg;           // a = 14, b = 10
reg [4:0] avg_good;

// will not work properly, avg = 4
avg = (a + b) >> 1;

// will work correctly, avg = 12
avg = (a + b + 0) >> 1;

// will work correctly, avg = 12
avg_good = (a + b) >> 1;
```



Sizing and Signing Examples (4/6)

- ❖ Using sign extension improperly

```
reg signed [15:0] o, o_bad;
reg signed [3:0] a, b;           // a = -4, b = 5

// will not work properly, o_bad = 1260
o_bad = { {4{a[3]}}, a } * { {4{b[3]}}, b };

// will work correctly, o = -20
o = a * b;
```



Sizing and Signing Examples (5/6)

❖ Signed 16-bit multiplier

```
wire [7:0] a, b;           // a = 100, b = -100
wire signed [15:0] c, c_bad;

// will not work properly, c_bad = -16
assign c_bad = $signed( $signed(a) * $signed(b) );

// will work correctly, c = -10000
assign c = $signed(a) * $signed(b);
```

- ❖ The expression in **\$signed()** should be evaluate first
 - The width of **\$signed(a) * \$signed(b)** in the **c_bad** expression is 8 bits, so this expression gives an incorrect result



Sizing and Signing Examples (6/6)

❖ Conditional operator

```
wire [7:0] a, b;                      // a = 100, b = -100
wire [15:0] c, d;                     // d = 66
wire condition;                      // condition = 1

// unsigned, c = 15600
assign c = (condition) ? $signed(a) * $signed(b) : d;

// signed, c = -10000
assign c = $signed(condition) ?
    $signed(a) * $signed(b) : $signed(d);

// signed, c = -10000
assign c = (condition) ? $signed(a) * $signed(b) : $signed(d);
```

Computer-Aided VLSI System Design

Chap.2-2 Logic Design at Behavioral Level

Graduate Institute of Electronics Engineering, National Taiwan University



NTU GIEE



Outline

- ❖ Behavioral Level Modeling and Event-Based Simulation
- ❖ Procedural Construct and Assignment
 - **initial** block
 - **always** block
 - procedural assignment
- ❖ Control Construct
- ❖ Functional Block
 - Sub-modules
 - **function** and **task**
 - **generate**
- ❖ Data Path Modeling
 - Data Path
 - Timing Parameters
- ❖ Finite State Machine (FSM)
 - Moore Machine & Mealy Machine
 - Behavior Modeling of FSM



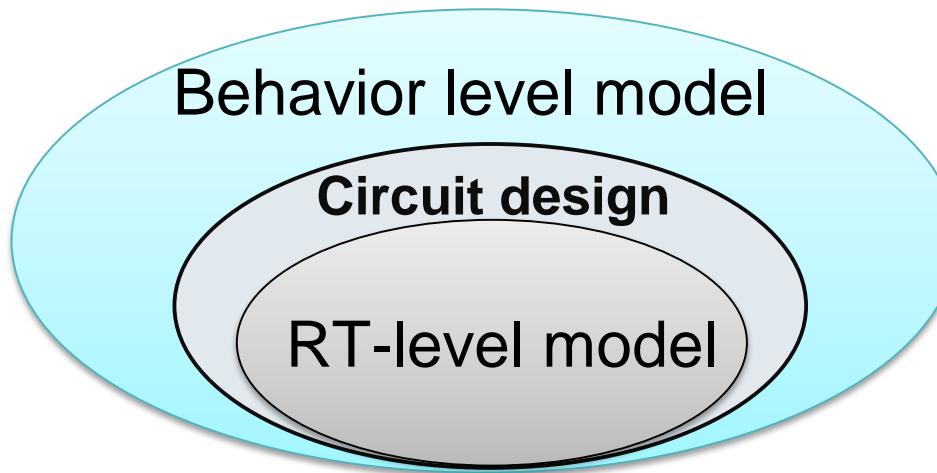
What is Behavioral Level

- ❖ Literally, modeling circuits using “behavioral” descriptions and events
- ❖ Description of an RTL model
 - Structure: constructs are separated for combinational and sequential circuits
 - Signal: **continuous evaluate-update**, pin accurate
 - Timing: cycle accurate
- ❖ Description of a behavioral level model
 - Structure: construct can be combinational, sequential, or **hybrid circuits**.
 - Signal: **event-driven**, timing, arithmetic (floating point or integer, ... to pin accurate)
 - Timing: untimed (ordered), approximate-timed (with delay notification), cycle accurate
- ❖ **Note: Only a small part of the behavioral level syntax is used for describing circuits, others are widely used for verification (testbench)** **(Behavioral Modeling != non-synthesizable)**



Relationship between RTL and Behavioral Level

- ❖ Behavior level model
 - Hardware modeling
 - Implicit structure description for modeling
 - Flexible



- ❖ RTL level model
 - ❖ Hardware modeling
 - ❖ Explicit structure description for modeling
 - ❖ Accurate



Event

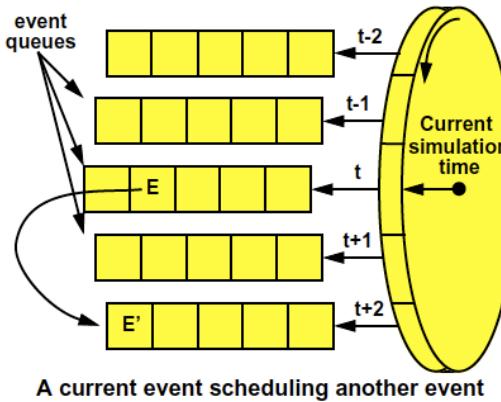
- ❖ A data type has state and timing description
 - state: **level change** ($1 \rightarrow 0$, $0 \rightarrow 1$, $1 \rightarrow X$, $0 \rightarrow Z$, etc.), **edge** (defined logic-level transition: e.g. $0 \rightarrow 1$)
 - timing: virtual time in simulator
 - example: signal s changed from 0 to 1 at 3ns





Event-Based Simulation

- ❖ Execute statement (evaluate expression and update variable) when defined event occurs
 - input transition cause an event on circuit
 - simulation is on the OR-ed occurrence of sensitive events
- ❖ Benefits
 - Accelerate simulation speed: Only evaluate expression when variables on RHS change
 - Allow high-level description (behavior) of a constructor





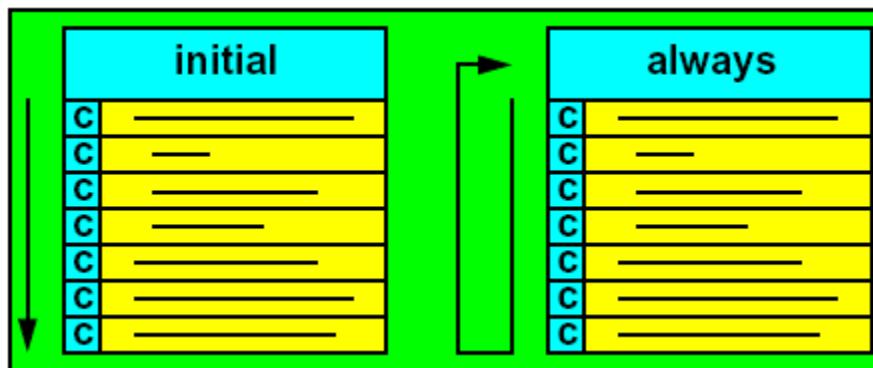
Outline

- ❖ Behavioral Level Modeling and Event-Based Simulation
- ❖ Procedural Construct and Assignment
 - **initial** block
 - **always** block
 - procedural assignment
- ❖ Control Construct
- ❖ Functional Block
 - Sub-modules
 - **function** and **task**
 - **generate**
- ❖ Data Path Modeling
 - Data Path
 - Timing Parameters
- ❖ Finite State Machine (FSM)
 - Moore Machine & Mealy Machine
 - Behavior Modeling of FSM



Procedural Blocks (1/3)

- ❖ There are two kinds of procedural block
 - **initial**: execute only once for initialization or waveform generation (**NON-SYNTHESIZABLE**) → There's no circuit can work only once then disappear
 - **always**: execute in a loop
- ❖ The **initial** and **always** procedural blocks are enabled at the beginning of a simulation
- ❖ All procedural blocks execute **concurrently**





Procedural Blocks (2/3)

- ❖ Procedural blocks contain the following components
- ❖ **Procedural assignment**
 - Describe the dataflow within the block
- ❖ **High-level constructs**
 - Describe the functional operation of the block
- ❖ **Timing controls**
 - To trigger the block and control the execution of the statements in the blocks

The diagram illustrates a procedural block with the following code:

```
always @(*) begin
    if (a > 0)
        c = #10 a - b;
    else
        c = #5 a;
end
```

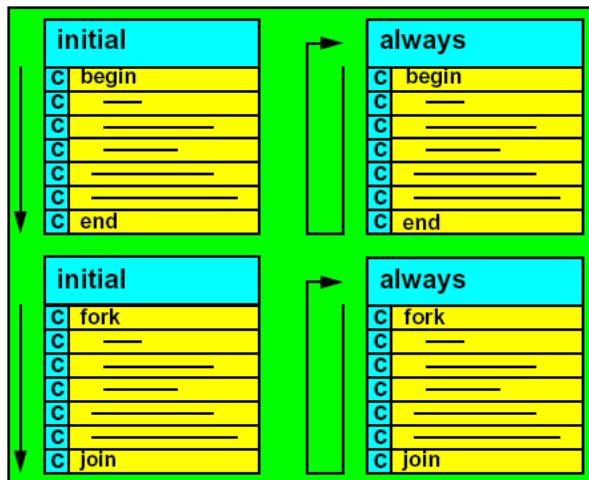
Annotations highlight specific parts of the code:

- High-level construct**: Points to the `if` and `else` keywords.
- Timing control**: Points to the `always @(*)` and `#10` timing specifications.
- Procedural assignment**: Points to the assignment statements `c = #10 a - b;` and `c = #5 a;`.



Procedural Blocks (3/3)

- ❖ Sequential block: **begin** and **end** group sequential procedural statements
- ❖ Concurrent block: **fork** and **join** group concurrent procedural statements (**NON-SYNTHESIZABLE**, not for modeling circuits)
- ❖ If the procedural block contains only one assignment, then both **begin-end** and **for-join** can be omitted
- ❖ Two types of blocks differ only if there's timing control inside



Two blocks do the same operations!

begin	fork
#5 a = 1;	#5 a = 1;
#5 a = 2;	#15 a = 3;
#5 a = 3;	#10 a = 2;
end	join



Example

```
module assignment_test;
    reg [3:0] r1,r2;
    reg [4:0] sum2;
    reg [4:0] sum1;

    always @(r1 or r2)
        sum1 = r1 + r2;

    initial
    begin
        r1 = 4'b0010; r2=4'b1001;
        sum2 = r1+r2;
        $display(" r1      r2      sum1      sum2");
        $monitor(r1, r2, sum1, sum2);

        #10 r1 = 4'b0011;
    end

endmodule
```

Result

r1	r2	sum1	sum2
0010	1001	01011	01011
0011	1001	01100	01011



Timing Control

- ❖ Procedural timing controls can be achieved by the following three methods
 1. A delay-based timing control
 - # (delay)
 2. An event-based timing control
 - @ (sensitivity list)
 3. A level sensitive timing control
 - wait(<expression>)



Delay-Based

- ❖ Using simple delay to delay stimulus in testbench or to approximate real-world delays in behavioral models
- ❖ Simple delay (# delay) is **NON-SYNTHESIZABLE**
 - because it is difficult to design a circuit that has constant delay under any environment (temperature, transistor... variant)
- ❖ Example: clock generation in the testbench

```
reg clk;  
parameter CYCLE = 10;  
  
initial clk = 0;  
always # (CYCLE/2) clk = ~clk;
```



Event-Based (1/2)

❖ Edge-sensitive control (@)

- The sensitivity list is described after “**always @**”
- This means if any signals inside change (have an edge in waveform), the **always** block is triggered.
- Keywords **or** are used to separate multiple signals
- Keywords **posedge** & **negedge** is used when the **always** block should be triggered by positive or negative edge transition of the signal
- Missing signals in sensitivity list may lead to wrong results!

```
always@ ( a or b or cin)
begin
  {cout, sum} = a + b + cin;
end
```

WRONG!

```
// initial a=0, b=0, x=0, y=1
always@(a or b) begin // 1. b changes to 1
  y = ~x;           // 2. y remains 1
  x = a | b;       // 3. x changes to 1
end
```

CORRECT!

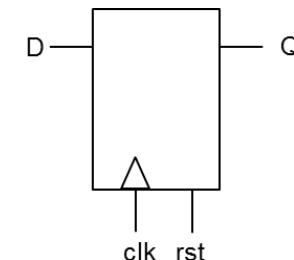
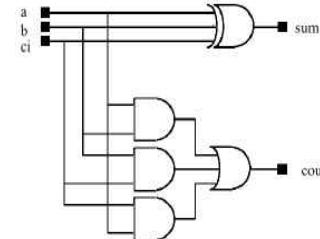
```
// initial a=0, b=0, x=0, y=1
always@(a or b or x) begin // 1. b changes to 1
  y = ~x;           // 2. y remains 1
                                // 4. y changes to 0
  x = a | b;       // 3. x changes to 1,
                    trigger again!
                                // 5. x remains 1
end
```



Event-Based (2/2)

- ❖ Easy way to use sensitivity list
 - For combinational circuit
 - Pure logic circuit
 - Use always @ (*)
 - For sequential circuit
 - D-FF circuit
 - Edge trigger of clock or reset signal

```
always @(*) begin
  {cout, sum} = ci + a + b;
end
```



combinational circuit

```
always @(posedge clk or posedge rst)
begin
  if (rst) Q <= 0;
  else      Q <= D;
end
```

sequential circuit



Level Sensitive Timing Control

- ❖ Using **wait** for level-sensitive timing control

wait(<expression>)

- ❖ Wait until the expression is asserted
- ❖ **wait** is **NON-SYNTHESIZABLE**
- ❖ Example

```
module D_latch (D, Q, enable);
    input D, enable;
    output Q;
    always @ (D) begin
        wait(enable)
        Q = D;
        // change of Q will be ignored
        // when it is still waiting
    end
endmodule
```

```
module testbench;
    initial # (TIME_OUT) $finish;

    initial begin
        wait(o_finish)
        // check output data here
        #10
        $finish;
    end
endmodule
```



Procedural Assignment

- ❖ Assignments inside procedural blocks are called procedural assignment
- ❖ The left-hand side (LHS) of a procedural assignment
 - register data type (e.g. *reg*)
- ❖ The data type of RHS signals is not restricted
- ❖ The variable without declared is default as *wire*, which may cause errors in the procedural block
- ❖ Two types of procedural assignment statements
 - **Blocking assignment (=)**
 - **Non-blocking assignment (<=)**



Blocking & Non-blocking Assignments

- ❖ There are two types of procedural assignment statements: **blocking (=)** and **non-blocking (<=)**
- ❖ Blocking assignment (=)
 - Evaluate the RHS and pass to the LHS
 - Suitable for model or design the **combinational** circuit
 - **Difficult to model the concurrency**
- ❖ Non-blocking assignment (<=)
 - Evaluate the RHS, but schedule the LHS
 - Update LHS only after evaluate all RHS
 - Greatly simplify modeling concurrency



Blocking or Non-Blocking?

❖ Blocking assignment

- Evaluation and assignment are immediate

```
always @ (a or b or c)
begin
    x = a | b;           1. Evaluate a | b, assign result to x
    y = a ^ b ^ c;       2. Evaluate a^b^c, assign result to y
    z = b & ~c;          3. Evaluate b&(~c), assign result to z
end
```

❖ Nonblocking assignment

- All assignment deferred until all right-hand sides have been evaluated (end of the virtual timestamp)

```
always @ (a or b or c)
begin
    x <= a | b;         1. Evaluate a | b but defer assignment of x
    y <= a ^ b ^ c;     2. Evaluate a^b^c but defer assignment of y
    z <= b & ~c;         3. Evaluate b&(~c) but defer assignment of z
end                                4. Assign x, y, and z with their new values
```



Outline

- ❖ Behavioral Level Modeling and Event-Based Simulation
- ❖ Procedural Construct and Assignment
 - **initial** block
 - **always** block
 - procedural assignment
- ❖ Control Construct
- ❖ Functional Block
 - Sub-modules
 - **function** and **task**
 - **generate**
- ❖ Data Path Modeling
 - Data Path
 - Timing Parameters
- ❖ Finite State Machine (FSM)
 - Moore Machine & Mealy Machine
 - Behavior Modeling of FSM



Conditional Statements (1/2)

❖ If and If-else statements

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

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

```
if (sel == 0)
begin
    a_w = data;
    b_w = b_r;
end
else begin
    b_w = data;
    a_w = a_r;
end
```

```
a_w = a_r;
b_w = b_r;

if (sel == 0)
    a_w = data;
else
    b_w = data;
```

```
if (sel == 0)
    a <= data;
else
    b <= data;
```



❖ Restrictions compared with C

- LHS in all cases should be **the same!**
 - Avoid latch
- Conditions should be **full-case**, if must be followed by else!
 - Avoid latch
- In short, think about **MUX**!

```
if (sel == 0)
    a_w = data;
else
    b_w = data;
```



```
if (sel == 0)
    a_w = data;
```





Conditional Statements (2/2)

Example 1

```
if (sel==3)
    y = d;
else
    if (sel==2)
        y = c;
    else
        if (sel==1)
            y = b;
        else
            if (sel==0)
                y = a;
```

VS.

Example 2

```
if (sel[1])
    if (sel[0])
        y = d;
    else
        y = c;
else
    if (sel[0])
        y = b;
    else
        y = a;
```

VS.

Example 3

```
if (sel==3)
    y = d;
else if (sel==2)
    y = c;
else if (sel==1)
    y = b;
else
    y = a;
```



Multiway Branching

- The nested ***if-else-if*** can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the ***case*** statement

```
case (expression)
  alternative1: statement1;
  alternative2: statement2;
  alternative3: statement3;
  ...
  default: default_statement;
endcase
```

Alternative way:

```
always @(*) begin
  y = a;

  case(sel)
    3: y = d;
    2: y = c;
    1: y = b;
  endcase
end
```

NOTICE:
Not full case would cause latch

if-else statement

```
if (sel==3)
  y = d;
else
  if (sel==2)
    y = c;
  else
    if (sel==1)
      y = b;
    else
      y = a;
```

case statement

```
case (sel)
  3: y = d;
  2: y = c;
  1: y = b;
default: y = a;
endcase
```



Looping Statements

- ❖ The for loop (conditionally synthesizable)
- ❖ The while loop (X)
- ❖ The repeat loop (X)
- ❖ The forever loop (X)



For Loop

- ❖ The keyword **for** is used to specify this loop. The **for** loop contain 3 parts:
 - An initial condition
 - A check to see if the terminating condition is true
 - A procedural assignment to change value of the control variable
- ❖ **Synthesizable only if the expanded form is synthesizable**
 - 沒有時間相依性
 - 不能和參數有關係

array[i+1] <= array[i] - 1;
array[i] = 2*i;



```
module bitwise_and(a, b, out);
  parameter size=2;
  input [size-1:0] a, b;
  output [size-1:0] out;
  reg [size-1:0] out;
  integer i;
  always @(a or b)
    begin
      for (i = 0;i < size; i = i + 1)
        out[i] = a[i] & b[i];
    end
  endmodule
```



While Loop

- ❖ The **while** loop executes until the **while**-expression becomes false
- ❖ NON-SYNTHESIZABLE

```
initial
begin
    reg [7:0] tempreg;
    count = 0;
    tempreg = reg;
    while (tempreg)
        begin
            if (tempreg[0]) count = count + 1;
            tempreg = tempreg >> 1;
        end
    rega = 101;
    tempreg      count
    101          1
    010          1
    001          2
```



Repeat Loop

- ❖ The keyword **repeat** is used for this loop. The **repeat** construct executes the loop a **fixed** number of times.

```
module multiplier(result, op_a, op_b);
    ...
    reg shift_opa, shift_opb;
    parameter size = 8;
    initial begin
        result = 0; shift_opa = op_a; shift_opb = op_b;
        repeat (size)
            begin
                #10 if (shift_opb[1])
                    result = result + shift_opa;
                    shift_opa = shift_opa << 1;
                    shift_opb = shift_opb >> 1;
            end
        end
    endmodule
```

default repeat
8 times



Forever Loop

- ❖ The keyword **forever** is used to express the loop. The loop does not contain any expression and executes forever until the **\$finish** task is encountered

```
//Clock generation          //Synchronize 2 register values
//Clock with period of 20 units) //at every positive edge of clock
reg clk;
reg clk;
reg x,y;

initial
begin
    clk=1'b0;
    forever #10 clk=~clk;
end

initial
forever @ (posedge clk) x=y;
```



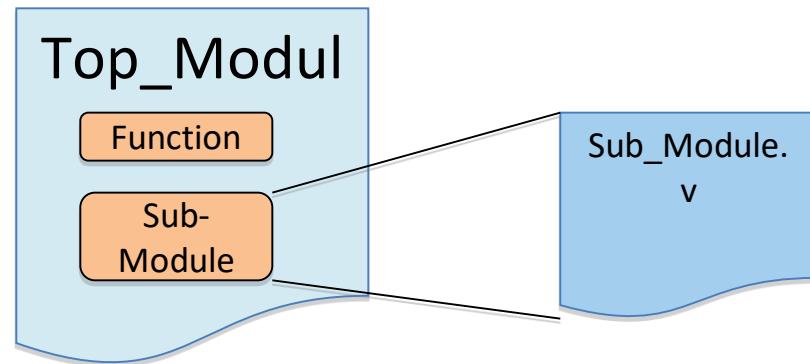
Outline

- ❖ Behavioral Level Modeling and Event-Based Simulation
- ❖ Procedural Construct and Assignment
 - **initial** block
 - **always** block
 - procedural assignment
- ❖ Control Construct
- ❖ Functional Block
 - Sub-modules
 - **function** and **task**
 - **generate**
- ❖ Data Path Modeling
 - Data Path
 - Timing Parameters
- ❖ Finite State Machine (FSM)
 - Moore Machine & Mealy Machine
 - Behavior Modeling of FSM



Functional Partition

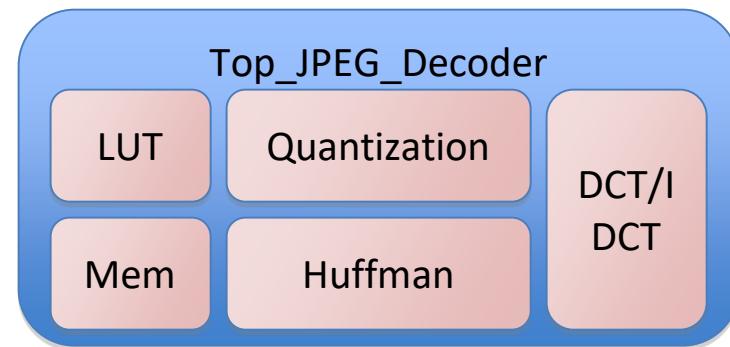
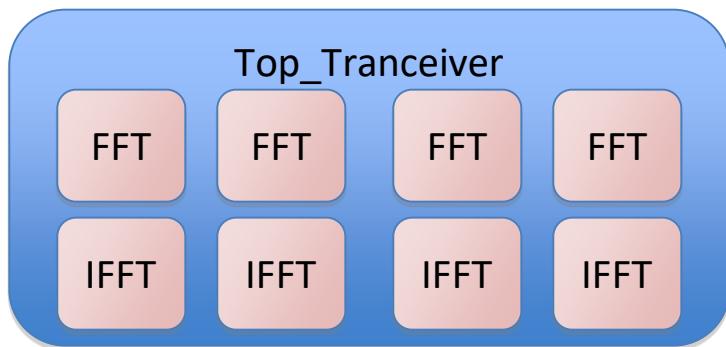
- ❖ A digital system may consist of many function blocks
 - A lumped Verilog module makes debugging & editing a great disaster
- ❖ Break the whole system into several function blocks
 - Sub-module
 - Function/Task





Sub-Module (1/2)

- ❖ When using a sub-module
 - Function block with many duplications
 - Function block containing specific computation



- ❖ Note: abusing sub-modules may also makes your design hard to read



Sub-Module (2/2)

- ❖ Instantiate a sub-module



Top-module

```
reg clock, reset;  
wire [2:0] counter;  
  
Counter8 u_cnt1(  
    .clk(clock),  
    .rst(reset),  
    .out(counter)  
)
```

Sub-module

```
module Counter8(  
    clk,  
    rst,  
    out  
)  
...  
endmodule
```



Function (1/3)

- ❖ Functions are declared **inside a module** with the keywords **function** and **endfunction**.
- ❖ A functions may be called from within procedural and continuous assignment statements
- ❖ It is typically used to perform a computation, conversions, or to represent combinational logic
 - No timing control
 - Only input arguments and exactly one return signal

```
function [ WIDTH-1: 0] my_function;  
<I/O ports declaration> // only input ports  
<local variables declaration> // register type  
begin  
    <function_statements>  
end  
endfunction
```



Function (2/3)

- ❖ Function name is regarded as the **output reg** of function
 - When a function is declared, a register with name (name of function) is declared implicitly inside.
 - Thus, the function cannot have more than 1 output.
- ❖ Functions **cannot invoke other tasks**. They can **only invoke other functions**



Function (3/3)

❖ Function example

```
// function definition
function [7:0] abs;          //unsigned
    input [7:0] number_in;   //signed
    begin
        abs = (number_in[7])?
                (~number_in+1'b1): number_in;
    end
endfunction

// call function
reg [7:0] var1, abs_var1;
always@(var1) begin
    abs_var1 = abs(var1);
end
```



Task (1/6)

- ❖ Task are declared **inside a module** with the keyword **task** and **endtask**
- ❖ Task take the control of the block, and return the control back to the block when the task is finish or disabled
- ❖ It can't be called from a **continuous assignment**
- ❖ It is typically used to perform debugging operations
 - Could contain timing control

```
task my_task;  
<I/O ports declaration> // input, output, inout ports  
<local variables declaration> // register type  
begin  
    <task_statements> // performs the work of the task  
end  
endtask
```



Task (2/6)

- ❖ Task arguments shall be passed follow the order of their declaration
- ❖ The output shall be register type

```
task my_task;  
    input a, b;  
    output c;  
    begin  
        # 10  
        c = a & b;  
    end  
endtask  
  
reg a, b;  
reg c;                      // wire c; => error  
initial begin  
    my_task(a, b, c);      // follow the declaration order  
end
```



Task (3/6)

- ❖ Task arguments are evaluated when the task is invoked (pass by value)
- ❖ Signals used in timing controls (such as clk) must not be inputs to the task, because input values are passed into the task only **once**

```
// task definition
task light;
output color;
input [31:0] tics;
begin
    repeat (tics) @ (negedge clk);
        color = off;
    end
endtask
// call task
always begin
    green = on;
    light(green, green_tics);
end
```



Task (4/6)

❖ Task example

```
module sequence;  
...  
reg clock;  
...  
initial  
    init_sequence;  
...  
always begin  
    asymmetric_sequence;  
end  
...
```

```
task init_sequence;  
begin  
    clock = 1'b0;  
end  
endtask  
  
task asymmetric_sequence;  
begin  
    #12 clock = 1'b0;  
    #5  clock = 1'b1;  
    #3  clock = 1'b0;  
    #10 clock = 1'b1;  
end  
endtask  
...  
endmodule
```

Directly copy & paste to
where you use it



Task (5/6)

```
task data_mon;
    input [7:0] a, b;
begin
    #20
    $display("@%d, a = %d, b = %d", $time, a, b);
    #10
    $display("@%d, task finish!", $time);
end
endtask

reg [7:0] a, b;

initial begin
    a = 1; b = 2;
    data_mon(a, b);
    #10
    a = 10; b = 20;
    #30
    $finish;
end
```

```
// with input declaration
> @ 20, a = 1, b = 2
> @ 30, task finish!
> @ 70, sim finish!
```



Task (5/6)

```
task data_mon;
// input [7:0] a, b;
begin
    #20
    $display("@%d, a = %d, b = %d", $time, a, b);
    #10
    $display("@%d, task finish!", $time);
end
endtask

reg [7:0] a, b;

initial begin
    a = 1; b = 2;
    data_mon;
    #10
    a = 10; b = 20;
    #30
    $finish;
end
```

// without input declaration
> @ 20, a = 1, b = 2
> @ 30, task finish!
> @ 70, sim finish!



Task (5/6)

```
task data_mon;
// input [7:0] a, b;
begin
    #20
    $display("@%d, a = %d, b = %d", $time, a, b);
    #10
    $display("@%d, task finish!", $time);
end
endtask

reg [7:0] a, b;

initial begin
    a = 1; b = 2;
    #10
    a = 10; b = 20;
    #30
    $finish;
end

initial data_mon;
```

```
// without input declaration
// concurrent
> @ 20, a = 10, b = 20
> @ 30, task finish!
> @ 40, sim finish!
```



Task (6/6)

- ❖ Use **disable** to cancel the task

```
task errmon;
    forever@(posedge data_ready) begin
        if (golden!=data)
            $display("ERR:data=%b,expected=%b",data,golden);
            $finish;
    end
endtask
initial begin
    fork
        errmon;
        begin
            runtest;
            disable errmon;
        end
    join
end
```



Comparison: Function & Task

Function	Task
1. Is typically used to perform a computation, or to represent combinational logic 2. Called in procedural statement or continuous assignment	1. Is typically used to perform debugging operations, or to behaviorally describe hardware 2. Called in procedural statement
Can enable other functions	Can enable other tasks and functions
must not contain any delay , event, or timing control statements (execute in 0 simulation time)	may contain delay, event, or timing control statements
must have at least one input argument .	may have zero or more arguments of type input, output or inout
Always return a single value. They cannot have output or inout arguments	Do not return with a value but can pass multiple values through output and inout arguments



Generate

- ❖ Using ***generate*** to either conditionally or multiply instantiate generate blocks
- ❖ Can't contain the following things in generate block
 - port declarations
 - parameter declarations
- ❖ **Loop generate constructs**
 - Generate blocks multiple times
- ❖ **Conditional generate constructs**
 - if-generate constructs
 - case-generate constructs



Loop generate constructs

❖ Example: array architecture

```
parameter LEN = 4;
genvar i;
reg [7:0] a [0:LEN-1];
reg [7:0] b [0:LEN-1];
reg [7:0] c [0:LEN];
generate
  for (i = 0; i < LEN; i = i+1) begin : array
    M1 u1(i_clk, a[i], b[i], c[i], c[i+1]); // scope array[i].u1
  end
endgenerate
```

```
module M1 (clk, a, b, ci, co)
input clk;
input [7:0] a, b, ci;
output [7:0] co;
reg [7:0] co_r, co_w;
assign co = co_r;

always @(*) begin
  co_w = ci + a*b;
end
always @(posedge clk) begin
  co_r <= co_w;
end
endmodule
```



Conditional generate constructs

❖ Example

```
generate
    if((a_width < 8) || (b_width < 8)) begin: mult
        Low_Power_Module #(a_width,b_width) u1(a, b, out);
    end
    else begin: mult
        High_Performance_Module #(a_width,b_width) u1(a, b, out);
    end
endgenerate
```



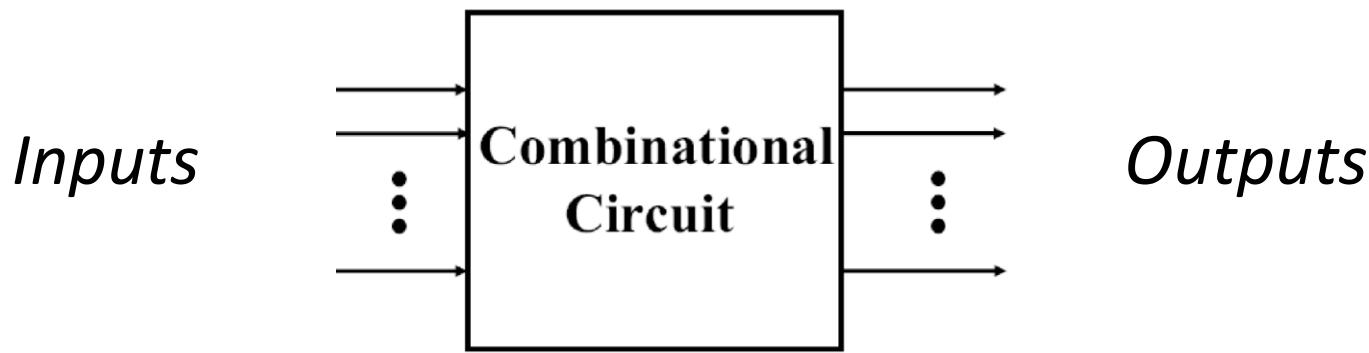
Outline

- ❖ Behavioral Level Modeling and Event-Based Simulation
- ❖ Procedural Construct and Assignment
 - **initial** block
 - **always** block
 - procedural assignment
- ❖ Control Construct
- ❖ Functional Block
 - Sub-modules
 - **function** and **task**
 - **generate**
- ❖ Data Path Modeling
 - Data Path
 - Timing Parameters
- ❖ Finite State Machine (FSM)
 - Moore Machine & Mealy Machine
 - Behavior Modeling of FSM



Combinational Data Path

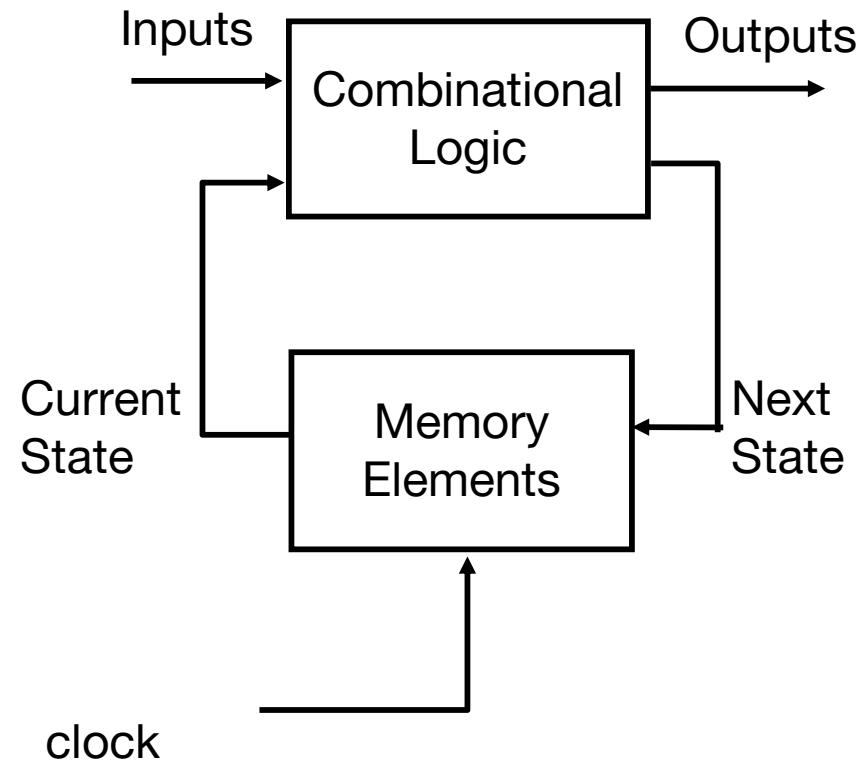
- ❖ Combinational logic circuits are **memoryless**
 - Any transition in inputs affect the whole circuit right away
 - Feedback path is not allowed
- ❖ Output can have **logical transitions** before settling to a stable value





Sequential Data Path

- ❖ Sequential circuits have memory
(i.e. remember the past)
- ❖ The output is
 - depend on inputs
 - depend on current state
- ❖ In synchronous system
 - clock orchestrates the sequence of events
- ❖ Fundamental components
 - Combinational circuits
 - Memory elements





Modeling of Flip-Flops (1/2)

- ❖ The use of **posedge** and **negedge** makes an **always** block sequential (edge-triggered)
- ❖ Unlike combinational always block, the sensitivity list does determine the behavior of synthesis

*D Flip-flop with **synchronous** clear*

```
moduledff_sync_clear(d, clearb,
clock, q);
input d, clearb, clock;
output q;
reg q;
always @ (posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

always block entered only at
each positive clock edge

*D Flip-flop with **asynchronous** clear*

```
moduledff_async_clear(d, clearb, clock, q);
input d, clearb, clock;
output q;
reg q;
always @ (negedge clearb or posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

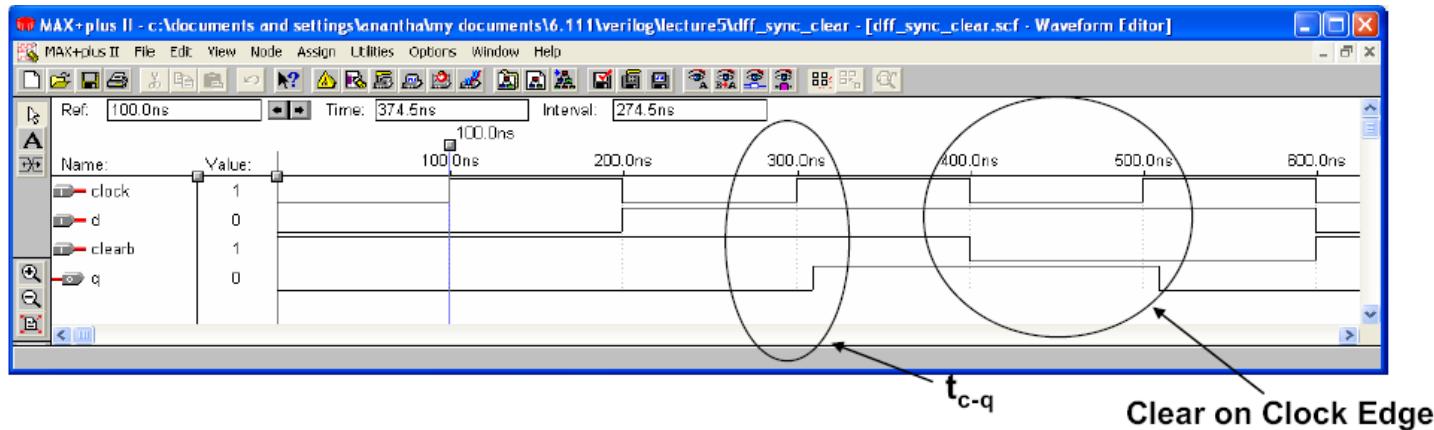
always block entered immediately
when (active-low) clearb is asserted

Note: The following is **incorrect** syntax: `always @ (clear or negedge clock)`
If one signal in the sensitivity list uses posedge/negedge, then all signals must.

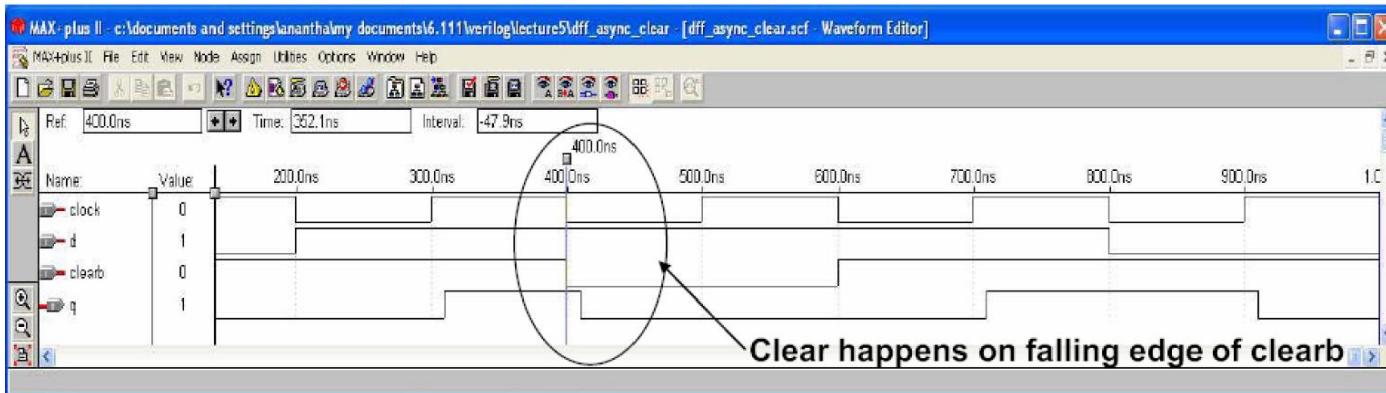


Modeling of Flip-Flops (2/2)

❖ Synchronous Reset



❖ Asynchronous Reset





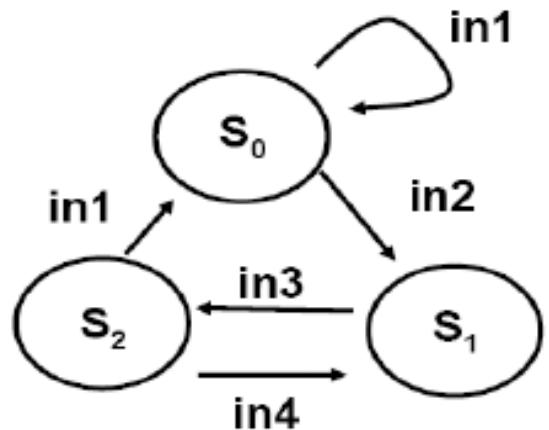
Outline

- ❖ Behavioral Level Modeling and Event-Based Simulation
- ❖ Procedural Construct and Assignment
 - **initial** block
 - **always** block
 - procedural assignment
- ❖ Control Construct
- ❖ Functional Block
 - Sub-modules
 - **function** and **task**
 - **generate**
- ❖ Data Path Modeling
 - Data Path
 - Timing Parameters
- ❖ Finite State Machine (FSM)
 - Moore Machine & Mealy Machine
 - Behavior Modeling of FSM



Finite State Machine (FSM)

- ❖ Model of computation consisting of
 - A set (of finite number) of states
 - An initial state
 - Input symbols
 - Transition function that maps input symbols and current states to a next state.



State transition diagram



Elements of FSM

❖ Memory Elements

- Memorize Current States (CS)
- Usually consist of FF or latch
- N-bit FF have 2^n possible states

❖ Next-state Logic (NL)

- Combinational Logic
- Produce next state
- Based on current state (CS) and input (X)

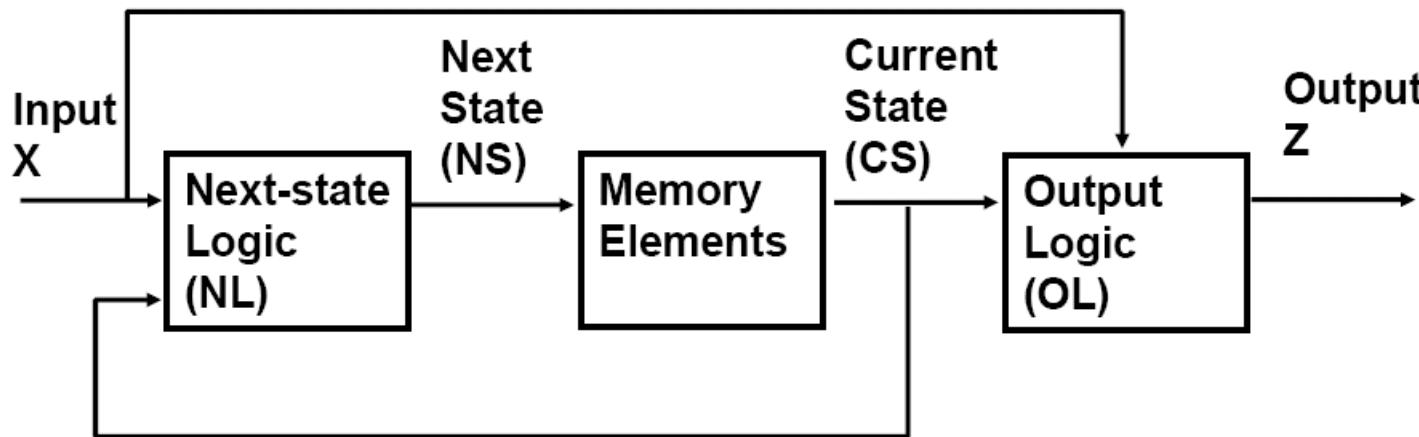
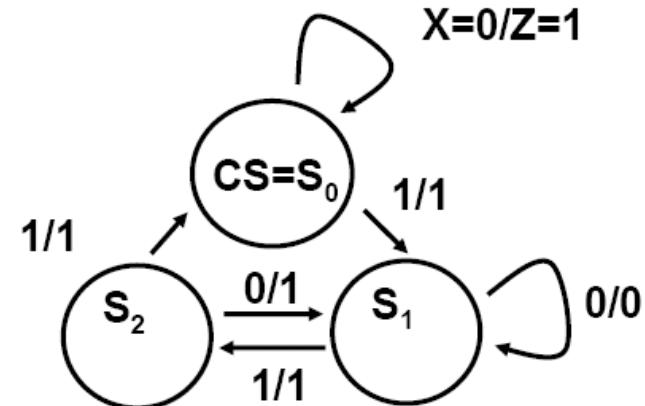
❖ Output Logic (OL)

- Combinational Logic
- Produce outputs (Z)
 - Based on current state
 - Based on current state and input



Mealy Machine

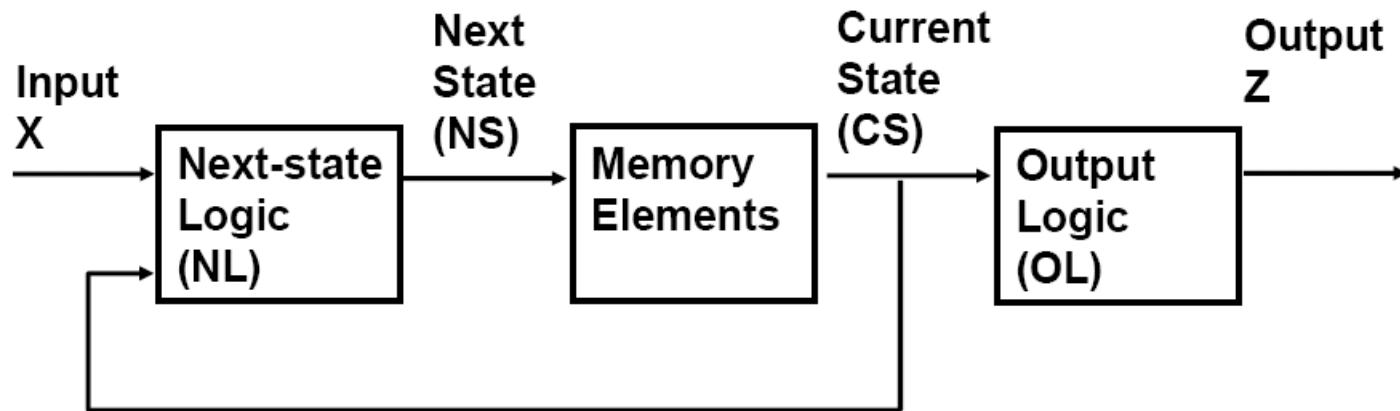
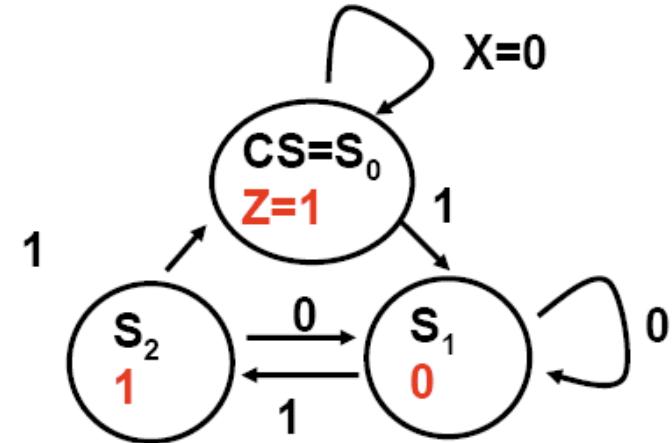
- ❖ Output is a function of
 - both current state & input





Moore Machine

- ❖ Output is a function of
 - only current state

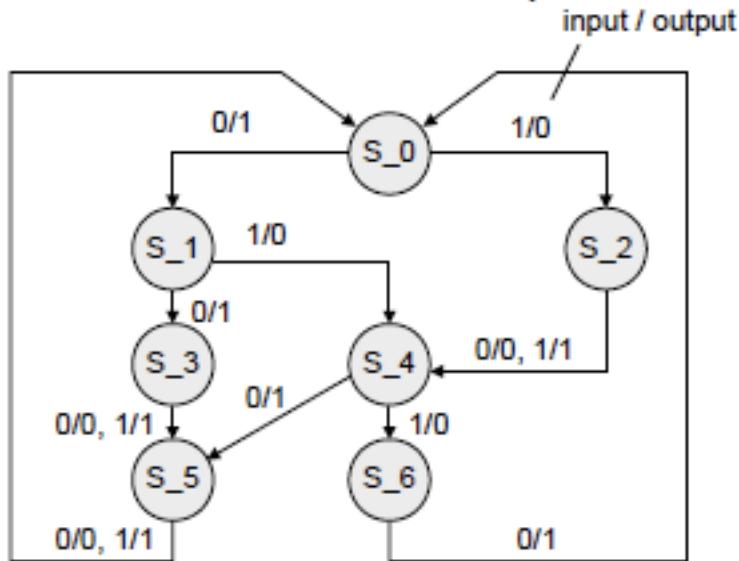




Manual Design of FSM (1/2)

- ❖ State Transition Graph describe the mechanism of FSM
- ❖ Then encode each state and derive the encoded table

State Transition Graph



Encoded Next state/ Output Table				
	state	next state		output
	$q_2 q_1 q_0$	$q_2^+ q_1^+ q_0^+$		
		input	input	
		0	1	0
S_0	000	001	101	1 0
S_1	001	111	011	1 0
S_2	101	011	011	0 1
S_3	111	110	110	0 1
S_4	011	110	010	1 0
S_5	110	000	000	
S_6	010	000	-	
	100	-	-	

Now we use
CAD tool to
do this!!!



Manual Design of FSM (2/2)

- ❖ Use the K-Map to derive the gate-level FSM design

q_2	q_1	B_n	00	01	11	10
00	00	1	1	1	1	1
01	01	0	0	0	0	0
11	11	0	0	0	0	0
10	10	x	x	1	1	1

$$q_0^+ = q_1'$$

q_2	q_1	B_n	00	01	11	10
00	00	1	1	0	1	1
01	01	0	0	0	1	1
11	11	0	0	1	1	0
10	10	x	x	0	0	0

$$q_2^+ = q_1'q_0'B_{in} + q_2'q_0B_{in}' + q_2q_1q_0$$

$$B_{out} = q_2'B_{in}' + q_2B_{in}$$

q_2	q_1	B_{in}	00	01	11	10
00	00	0	0	1	1	1
01	01	0	0	0	1	1
11	11	0	1	0	1	0
10	10	x	x	1	1	0

$$q_1^+ = q_0$$

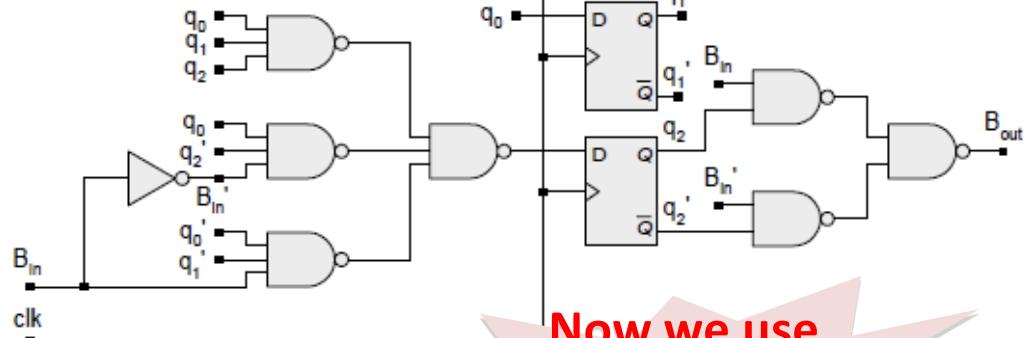
q_2	q_1	B_{in}	00	01	11	10
00	00	1	1	0	1	1
01	01	1	0	0	1	1
11	11	0	1	1	1	0
10	10	x	1	0	0	0

$$q_2^+ = q_1'q_0'B_{in} + q_2'q_0B_{in}' + q_2q_1q_0$$

$$\overline{q_2^+} = \overline{q_1'q_0'B_{in}} + \overline{q_2'q_0B_{in}'} + \overline{q_2q_1q_0}$$

$$\overline{q_2^+} = \frac{\overline{q_1'q_0'B_{in}}}{\overline{q_1'q_0'B_{in}}} \cdot \frac{\overline{q_2'q_0B_{in}'}}{\overline{q_2'q_0B_{in}'}} \cdot \frac{\overline{q_2q_1q_0}}{\overline{q_2q_1q_0}}$$

$$q_2^+ = \frac{q_1'q_0'B_{in}}{q_1'q_0'B_{in}} \cdot \frac{q_2'q_0B_{in}'}{q_2'q_0B_{in}'} \cdot \frac{q_2q_1q_0}{q_2q_1q_0}$$



Now we use
CAD tool to
do this!!!



Behavior Modeling of FSM

❖ Combinational Part

- Next-state logic (NL)
- Output logic (OL)

❖ Sequential Part

- Current state (CS) stored in flip-flops

❖ 3 Coding Style

- Separate CS, OL and NL
- Combine NL+ OL, separate CS
- Combine CS + NL, separate OL



Coding Style 1:

Separate CS, OL and NL

❖ CS

```
always @ (posedge clk)
    current_state <= next_state;
```

❖ NL

```
always @ (current_state or In)
    case (current_state)
        S0: case (In)
            In0: next_state = S1;
            In1: next_state = S0;
            .
            .
            endcase //In
        S1: . . .
        S2: . . .
    endcase //current_state
```

❖ OL

```
// if Moore
always @ (current_state)
    z = output_value;
```

```
// if Mealy
always @ (current_state or In)
    z = output_value;
```



Coding Style 2:

Combine NL+ OL, Separate CS

❖ CS

```
always @ (posedge clk)
    current_state <= next_state;
```

❖ NL+OL

```
always @ (current_state or In)
    case (current_state)
        S0: begin
            case (In)
                In0: begin
                    next_state = S1;
                    Z =values; // Mealy
                end
                In1: . .
            endcase // In
            Z =values; // Moore
        end //S0
        S1: . .
    endcase // current_state
```



Coding Style 3:

Combine CS+NL, Separate OL

❖ CS+NL

```
always @ (posedge clk)
begin
    case (state)
        S0: case (In)
            In0: state<= S1;
            In1: state<= S0;
            ...
        endcase //In
        S1: ...
    endcase //state
end
```

Do not mix comb and seq in one always block

❖ OL

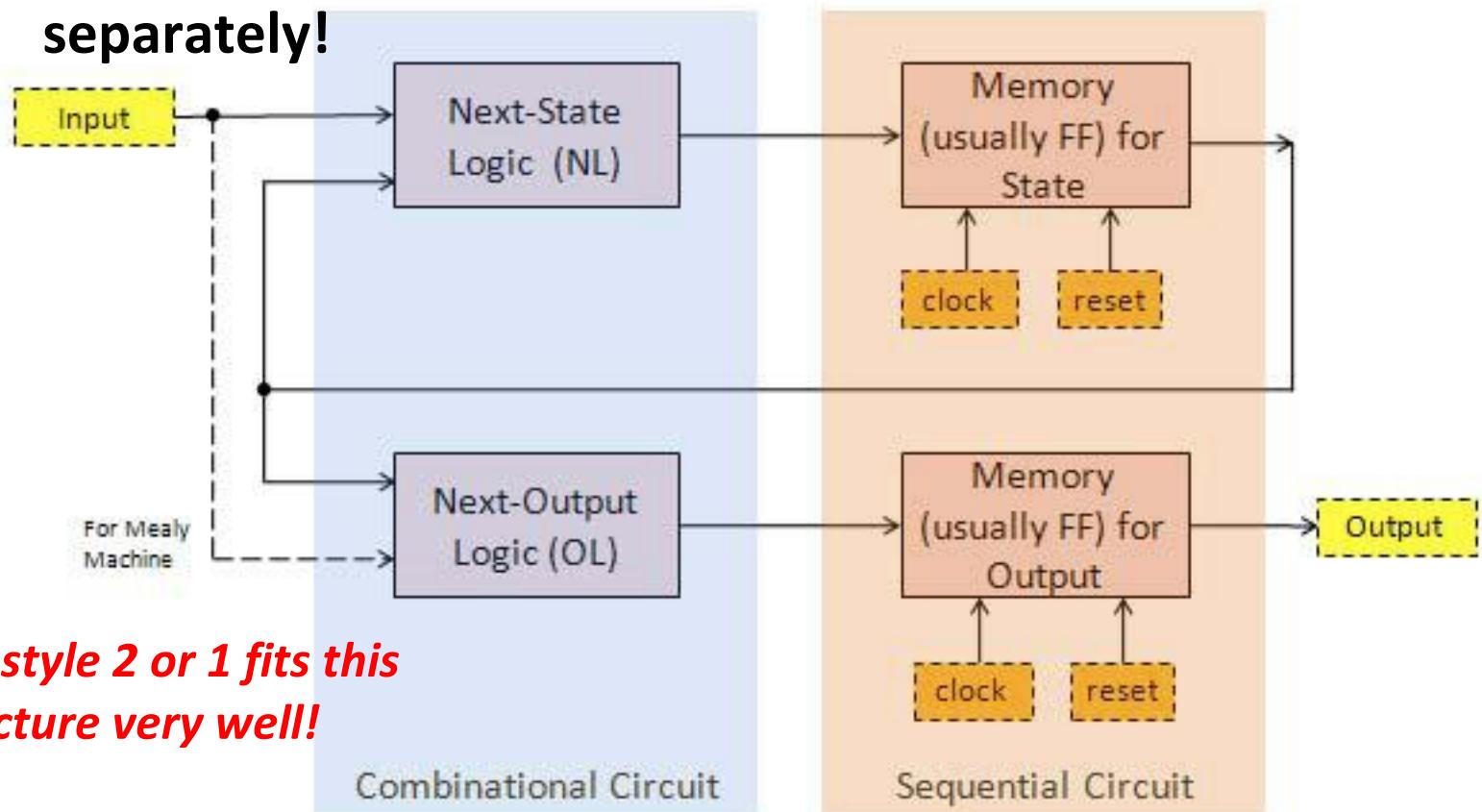
```
// if Moore
always @ (state)
    z = output_value;
```

```
// if Mealy
always @ (state or In)
    z = output_value;
```



Architecture of FSM

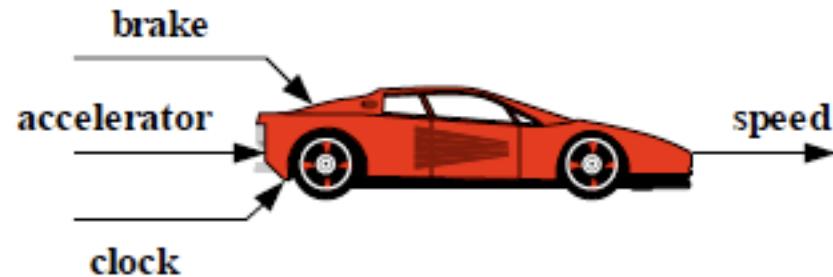
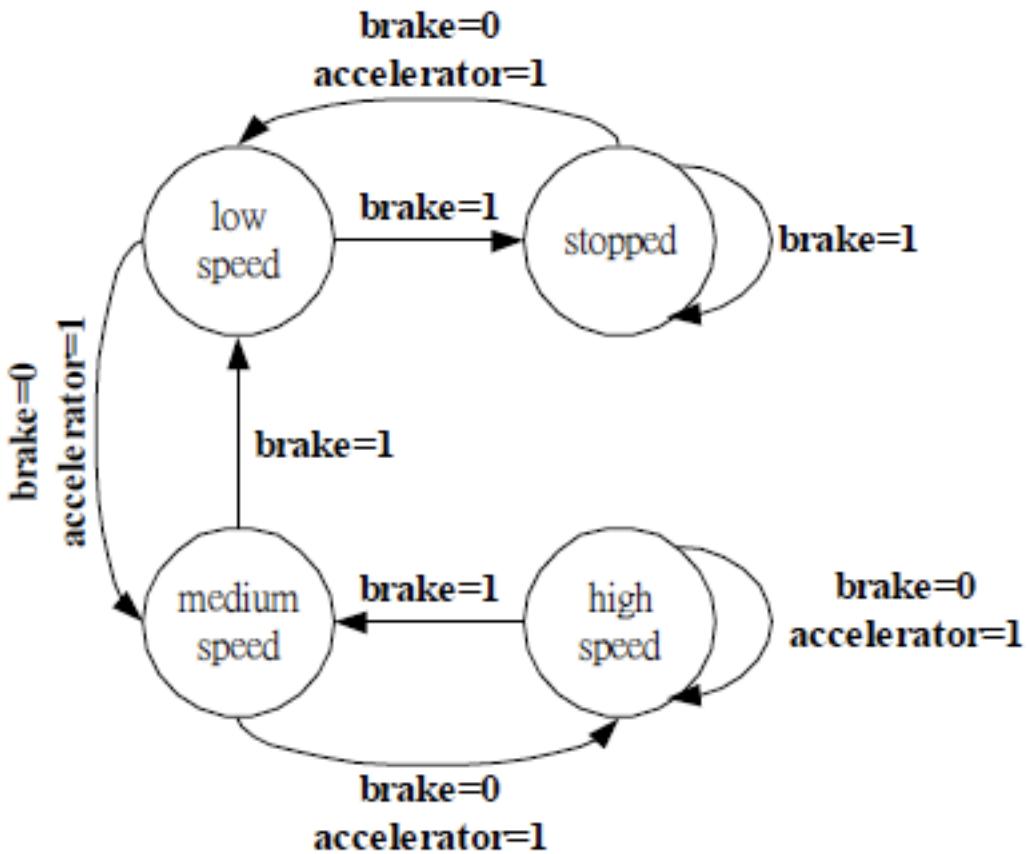
Build combinational and sequential parts separately!



Coding style 2 or 1 fits this architecture very well!



FSM Example: Speed Machine





FSM Example: Reference Code Using Coding Style 1

```

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     input      clock, reset;
18     input      accelerator, brake;
19     output [1:0] speed;
20
21 //==== REG/WIRE DECLARATION =====-
22     //--- wires ---
23     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     assign next_speed = state;
33
34     //--- next-state logic (NL) ---
35     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     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     end
69 endmodule

```



FSM Design Notice

- ❖ Partition FSM and non-FSM logic
- ❖ Partition combinational part and sequential part
 - Coding style 1, 2 are preferred
 - **For beginner**, Do not use Coding style 3
- ❖ Use **parameter** to define names of the state vector
- ❖ Assign a default (reset) state



References

- ❖ TSRI Verilog 上課講義
- ❖ IEEE Standard for Verilog Hardware Description Language
- ❖ Verilog 教學網站