Outline

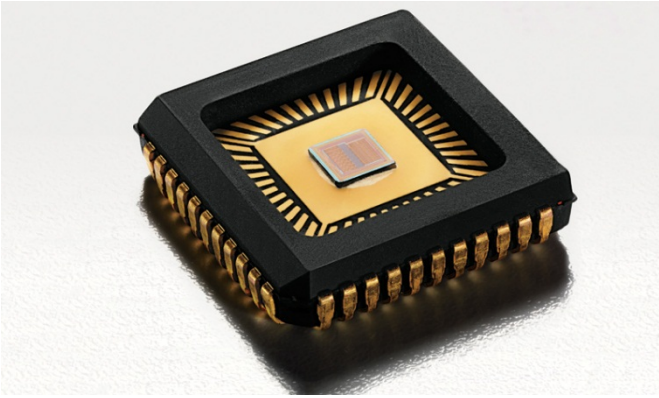
- **Introduction** — Part 1
- **Verilog for RTL**
  - **Module Description and Declaration**
  - **Data Type and Operators**
  - **Combinational Behavior**
  - **Sequential Behavior**
  - **Finite State Machine** — Part 2
  - **Advanced Topics**
- **Design Example**
- **RTL Simulation Tool**
- **Synthesis Tool** — Part 3

Outline

- **Introduction**
- **Verilog for RTL**
  - Module Description and Declaration
  - Data Type and Operators
  - Combinational Behavior
  - Sequential Behavior
  - Finite State Machine
  - Advances Topics
- **Design Example**
- **RTL Simulation Tool**
- **Synthesis**

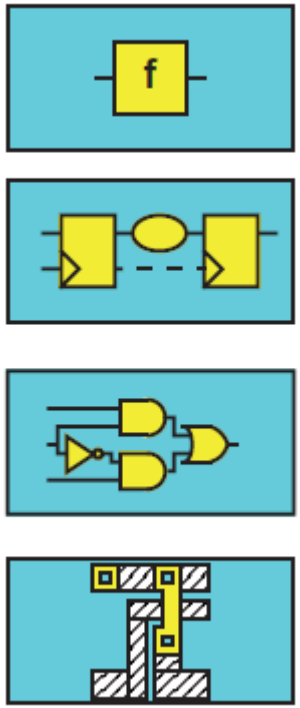
Introduction

- **What is Verilog?**
  - Hardware Description Language (HDL)
  - Mostly use in digital design.
- **Why we need to learn Verilog?**
  - Millions or billions the number of transistors!
  - We need EDA (Electronic design automation) tools.



Verilog-Supported Levels of Abstraction

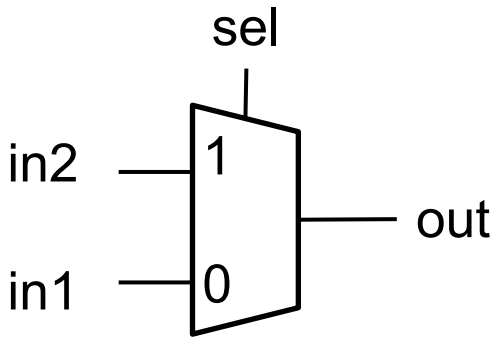
- **Behavior Level**
  - Modeling the circuit's behavior in high-level.
- **Register Transfer Level (RTL)** We focus on RTL in this course.
  - Describes the flow of data between registers and how a design process the data.
- **Gate Level**
  - Describes the logic gates and the interconnections.
- **Transistor Level**
  - Describes the transistors and the interconnections.



Verilog-Supported Levels of Abstraction

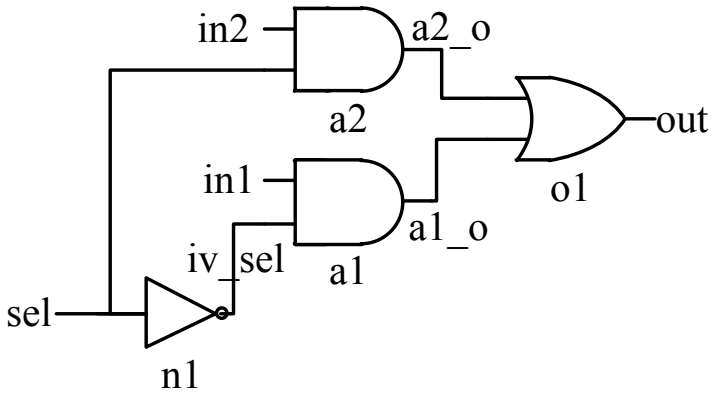
- **RTL**

```
module mux2to1(in1, in2, sel, out)
  input in1, in2, sel;
  output out;

  always@(*) begin
    if(sel==0) out=in1;
    else  out=in2;
  end
endmodule
```

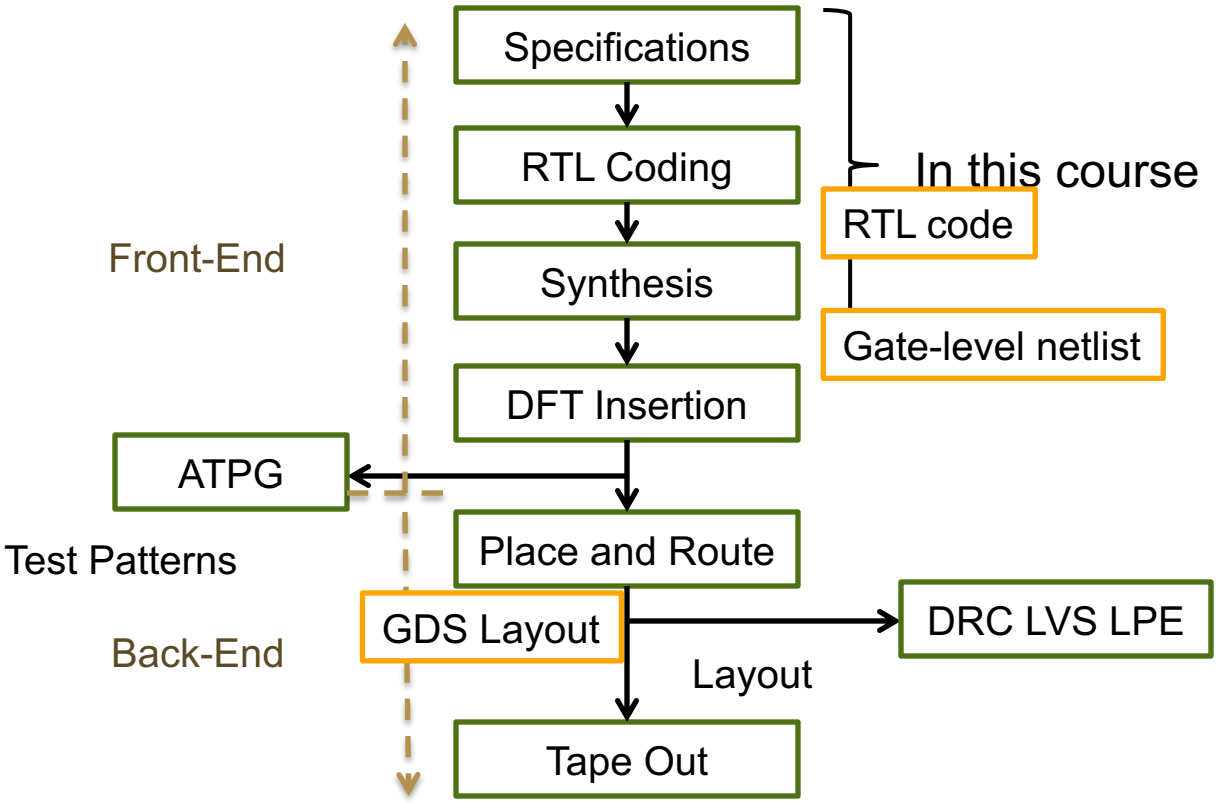


- **Gate Level**

```
module mux2to1(in1, in2, sel, out)
  input in1, in2, sel;
  output out;
  wire inv_sel, a1_o, a2_o;

  and a1(a1_o, in1, iv_sel);
  not n1(iv_sel, sel);
  and a2(a2_o, in2, sel);
  or  o1(out, a1_o, a2_o);
endmodule
```



Cell-Based Design



Summary

- **Verilog is important for digital IC design**
- **Three level**
  - RTL
  - Gate
  - Transistor

Outline

- Introduction
- **Verilog for RTL**
  - **Module Description and Declaration**
  - **Data Type and Operators**
  - **Combinational Behavior**
  - **Sequential Behavior**
  - Finite State Machine
  - Advances Topics
- Design Example
- RTL Simulation Tool
- Synthesis

Module Description and Declaration

```
4  module counter( clk, rst, down, num);
5  // --- Input/Output declaration
6    input         clk;
7    input         rst;
8    input         down;
9    output [3:0] num;
10
11 // --- MARCRO declaration for FSM
12   parameter IDLE = 2'b00;  // 2 bit binary 00
13   parameter DOWN = 2'd1;   // 2 bit decimal 1 equals to 00
14   parameter ZERO = 2'd2;
15
16 // --- Wire/ Reg declaration  ---
17   reg  [1:0] state, next_state;
18   reg  [3:0] count;
19   reg  [3:0] next_count;
20   reg  [3:0] num,   next_num;
21   wire [3:0] wire_next_num;
22
```

module 宣告

MARCO

wire / reg 宣告

```
56
57 // ------------------------------------
58 //   Sequential Part
59 // ------------------------------------
60   always@(posedge clk or posedge rst)
61   begin
62     if(rst) begin
63       state <= IDLE;
64       count <= 4'd10;
65       num   <= 4'd10;
66     end
67     else begin
68       state <= next_state;
69       count <= next_count;
70       num   <= next_num;
71     end
72   end
73 endmodule
74
```

Sequential Part

module 宣告

```
22
23 // ------------------------------------
24 //   Combinational Part
25 // ------------------------------------
26
27  // continous assignment
28  assign wire_next_num = next_num;
29
30  // procedural assignment
31  always@(*) begin
32    case(state)
33      IDLE: begin
34        if (down==1'b1) next_state = DOWN;
35        else next_state = IDLE;
36        next_num = 4'd10;
37        next_count = (down ==1'b1) ? count-1'b1 :4'd10;
38      end
39      DOWN: begin
40        next_state = (count==4'd0) ? ZERO : DOWN;
41        next_count = (count==4'd0) ? 4'd0 : count-1'b1;
42        next_num   = count;
43      end
44      ZERO: begin
45        next_state = IDLE;
46        next_count = 4'd10;
47        next_num   = count;
48      end
49      default: begin
50        next_state = state;
51        next_count = count;
52        next_num   = num;
53      end
54    endcase
55  end
56
```

Combinational Part

Module Description and Declaration

- **It's essential part and description of module interface.**
- **Basic format**

```
module  <module name>   (  <port list>  );
   input    [vector length] <signal A> ;
   input    [vector length] <signal B> ;
   output   [vector length] <signal C> ;
…
endmodule
```

- **Example**

```
module counter( clk, rst, down, num);
// --- Input/Output declaration
  input        clk;
  input        rst;
  input        down;
  output [3:0] num;
endmodule
```

Data Type and Operators

- **Number representation**

  <number of bits>' <radix type> <value>

  – 4'd9 : means 4 bits in decimal 9 (4'b1001 in binary)

  – 10'b01: means 10 bits in binary 00_0000_0001

  – 8'h10 : means 8 bits hexadecimal 10 (8'b0001_0000 in binary)

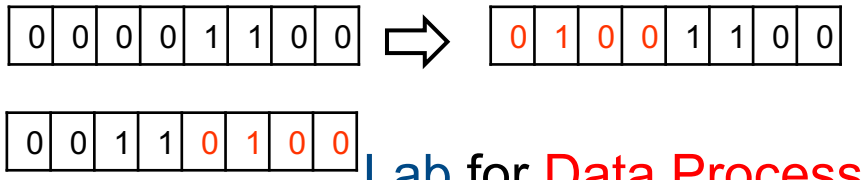
  – default value is decimal if there is no specific radix type

  – Bad example : 4'd18, 3'b6, 6'hAB….

- **Verilog support bitwise assignment**

```
<signal A>          = f( < signal B> [N:M] ) ;  //  only used some bits
<signal A> [ N:M] = <function value>            //  only assign some bits
```

```
A = 8'd12;
B = 8'h34;
A[7:4] = B[3:0];
```



Data Type and Operators

- **Common used operators**

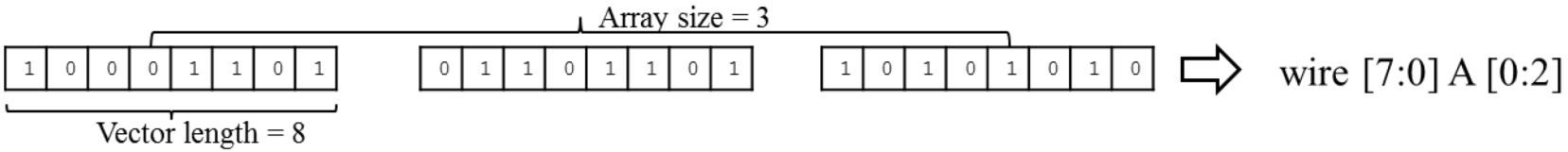
Type	Operators	Examples	
Arithmetic	+, -, *, /, %	A = B+C;	1001 = 0111+0010
Bitwise	~, &, |, ^, ~^	A = B^C;	0101 = 0111^ 0010
Logical	!=, ==, &&, ||	A == B	0 = (0111 == 0101)
Relational	>, >=, <, <=	A>B	1 = (0111 > 0101)
Shift	>>, <<	A= B << 1;	1110 = 0111 << 1
Concatenation	{ }	A = {B, C}	0110 = {01, 10}

Data Type and Operators

- **Frequently used data type are “wire” and “reg”**
- **Basic format**

wire [vector length] < variable name> [array size] ;

reg [vector length] < variable name> [array size] ;



- **Example**

```
// --- Wire/ Reg declaration  ---
  reg  [1:0] state, next_state;
  reg  [3:0] count;
  reg  [3:0] next_count;
  reg  [3:0] num,   next_num;
  reg  [7:0] example_array [0:127]; //128 words, 8bit
  wire [3:0] wire_next_num;
```

Data Type and Operators

- **wire**
  - input / output port is set to wire as default.
  - Value can only be changed by “assign”.
- **reg**
  - Value can only be changed by “always” block.
- **Example**

```
// --- The use of wire and reg ----
wire A,B,C;
reg D;

assign C = A & B;
always@ (*)
begin
  D = A & B;
end
```

Data Type and Operators

- **parameter**
  - "constant" in Verilog
  - Basic format

parameter < variable name> = number;

- **Example**

```
// --- parameter ----
parameter width = 3'd7;
wire [width:0] w1;
```

Combinational Behavior (1/4)

- **Describing the combinational circuit has two style:**
  - Continuous assignment
  - Procedural assignment 
- **Continuous assignment**

  assign <signal A> = <function value> ;

  - Signal at left hand side(LHS) must be **wire** data type
  - Signal function value at RHS can be wire or reg data type
  - Ensure bit length is long enough

```
// --- Continuous assignment ----
assign C = A & B;
assign D = (A==B)?A:B;
assign E = A + B;
assign F = A << 2;
```

Combinational Behavior (2/4)

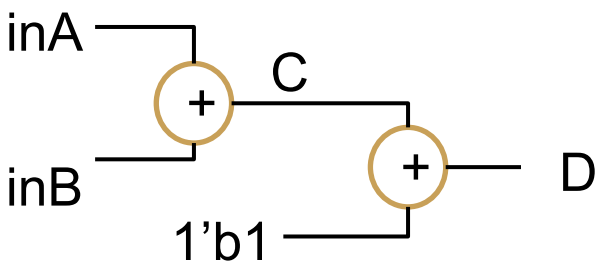
- **Procedural assignment**

```
always @ (sensitivity list)
      <signal A> = <function value>;
```

  - Signal at LHS must be **reg** datatype
  - Signal at RHS can be reg or wire
  - If there are 2 or more statements in always block, remember to add “begin” and “end”
  - All signal in RHS must include in sensitivity list. (or use ”∗”)

```
wire [3:0] inA, inB;
reg  [3:0] C, D;

always@(inA or inB or C)
begin
    C = inA+inB;
    D = C+1'b1;
end
```

```
wire [3:0] inA, inB;
reg  [3:0] C, D;

always@(*)
begin
    C = inA+inB;
    D = C+1'b1;
end
```



Combinational Behavior (3/4)

- **Conditional Statement**
  - For procedural assignment

```
case ( target )
    condition1: statement1 ;
    condition2: statement2 ;
            …
    default:    statement N;
endcase
```

```
if  ( condition1 )
    statement1 ;
else if  ( condition2 )
    statement2 ;
else
    statement3 ;
```

```
case(en)
  0: begin
    out1 = inB;
    out2 = 0;
  end
  1: begin
    out1 = inA;
    out2 = inA+inB;
  end
  default: begin
    out1 = inB;
    out2 = 0;
  end
endcase
```

```
if(en==1'b1) begin
  out1 = inA;
  out2 = inA+inB;
end
else begin
  out1 = inB;
  out2 = 0;
end
```

※ Remember write all possible case or use default.

※ The incomplete case will cause a Latch after synthesis.

Combinational Behavior (4/4)

- **Conditional Statement**
  - For continuous assignment

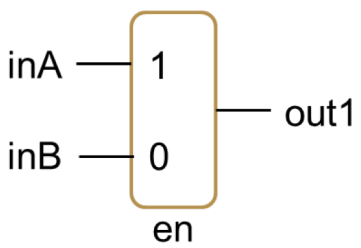
assign signal A = (contition) ? (True Signal) : (False Signal) ;

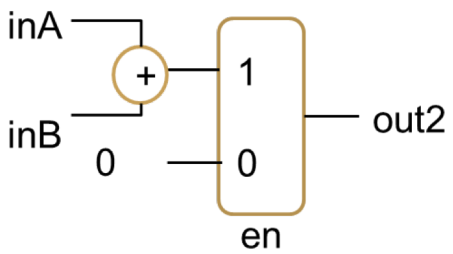
```
assgin out1 = (en==1'b1) ? inA : inB ;
assgin out2 = (en==1'b1) ? inA+inB : 0;
```

  - You can't use if-else or case statement in continuous assignment statement.

```
if (en==1)
  assign out1= inA;
else
  assgin inB;
```

X





Sequential Behavior (1/4)

- **Describe the behavior of sequential circuit**

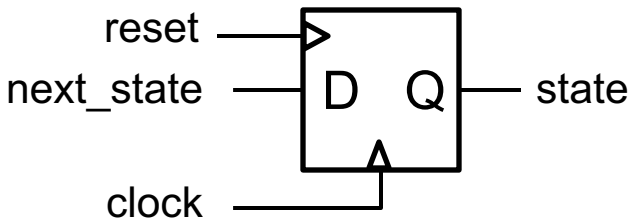
```
always @ ( <sensitivity edge > <clock>  [or  <sensitivity edge> < reset >] )
      if ( reset ) … ;     //  reset mode
      else         … ;     //  normal mode
```

Asynchronous reset

```
58  always @ (posedge clock or posedge reset) begin
59      if ( reset ) begin
60          state <= RED;
61          count <= 0;
62      end
63      else begin
64          state <= next_state;
65          count <= next_count;
66      end
67  end
```

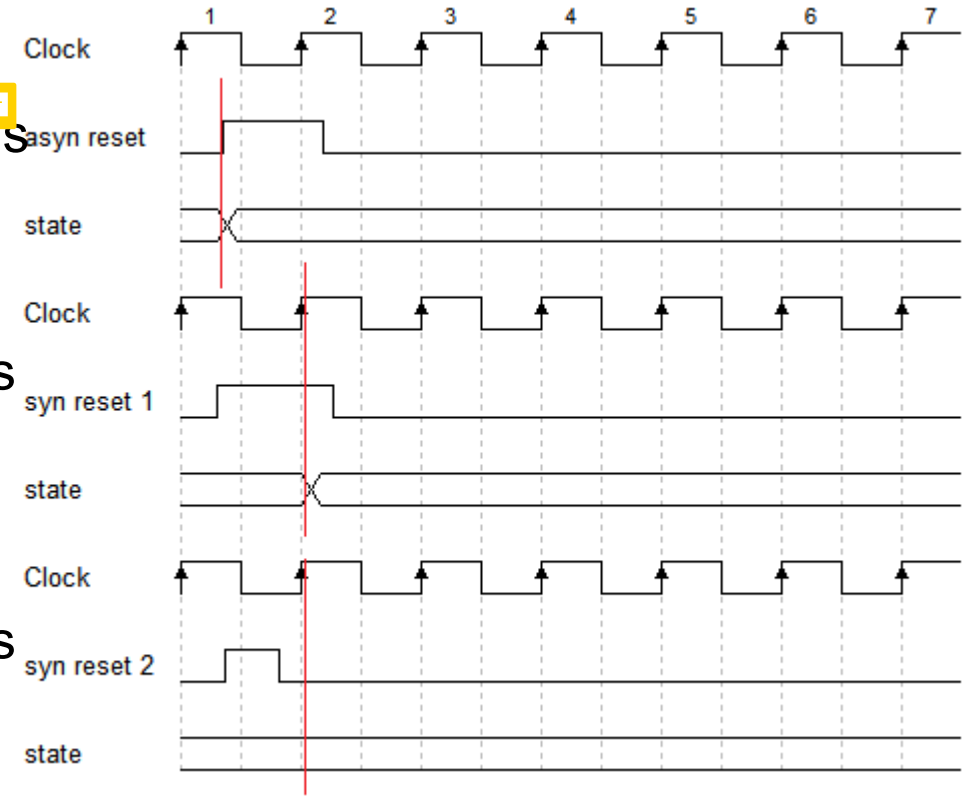
Synchronous reset

```
58  always @ (posedge clock) begin
59      if ( reset ) begin
60          state <= RED;
61          count <= 0;
62      end
63      else begin
64          state <= next_state;
65          count <= next_count;
66      end
67  end
```



Sequential Behavior (2/4)

- **Difference between asynchronous reset and synchronous reset**



Asynchronous reset

reset at the posedge of reset signal

Synchronous reset

reset at the posedge of clock signal

Synchronous reset

no reset occurs

※The circuit is synchronous to posedge clk and active high reset

Sequential Behavior (3/4)

- The circuit must have a reset signal. Otherwise, the initial value of Flip-Flop will be unknown.
- Asynchronous or synchronous reset type depends on the circuit's spec.
- non-blocking assignment " <=" for sequential circuit.
- blocking assignment "=" for combinational circuit.

blocking assignment

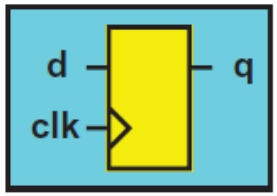
```
always @ (*) begin
    case( i_IDEX_AluCtrl )
        4'b0000:  AluResult = AluIn1 & AluIn2;
        4'b0001:  AluResult = AluIn1 | AluIn2;
        4'b0101:  AluResult = AluIn1 ^ AluIn2;
        4'b0110:  AluResult = AluIn1 - AluIn2;
        4'b0111:  AluResult = AluIn1<AluIn2)? 32'b1: 32'b0;
        4'b1010:  AluResult = AluIn2 << i_IDEX_Extend[10:6];
        4'b1101:  AluResult = ~(AluIn1 | AluIn2);
        4'b1110:  AluResult = AluIn2 >> i_IDEX_Extend[10:6];
        default:  AluResult = AluIn1 + AluIn2;
    endcase
end
```

non-blocking assignment

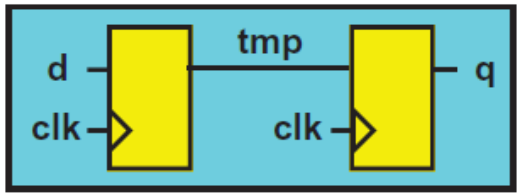
```
58  always @ (posedge clock or posedge reset) begin
59      if ( reset ) begin
60          state <= RED;
61          count <= 0;
62      end
63      else begin
64          state <= next_state;
65          count <= next_count;
66      end
67  end
```

Sequential Behavior (4/4)

- **Difference between non-blocking assignment and blocking assignment in synchronous block.**




```
module block (q, d, clk);
input d, clk;
output q; reg q;
  reg tmp;
  always @(posedge clk)
    begin
      tmp = d;
      q = tmp;
    end
endmodule
```




```
module noblock (q, d, clk);
input d, clk;
output q; reg q;
  reg tmp;
  always @(posedge clk)
    begin
      tmp <= d;
      q <= tmp;
    end
endmodule
```