Verilog

Why Should I Learn This Language?

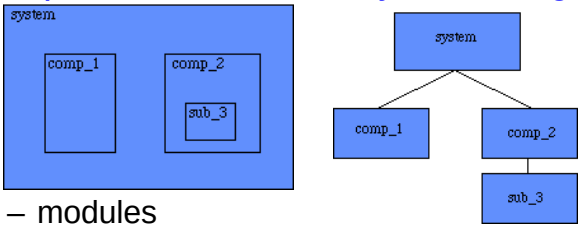
- 俗話說得好：〝大塊假我以文章〞，當硬體發展到VLSI這種〝大塊〞的地步，以非人力可以處理，故吾人學習寫HDL文章，是不得不的趨勢。

Different Levels of Abstraction

- Algorithmic
  - the function of the system
- RTL
  - the data flow
  - the control signals
  - the storage element and clock
- Gate
  - gate-level net-list
- Switch
  - transistor-level net-list

Hierarchical structure

- Represent the hierarchy of a design



  - modules
    - the basic building blocks
  - ports
    - the I/O pins in hardware
    - input, output or inout

Modules

module module_name (port_name);

port declaration

data type declaration

module functionality or structure

endmodule

Examples

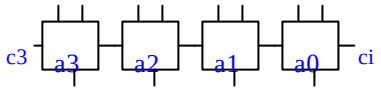
- 4-bit adder

```
module add4 (s,c3,ci,a,b)
    input [3:0] a,b ;  // port declarations
    input ci ;
    output [3:0] s :   // vector
    output c3 ;
    wire [2:0] co ;
    add a0 (co[0], s[0], a[0], b[0], ci) ;
    add a1 (co[1], s[1], a[1], b[1], co[0]) ;
    add a2 (co[2], s[2], a[2], b[2], co[1]) ;
    add a3 (c3, s[3], a[3], b[3], co[2]) ;
endmodule
```

port

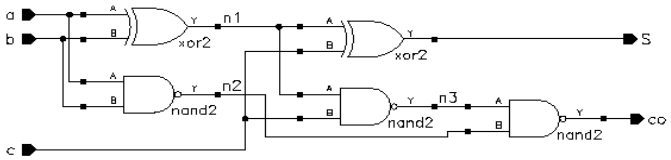
data type

Structure



- A full-adder

```
module add (co, s, a, b, c)
    input a, b ,c ;
    output co, s ;
        xor (n1, a, b) ;
        xor (s, n1, c) ;
        nand (n2, a, b) ;
        nand (n3,n1, c) ;
        nand (co, n3,n2) ;
endmodule
```



Sub-module

```
module adder (co,s,a,b,c);
.......
endmodule
```

```
module adder8(....) ;
adder  add1(c1,s1,a1,b1,1'b0) ,
       add2(.a(a2),.b(b2),.c(c1),.s (s2)
            ,.co(c2)) ;
.....
endmodule
```

Mapping port positions

Mapping names

Data Type

- Data type:
  - input : default " wire"
  - output :

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

```
dff dff1(D,clk,Q);
```

"wire"

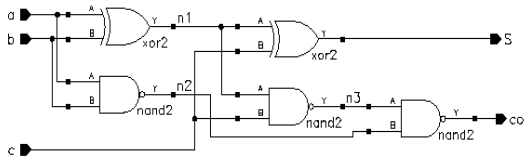
```
always@ (D or clk or Q)begin
    Q=(D&clk)|(~clk&Q);
end
```

" reg "

Gate-Level Modeling

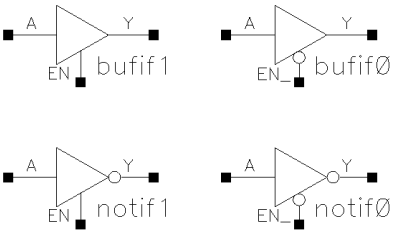
- Net-list description
  - built-in primitives gates
- A full-adder

```
module add (co, s, a, b, c)
    input a, b ,c ;
    output co, s ;
        xor (n1, a, b) ;
        xor (s, n1, c) ;
        nand (n2, a, b) ;
        nand (n3,n1, c) ;
        nand (co, n3,n2) ;
endmodule
```



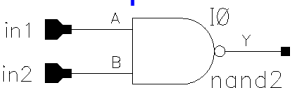
Verilog Primitives

- Basic logic gates only
  - and
  - or
  - not
  - buf
  - xor
  - nand
  - nor
  - xnor
  - bufif1, bufif0
  - notif1, notif0

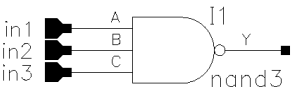


Primitive Pins Are Expandable

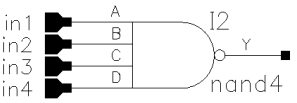
- One output and variable number of inputs



nand (y, in1, in2) ;



nand (y, in1, in2, in3) ;



nand (y, in1, in2, in3, in4) ;

- not and buf
  - variable number of outputs but only one input

Modeling Structures

- Continuous assignment
- Procedural blocks
  - initial block (execute only once)
  - always block (execute in a loop)

  they start together at simulation time-zero

Continuous Assignments

- Describe combinational logic
- Operands + operators
- Drive values to a net
  - assign out = a&b ; // and gate
  - assign eq = (a==b) ; // comparator
  - wire #10 inv = ~in ; // inverter with delay
  - wire [7:0] c = a+b ; // 8-bit adder
- Avoid logic loops
  - assign a = b + a ;
  - asynchronous design

Operators

{ }	concatenation
+ - * /	arithmetic
%	modulus
> >= < <=	relational
!	logical NOT
&&	logical AND
||	logical OR
==	logical equality
!=	logical inequality
? :	conditional

~	bit-wise NOT
&	bit-wise AND
|	bit-wise OR
^	bit-wise XOR
^~ ~^	bit-wise XNOR
<<	shift left
>>	shift right

Operator Precedence

[ ]	bit-select or part-select
( )	parentheses
!, ~	logical and bit-wise negation
&, |, ~&, ~|, ^, ~^, ^~	reduction operators
+, -	unary arithmetic
{ }concatenation	
*, /, %	arithmetic
+, -	arithmetic
<<, >>	shift

>, >=, <, <=	relational
==, !=	logical equality
&	bit-wise AND
^, ^~, ~^	bit-wise XOR and XNOR
|	bit-wise OR
&&logical AND	
||	logical OR
? :	conditional

RTL Statements

- Procedural and RTL assignments
  - reg
  - out = a + b ;
- begin . . . end block statements
  - group statements
- if. . . else statements
- case statements
- for loops
- while loops
- forever loops

Combinational Always Blocks

- A complete sensitivity list (inputs)

```
always @(a or b or c)
    f = a&~c | b&c ;
```

- Simulation results

```
always @(a or b)
    f = a&~c | b&c ;
```

- Parentheses

```
always @(a or b or c or d)
    z = a + b + c + d ;        // z = (a+b) + (c+d) ;
```

Sequential Always Blocks

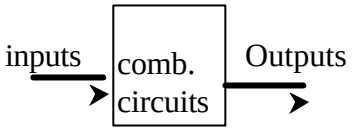
- Inferred latches (Incomplete branch specifications)

```
module infer_latch(D, enable, Q);
  input D, enable;
  output Q;
  reg Q;
    always @ (D or enable) begin
      if (enable)
      Q <= D;
    end
endmodule
```

  - the Q is not specified in a branch
    - a latch like 74373

Combinational Circuit Design

- Outputs are functions of inputs

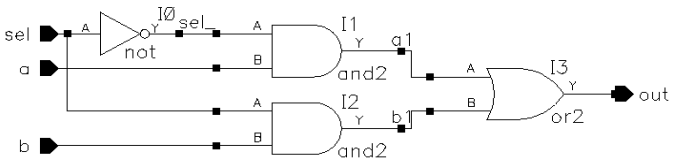


- Examples
  - MUX
  - decoder
  - priority encoder
  - adder

Multiplexor

- Net-list (gate-level)

```
module mux2_1 (out,a,b,sel) ;
        output out ;
        input a,b,sel ;
        not (sel_, sel) ;
        and (a1, a, sel_) ;
        and (b1, b, sel) ;
        or (out, a1, b1) ;
endmodule
```



Multiplexor

- Continuous assignment

```
module mux2_1 (out,a,b,sel) ;
    output out ;
    input a,b,sel ;
    assign out = (a&~sel)|(b&sel) ;
endmodule
```

- RTL modeling

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

Multiplexor

- 4-to-1 multiplexor

```
module mux4_1 (out, in0, in1, in2, in3, sel) ;
  output out ;
  input in0,in1,in2,in3 ;
  input [1:0] sel ;
    assign out = (sel == 2'b00) ? in0 :
          (sel == 2'b01) ? in1 :
          (sel == 2'b10) ? in2 :
          (sel == 2'b11) ? in3 :
                1'bx ;
endmodule
```



```
module mux4_1 (out, in, sel) ;
  output out ;
  input [3:0] in ;
  input [1:0] sel ;
  reg out ;
    always @(sel or in) begin
      case(sel)
         2'd0: out = in[0] ;
        2'd1: out = in[1] ;
         2'd2: out = in[2] ;
         2'd3: out = in[3] ;
         default: 1'bx ;
      endcase
    end
endmodule
```

Decoder

- 3-to 8 decoder with an enable control

```
module
   decoder(o,enb_,sel) ;
output [7:0] o ;
input enb_ ;
input [2:0] sel ;
reg [7:0] o ;
always @ (enb_ or sel)
  if(enb_)
     o = 8'b1111_1111 ;
  else
     case(sel)
     3'b000 : o = 8'b1111_1110 ;
     3'b001 : o = 8'b1111_1101 ;
     3'b010 : o = 8'b1111_1011 ;
     3'b011 : o = 8'b1111_0111 ;
     3'b100 : o = 8'b1110_1111 ;
     3'b101 : o = 8'b1101_1111 ;
     3'b110 : o = 8'b1011_1111 ;
     3'b111 : o = 8'b0111_1111 ;
     default : o = 8'bx ;
     endcase
endmodule
```


Priority Encoder

```
always @ (d0 or d1 or d2 or d3)
     if (d3 == 1)
          {x,y,v} = 3'b111 ;
     else if (d2 == 1)
          {x,y,v} = 3'b101 ;
     else if (d1 == 1)
          {x,y,v} = 3'b011 ;
     else if (d0 == 1)
          {x,y,v} = 3'b001 ;
     else
          {x,y,v} = 3'bxx0 ;
```

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Adder

- RTL modeling

```
module adder(c,s,a,b) ;
output c ;
output [7:0] s ;
input [7:0] a,b ;
      assign {c,s} = a + b ;
endmodule
```

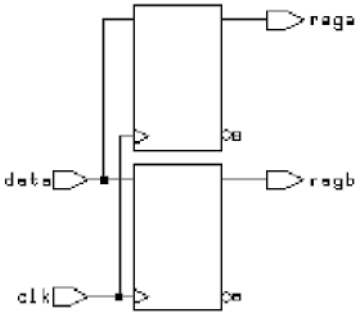
- Logic synthesis
  - CLA adder for speed optimization
  - ripple adder for area optimization

Procedural Assignments

- Blocking assignments

```
always @(posedge clk) begin
      rega = data ;
      regb = rega ;
end
```

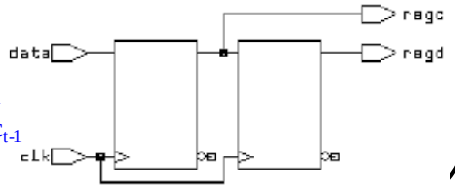
$rega_t$=data
$regb_t$=data



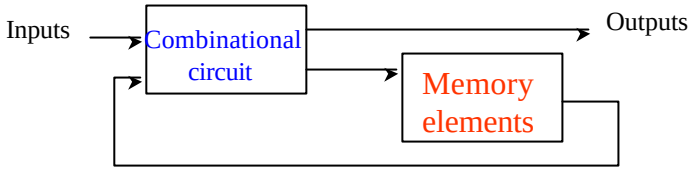
- Non-blocking assignments

```
always @(posedge clk) begin
      regc <= data ;
      regd <= regc ;
end
```

$regc_t$=data
$regd_t$=$regc_{t-1}$



Sequential Circuit Design



- a feedback path
- the state transition
  - synchronous circuits
  - asynchronous circuits

- Examples
  - D flip-flop
  - D latch
  - register
  - shifter
  - counter
  - pipeline
  - FSM

Flip-Flop

- Synchronous clear

```
module d_ff (q,d,clk,clr_) ;
output q ;
input d,clk,clr_ ;
reg q ;
always @ (posedge clk)
   if (~clr_)  q = 0 ;
   else        q = d ;
endmodule
```

- Asynchronous clear

```
always @ (posedge clk or negedge clr_)
   if (~clr_)  q = 0 ;
   else        q = d ;
```


Register

```
module register (q,d,clk,clr_, set_) ;
output [7:0] q ;
input [7:0] d ;
input clk,clr_, set_ ;
reg [7:0] q ;
always @ (posedge clk or negedge clr_ or negedge set_)
   if (~clr_)
     q = 0 ;
   else if (~set_)
     q = 8'b1111_1111 ;
   else
     q = d ;
endmodule
```

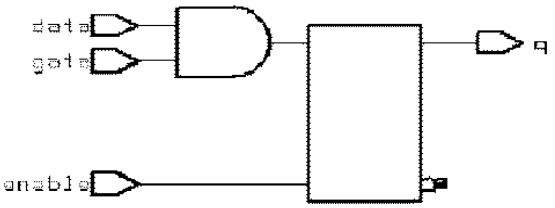
D Latches

- D latch

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

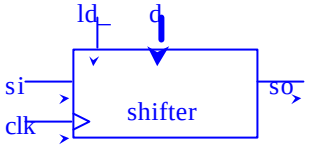
- D latch with gated asynchronous data

```
always @ (enable or data or gate)
   if (enable)
     q = data & gate
```



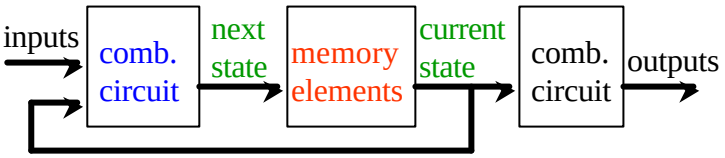
Shifter

```
module shifter (so,si,d,clk,ld_,clr_) ;
output so ;
input [7:0] d ;
input si,clk,ld_,clr_ ; // asynchronous clear and synchronous
   load
reg [7:0] q ;
assign so = q[7] ;
always @ (posedge clk or negedge clr_)
   if (~clr_)
     q = 0 ;
   else if (~ld_)
     q = d ;
   else
   q[7:0] = {q[6:0],si} ;
endmodule
```

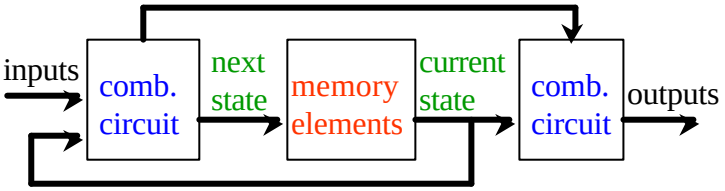


Finite State Machine

- Moore model



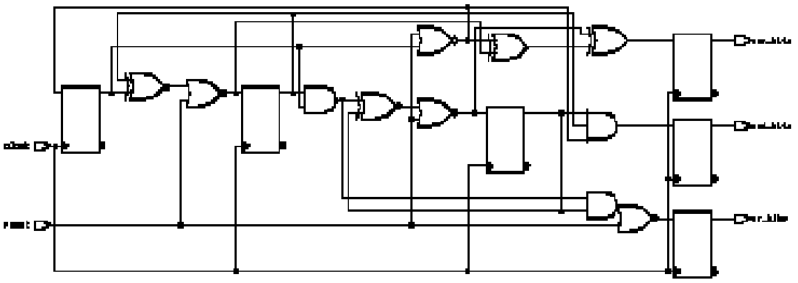
- Mealy model



Inefficient Description

```
module count (clock, reset, and_bits, or_bits, xor_bits);
input clock, reset;
output and_bits, or_bits, xor_bits;
reg and_bits, or_bits, xor_bits;
reg [2:0] count;
   always @(posedge clock) begin
    if (reset)
       count = 0;
    else
       count = count + 1;
       and_bits = & count;
       or_bits  = | count;
       xor_bits = ^ count;
   end
endmodule
```

- Six implied registers



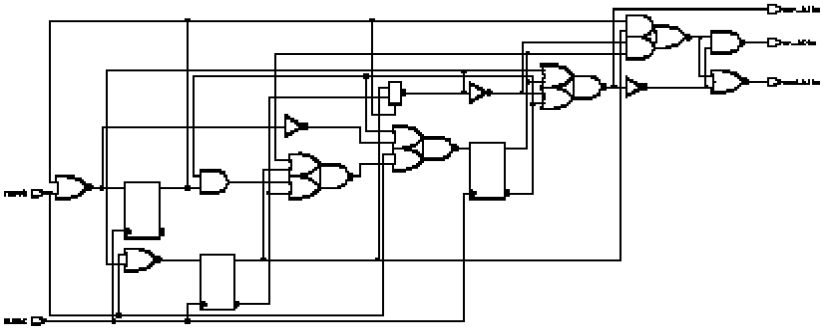
Efficient Description

- Separate combinational and sequential circuits

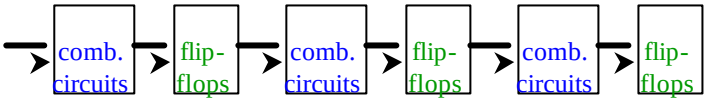
```
module count (clock, reset,
    and_bits, or_bits, xor_bits);
input clock, reset;
output and_bits, or_bits,
    xor_bits;
reg and_bits, or_bits, xor_bits;
reg [2:0] count;
always @(posedge clock) begin
    if (reset)
        count = 0;
    else
        count = count + 1;
    end
endmodule
```

```
// combinational circuits
always @(count) begin
    and_bits = & count;
    or_bits  = | count;
    xor_bits = ^ count;
    end
endmodule
```

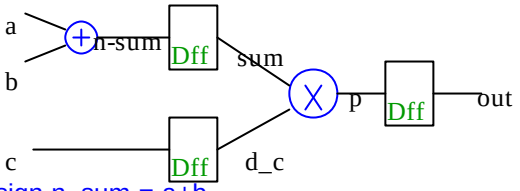
- Three registers are used



Pipelines



- An example




```
assign n_sum = a+b
assign p = sum * d_c
// plus D flip-flops
always @ (posedge clk)
   sum = n_sum ;
```

Memory

```
module memory (data, addr, read, write);
   input read, write;
   input [4:0] addr;
   inout [7:0] data;
   reg [7:0] data_reg;
   reg [7:0] memory [0:8'hff];
   parameter load_file = "cput1.txt";
   assign data = (read) ? memory [addr] : 8'hz;
   always @ (posedge write)
     memory[addr] = data;
   initial
     $readmemb (load_file, memory);
endmodule
```