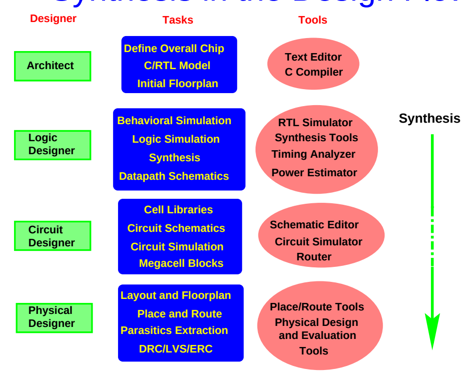
13. Introduction to Verilog

- Last module:
  - Sequential circuit design
  - Design styles
- This module
  - Synthesis
  - Brief introduction to Verilog

Synthesis in the Design Flow



Logic Synthesis

- Design described in a Hardware Description Language (HDL)
  - Verilog, VHDL
- Simulation to check for correct functionality
  - Simulation semantics of language
- Synthesis tool
  - Identifies logic and state elements
  - Technology-independent optimizations (state assignment, logic minimization)
  - Map logic elements to target technology (standard cell library)
  - Technology-dependent optimizations (multi-level optimization, gate strengths, etc.)

Features of Verilog

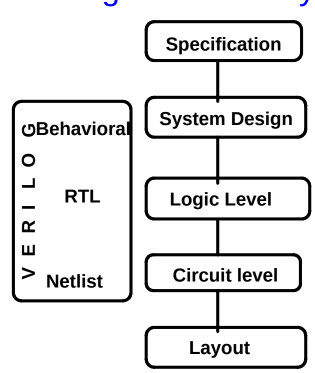
- A **concurrent** language (syntax similar to C)
- Models hardware
- Provides a way to specify concurrent activities
- Allows timing specifications

- Originally developed by *Gateway Design Automation*, acquired by *Cadence*, now IEEE Standard 1364-1995 (*Open Verilog International*)

Applications of Verilog

- Description of design at a higher level
- Development of formal models
- System documentation
- Simulation to uncover errors (bugs) in design
- Synthesis of designs
- Design reuse

Design Process Cycle

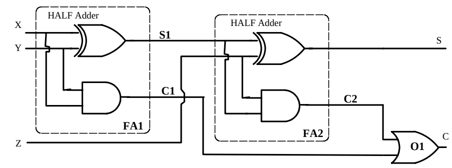


Verilog Modes

1. Structural: describes the structure of the hardware components, including how ports of modules are connected together
   - module contents are built in gates (and, or, xor, not, nand, nor, xnor, buf) or other modules previously declared
2. Behavoral: describes what should be done in a module
   - module contents are C-like assignment statements, loops

Structural Model of Adder

- Structural models: interconnections of primitive gates (*AND, OR, NAND, NOR*, etc...) and other modules




```
module HA( a, b, s, c)
input a,b;
output s, c;

xor G1 (s, a, b);
and G2 (c,a,b);

endmodule
```

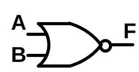
```
module FA(X,Y,Z,S,C
input   X, Y, Z;
output  S, C;

HA FA1 (X, Y, S1, C
HA FA2 (S1,Z, S, C2
or    O1    (C, C2,

endmodule
```


Specification of Gate-Level Model

- Gate function
  - Verilog built-in: *and, nand, or, nor, xor, xnor, buf, not,* etc.
- Gate specification format example:
  - *and #delay inst-name (out,in1, in2,...,ink);*
  - Delay and inst-name are optional

Logic Values

- A bit can have any of these values
  - 0 representing logic low (false)
  - 1 representing logic high (true)
  - x representing either 0, 1, or z
  - z representing high impedance for tri-state
  - (unconnected inputs are set to z)

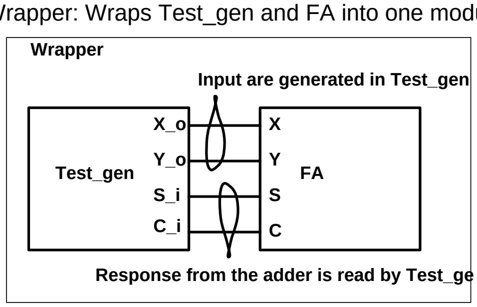


A \ B	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

F

Test Generation

- Test_gen: Generates test and capture the response
- FA: The design under test
- Wrapper: Wraps Test_gen and FA into one module



Test Generation, Cont'd

```
module HA( a, b, s, c)
input a,b;
output s, c;

xor G1 (s, a, b);
and G2 (c,a,b);

endmodule

module FA(X,Y,Z,S,C);
input   X, Y, Z;
output  S, C;

HA FA1 (X, Y, S1, C1);
HA FA2 (S1,Z, S, C2);
or    O1    (C, C2, C1);

endmodule
```

```
module test_gen(a,b,c,sum,cout);
input a,b,c;
output   sum, cout;
reg  a, b;
initial begin
$monitor($time," A=%b  B=%b Sum=%b Cout=%b",
                         a, b, sum, cout);
      a  = 0;   b=0;
   #5 a = 1;
   #5 b = 1;
   #5 c = 1;
end
endmodule
```

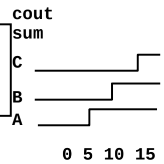
```
module wrapper;
wire   a, b, sum, cout;

FA(a,b,c,sum,cout);
test_gen(a,b,c,sum,cout);
endmodule
```

```
- initial
     tells verilog to execute all statement within begin end once it starts
- monitor
     tells verilog to monitor the list of variables and everytime a var. changes
     it prints the string
```

Test Generation, Cont'd

```
module test_gen(a,b,c,sum,cout);
input  a,b,c;
output  sum, cout;
reg  a, b;
initial begin
$monitor($time," A=%b  B=%b Sum=%b Cout=%b", a, b, sum, cout);
   a  = 0;   b=0;  c=0;
  #5 a = 1;
  #5 b = 1;
  #5 c = 1;
end
endmodule
```

```
module wrapper;
  wire   a, b, sum, cout;
  FA(a,b,c,sum,cout);
  test_gen(a,b,c,sum,cout);
endmodule
```



Behavioral Modeling

- Behavioral model consists of *always* and *initial* constructs
- All behavioral statements must be within these blocks
- Many *initial/always* statements can exist within module

initial construct execute once at the start of the simulation

```
initial
    begin
          statements
    end
```

always construct executes at the beginning of the simulation and continually loops

```
always @(sensitivity list)
    begin
          statements
    end
```

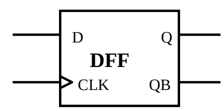

Behavioral Timing

- Advance time when:
  - #20 delay 20 time units
  - @(list) delay until an event occurs
  - wait: delay until a condition is satisfied

```
@r rega = regb;  // load rega when r changes
 @(posedge r) rega = regb;
            // load rega on positive edge of r
 @(negedge r) rega = regb;
            // load rega on negative edge of r
 wait(!r) rega = regb;
             // execution is suspended until r is 0
```


Edge Triggered DFF

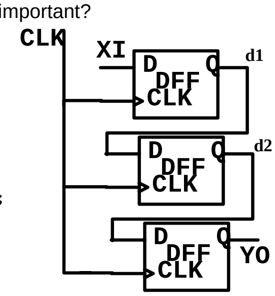
```
module  dff(q,qb,d,clk);
input d, clk;
output  q, qb;

reg     q;

always @(posedge clk)
    begin
       q = d;        // left hand side must be a register
    end
not G1 (qb, q);

endmodule
```



Shift Register

Is the order of execution important?

```
module  dff(q,d,clk);
input d, clk;
output  q;
reg     q;
     always @(posedge clk)
              q = d;
endmodule
module shift(YO,XI,CLK);
   input CLK, XI;
   output YO;

   dff (d1,XI,CLK);
   dff (d2,d1,CLK);
   dff (YO,d2,CLK);
endmodule
```



Behavioral Statements

```
if( expr ) then statement; else statement;

case(selector) val0: stat.; val1: stat.;
                default: stat; endcase;

for(i=0; i < 10; i=i+1) A = B + i;

i=0; while(i < 15)
          begin A = B + i; i = i + 1; end
```

Concurrent Constructs

- @ means wait for a change in value
  - @(a) w = 4; Wait for 'a' to change to resume execution
- wait(condition)
  - wait(a==1) w = 4; wait for a to become 1 to resume execution

Another Example

```
module mux(f, sel, b, a);
    input sel, b, a;
    output f;

    reg   f;

always @(sel or a or b)
        if( sel == 1 ) f = b;
                else f = a;

endmodule


assign f = sel ? b : a;
```


Case Statement

```
module function_table(f,a,b,c);
input a,b,c;
output f;

reg  f;

always @( a or b or c)
    case( {a,b,c} )     // concatenate a,b,c to form a 3 bit number
        3'b000:  f = 1'b0;
        3'b001:  f = 1'b1;
        3'b010:  f = 1'b0;
        3'b011:  f = 1'b0;
        3'b100:  f = 1'b0;
        3'b101:  f = 1'b1;
        3'b110:  f = 1'b0;
        3'b111:  f = 1'b0;
    endcase
endmodule
```


Adder With Delays

```
module adder(co,su,a,b,ci);
input a,b,ci;
output co,su;

xor  #(4,5)  (sum, a, b, cin);  // rise time=4 , fall time = !
or    #5       (co, ab,bc,ac);  // rise time = fall time = 5
and #(4,2)   (ab,a,b).             // three similar and gates
                 (ac,a,ci),
                 (bc,bci);
endmodule
```


Blocking and Nonblocking Assignments

- Blocking assignments are executed in the order they are coded
  - sequential execution of statements
  - block the execution of the next statement till the current statement is executed

    a = b;
- Nonblocking assignments are executed in parallel
  - execution of next statement is not blocked due to execution of current statement

    a <= b;

Adder With Continuous

```
module adder(co,su,a,b,ci);
input a,b,ci;
output co,su;

assign su = a ^ b ^ cin,
        co = a&b  | b&cin | a&ci ;
endmodule
```

Specifies combinational logic

Any time the right hand side changes, the simulator re-evaluates the output

Another Version with Delays

```
module adder(co,su,a,b,ci);
input a,b,ci;
output co,su;

assign  #(5,4) su = a ^ b ^ cin;
assign #(10,4) co = a&b  | b&cin | a&ci ;
endmodule
```

Counter

```
module counter(Q , clock, clear);
   output [3:0] Q;
   input clock, clear;
   reg [3:0] Q;

   always @( posedge clear or negedge clock)
   begin
      if (clear)   Q = 4'd0;
      else  Q = (Q + 1) ; //   Q = (Q + 1) % 16;
   end
   endmodule

module stimulus;
   reg CLOCK, CLEAR;
   wire [3:0] Q;
```


Counter, Cont'd

```
counter c1(Q, CLOCK, CLEAR);

   initial
   begin
         $monitor($time, " Count Q = %b Clear= %b",  Q[3:0],CLEAR);
         CLEAR = 1'b1;
         #34 CLEAR = 1'b0;
         #200 CLEAR = 1'b1;
         #50 CLEAR = 1'b0;
         #400 $stop;
   end

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

   endmodule
```


Bus Select

```
module select_bus( out, b0, b1, b2, b3, enable, s);
   parameter    n = 16;
   parameter    Z = 16'bz;  // define a 16 bit of z
   output [1:n] out;        // n-bit output
   input  [1:n] b0, b1, b2, b3;  // n-bit inputs
   input  enable;
   input  [1:2] s;

   tri   [1:n] data;          // tri-state net
   tri   [1:n] out = enable ? data: Z; // net declaration with
                                        // continuous assignment
   assign
        data = (s==0) ? b0 : Z,      // 4 continuous assignment
        data = (s==1) ? b1 : Z,
        data = (s==2) ? b2 : Z,
        data = (s==3) ? b3 : Z;
   endmodule
```


Tri-State Latch

```
module tri_latch( q, nq, clk, data, enable);
   output q, nq;
   input  clk, data, enable;
   tri    q, nq;

   not #5 (ndata, data);
   nand #(3, 5) (wa, data, clk), (wb, ndata, clk);
   nand #(12,15) (ql, nql, wa), (nql, ql, wb);

   bufif1 #(3,5,13)  // rise, fall, change to z
     q_dr ( q, ql, enable),     // when enable = 1, q=ql
     nq_dr (nq, nql, enable);   // when enable = 0, q=ql=z
   endmodule
```


User-Defined Primitives

```
primitive mux(mux,cntr,A,B);
   output mux;
   input  cntr, A, B;
      table
      // cntr A B   mux
         0   1 ? : 1 ;  // ? represents don't care
         0   0 ? : 0 ;
         1   ? 0 : 0 ;
         1   ? 1 : 1 ;
         x   0 0 : 0 ;
         x   1 1 : 1 ;
      endtable
   endprimitive
```


User-Defined Primitives, Cont'd

```
primitive latch(q, clk, data);
   output    q; reg q;
   input     clk, data;

      table
        //  clk  data state output/nxtstate
            0    1  : ?  :  1 ;
            0    0  : ?  :  0 ;
            1    ?  : ?  :  - ; // represent no change
      endtable
   endprimitive
   endprimitive
```

Looping

```
module mult( res, a, b);
  parameter size = 8, longsize = 16;
  input [size:1] a, b;
  output [longsize:1] res;

  reg [size:1] a, b;
  reg [longsize:1] res;

always @( a or b)
      begin :mult
        reg [longsize:1] shifta, shiftb;
        shifta = a;
        shiftb = b;
        result = 0;
        repeat (size)
          begin
            if( shiftb[1] )  // check if bit1 == 1
                res = res + shifta;
            shifta = shifta << 1;
            shiftb = shiftb >> 1;
          end
      end
endmodule
```


Looping, Cont'd

```
module count_ones( res, a);
   output [3:0] res; reg    [3:0] res;
   input  [7:0] a;

   always @(a)
     begin :count1s
          reg [7:0] tempa;
          res = 0; tempa = a;
          while(tempa) // while tempa != 0
            begin
             if( tempa[0] ) res = res + 1;
             tempa = tempa >> 1;
            end
     end
   endmodule
```


Parallel Blocks

- Statements executed concurrently
- Delay values for each statement are relative to the simulation when control enters block
- Control passed out of the block when the last time-ordered statement is executed

Order of statements is not important

Parallel block

```
fork
  #100 r = 100;
  #50 r = 50;
  #150 r = 150;
join
```

Equivalent sequential block assuming c changes every 50 units:

```
begin
  @c r = 50;
  @c r = 150;
  @c r = 200;
end
```


More Examples

- *areg* is loaded when both A and B occur in any order

```
begin
     fork
       @A;
       @B;
     join
     areg = breg;
end
```


Tasks

- Tasks provide a way to execute common procedures at different places
- They provide the means of breaking a large procedure into smaller ones
- They make it easy to debug and read the source code

Task Format

```
task proc_name;
      input a, b;
      inout c;
      output d, e;

     <<statement>>

endtask
```

This can be enabled by:

proc_name(v, w, x, y, z);

which performs the following assignment:
a = v ; b = w; c = x;

Following performed upon completion of the task:

x = c; y = d; z = e;

Functions

- Functions differ from tasks in the following ways:
  - They return a value to be used in an expression
  - They must have zero simulation time duration
  - They must have at least one input

D. Z. Pan 13. Introduction to Verilog 37

Function Format

```
function [7:0] func_name;
    input [15:0] a;

    begin
      <<statement>>;
      func_name = expression;
    end
  endfunction
```

To enable this:

```
new_address = b * func_name(old_address)
```

D. Z. Pan 13. Introduction to Verilog 38

Latches and Flip-Flops (Flops)

Latch is synthesized if you write:

```
always @(CLK)  begin
     if (CLK) begin
        LatchOut = LatchInput
     end
 end
```

Flop is synthesized with:

```
always @(posedge CLK)  begin
        LatchOut = LatchInput
 end
```

D. Z. Pan 13. Introduction to Verilog 39