Lecture 20: Hardware Description Languages & Logic Simulation

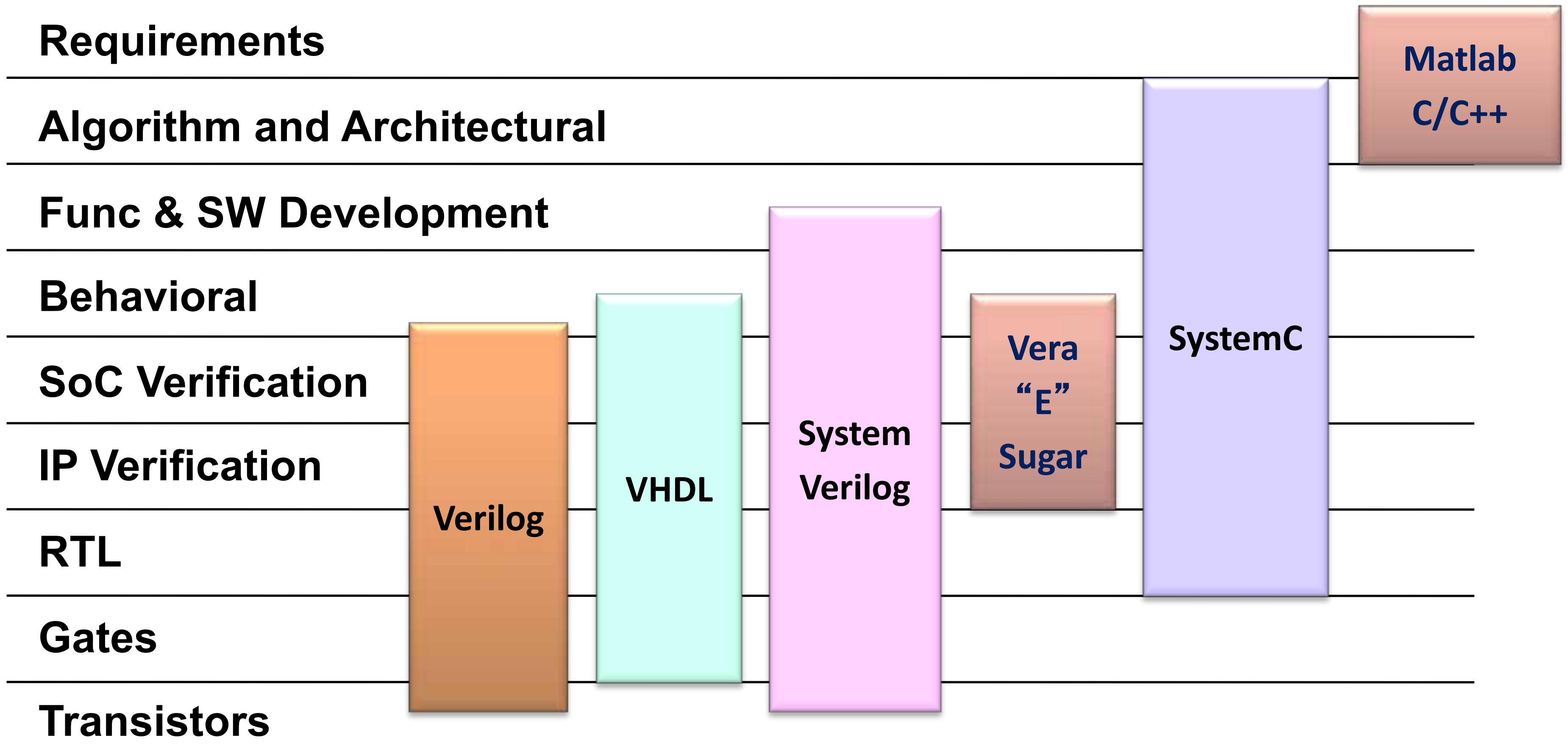
Mark McDermott
Electrical and Computer Engineering
The University of Texas at Austin

Agenda

- Verilog Hardware Description Language
- Brief Overview of Logic Synthesis

Verilog Hardware Description Language

Languages Usage



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 Phil Moorby at *Gateway Design Automation*, acquired by *Cadence*, now IEEE Standard 1364 (*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

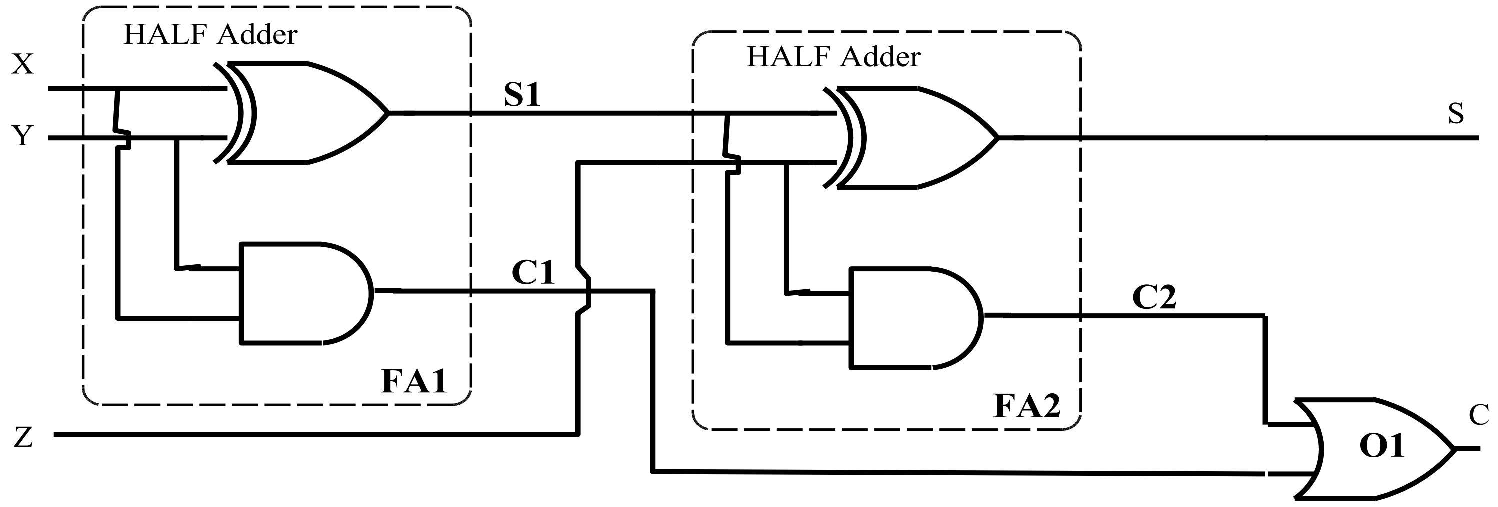
Verilog Modes

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. **Behavioral: 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,(
input   X, Y, Z;
output  S, C;

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

endmodule
```

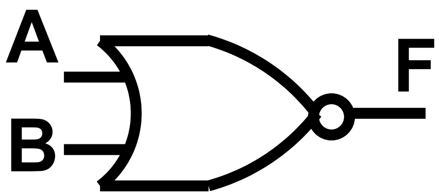
Specification of Gate-Level Model

- **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

```
xor  #(4,5)  (sum, a, b, cin);  // rise time=4 , fall time = 5
or   #5      (co, ab,bc,ac);    // rise time = fall time = 5
```

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)

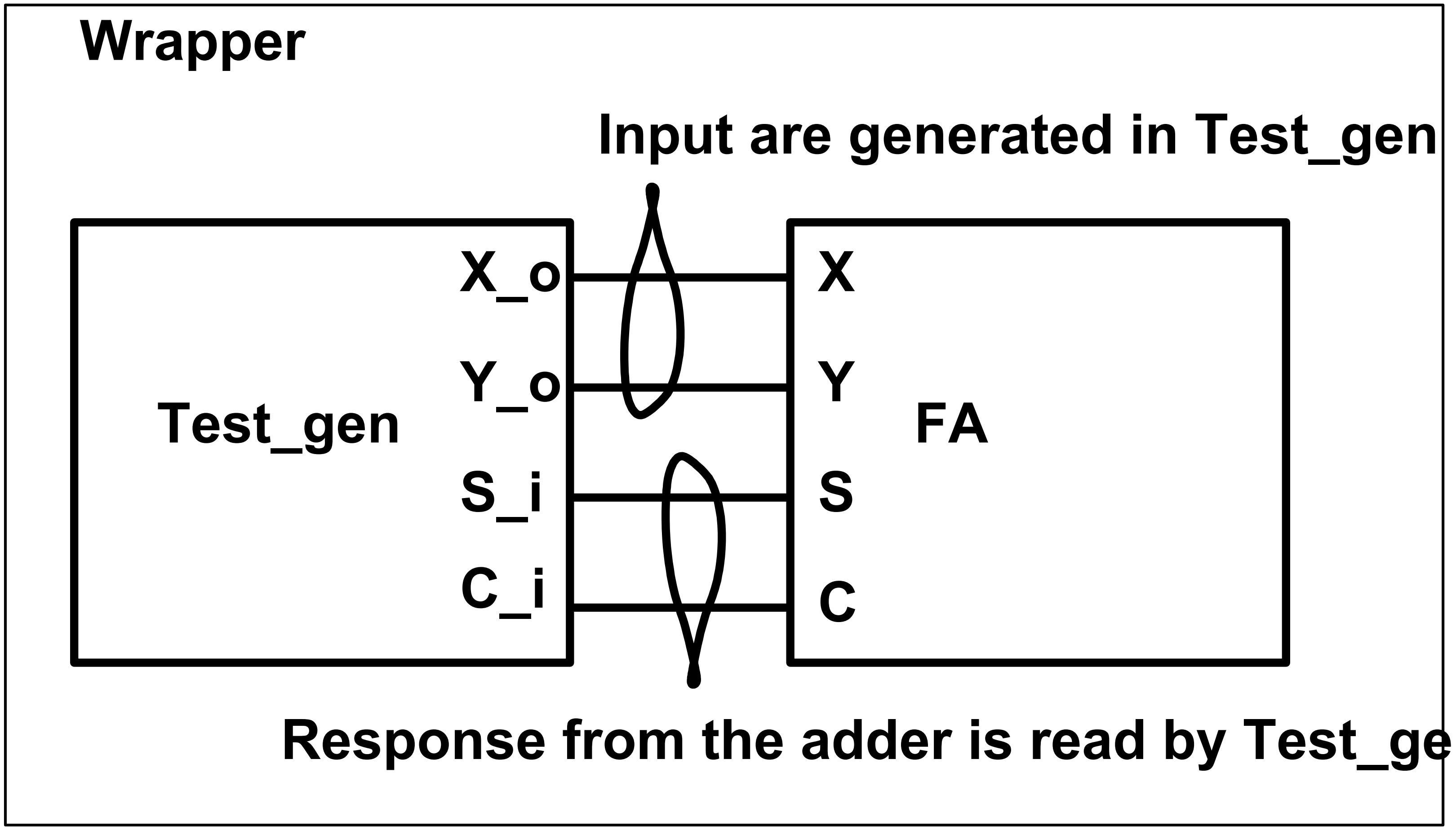


A \ B	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

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


Wrapper
Input are generated in Test_gen
Test_gen
X_o
Y_o
S_i
C_i
X
Y
S
C
FA
Response from the adder is read by Test_ge

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
```

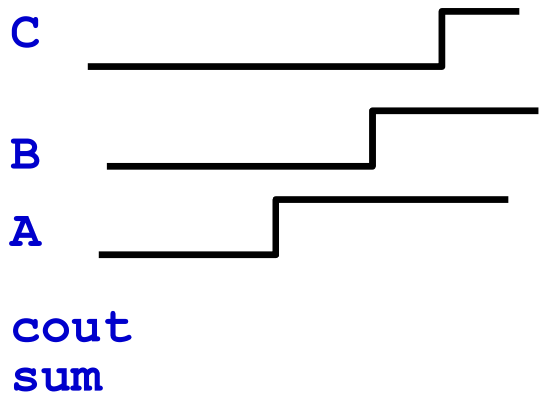
- initial
    tells Verilog to execute all statement within begin end once it starts
- monitor
    tells Verilog to monitor the list of variables and every time a var. changes

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 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

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
```

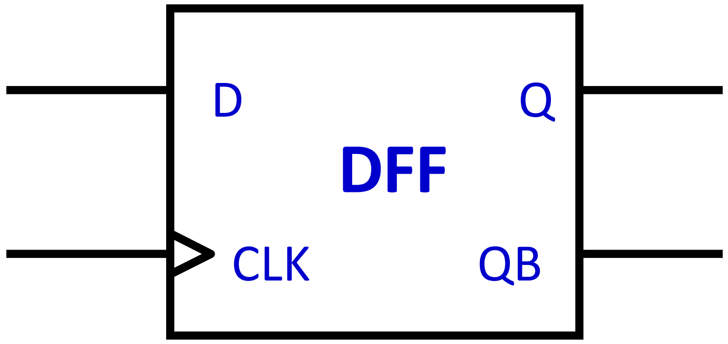
Edge Triggered DFF

```
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
```



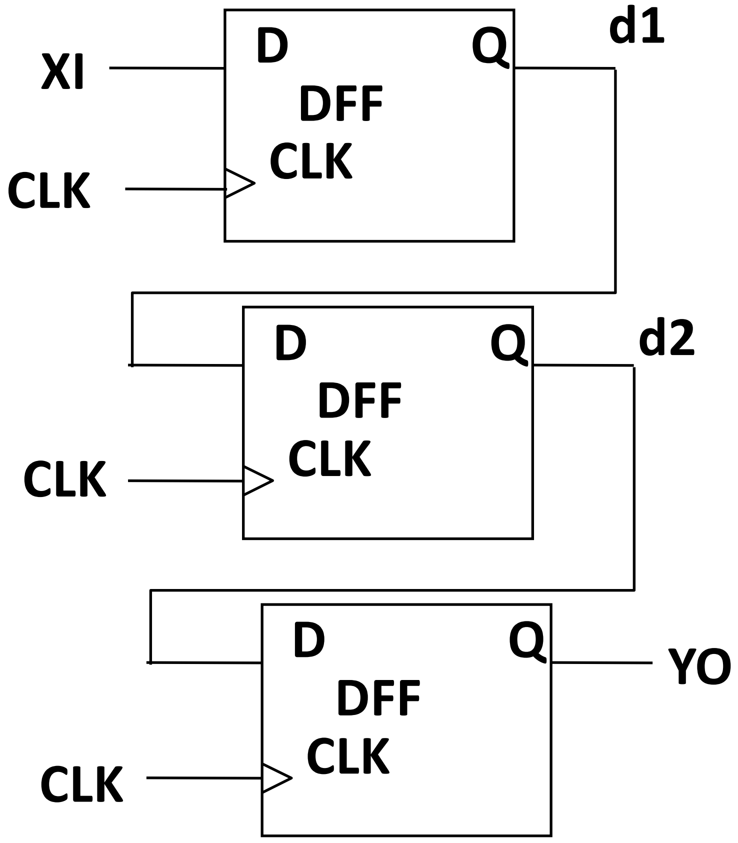
Shift Register

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
```



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

// Another way to do the same thing
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 = 5
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 using Continuous Assignments

```
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 (UDP)

```
primitive mux(mux, ctl, A, B);
     output mux;
     input  ctl, A, B;
         table
         // ctl   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
```

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;
        <<operation>>

endtask


// The task is called as so:
     proc_name(v, w, x, y, z);

// Performs the following assignments upon entry to the task:
     a = v ; b = w; c = x;

// Upon completion of the operation the following assignments are made:
     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

Function Format

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

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


  // The function is called as so:

  new_address = b * func_name(old_address)
```

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
```

Logic Synthesis

Brief History of Logic Synthesis

- 1960s: first work on automatic test pattern generation used for Boolean reasoning
  - D-Algorithm
- 1978: Formal Equivalence checking introduced at IBM in production for designing mainframe computers
  - SAS tool based on the DBA algorithm
- 1979: IBM introduced logic synthesis for gate array based main frame designed
  - LSS, next generation was BooleDozer
- End 1986: Synopsys founded
  - first product “remapper” between standard cell libraries
  - later extended to full blown RTL synthesis
- 1990s other synthesis companies enter the marker
  - Ambit, Compass, Synplicity. Magma, Monterey, ...

Synthesis in the Design Flow

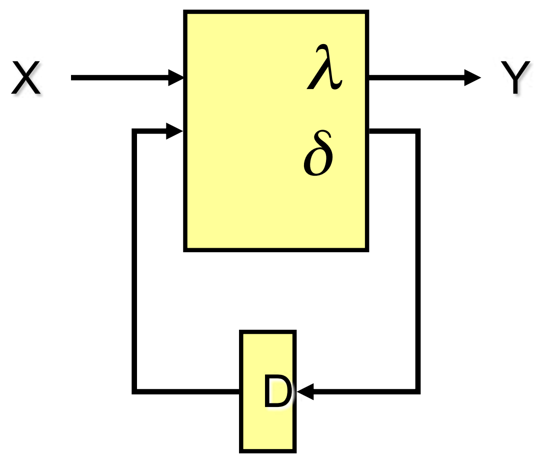
Designer	Tasks	Tools
Architect	Define Overall Chip C/RTL Model Initial Floorplan	Text Editor C Compiler
Logic Designer	Behavioral Simulation Logic Simulation Synthesis Datapath Schematics	RTL Simulator Synthesis Tools Timing Analyzer Power Estimator
Circuit Designer	Cell Libraries Circuit Schematics Circuit Simulation Megacell Blocks	Schematic Editor Circuit Simulator Router
Physical Designer	Layout and Floorplan Place and Route Parasitics Extraction DRC/LVS/ERC	Place/Route Tools Physical Design and Evaluation Tools

Logic Synthesis

What is 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.)

What is Logic Synthesis? (cont)



Given: Finite-State Machine F(X,Y,Z,λ,δ) where:

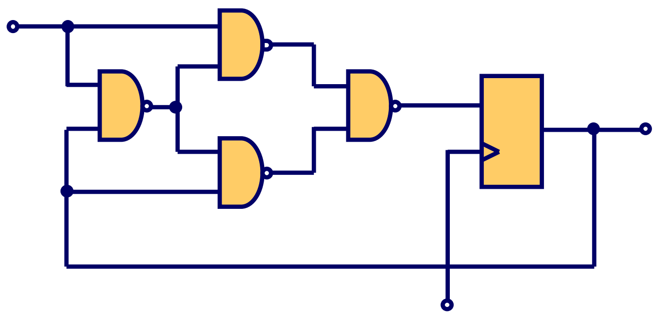
X: Input alphabet

Y: Output alphabet

Z: Set of internal states

λ : X x Z $\rightarrow$ Z (next state function)

δ : X x Z $\rightarrow$ Y (output function)



Target: Circuit C(G, W) where:

G: set of circuit components g$\in${Boolean gates, flip-flops, etc}

W: set of wires connecting G

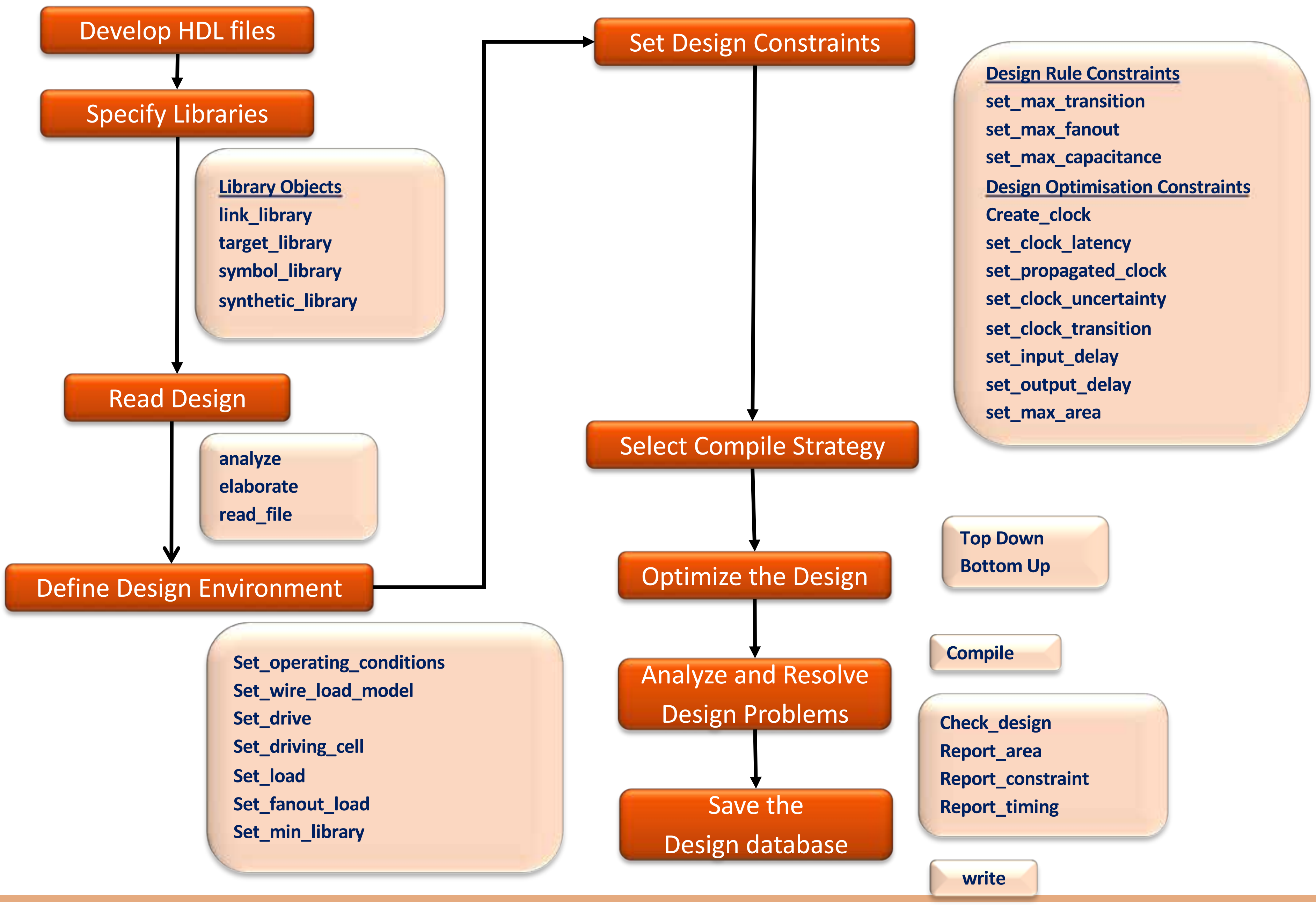
Objective Function for Synthesis

- **Minimize area**
  - in terms of literal count, cell count, register count, etc.
- **Minimize power**
  - in terms of switching activity in individual gates, deactivated circuit blocks, etc.
- **Maximize performance**
  - in terms of maximal clock frequency of synchronous systems, throughput for asynchronous systems
- **Any combination of the above**
  - combined with different weights
  - formulated as a constraint problem
    - "minimize area for a clock speed > 300MHz"
- **More global objectives**
  - feedback from layout
    - actual physical sizes, delays, placement and routing

Constraints on Synthesis

- **Given implementation style:**
  - two-level implementation (PLA, CAMs)
  - multi-level logic
  - FPGAs
- **Given performance requirements**
  - minimal clock speed requirement
  - minimal latency, throughput
- **Given cell library**
  - set of cells in standard cell library
  - fan-out constraints (maximum number of gates connected to another gate)
  - cell generators

Synthesis Flow



Library Objects
link_library
target_library
symbol_library
synthetic_library

analyze
elaborate
read_file

Set_operating_conditions
Set_wire_load_model
Set_drive
Set_driving_cell
Set_load
Set_fanout_load
Set_min_library

Design Rule Constraints
set_max_transition
set_max_fanout
set_max_capacitance
Design Optimisation Constraints
Create_clock
set_clock_latency
set_propagated_clock
set_clock_uncertainty
set_clock_transition
set_input_delay
set_output_delay
set_max_area

Top Down
Bottom Up

Compile

Check_design
Report_area
Report_constraint
Report_timing

write

References

- http://www.verilog.net/
- http://www.eda.org/
- http://www.verilogtutorial.info/
- http://www.asic-world.com/verilog/veritut.html

Logic Simulation

Courtesy of: dropzone.tamu.edu/~wshi/475/Logic_Simulation.ppt

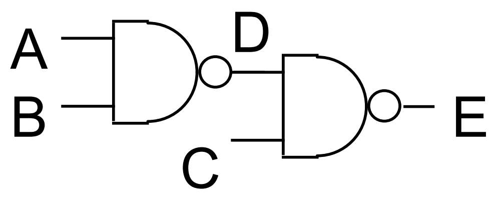
Logic Simulation

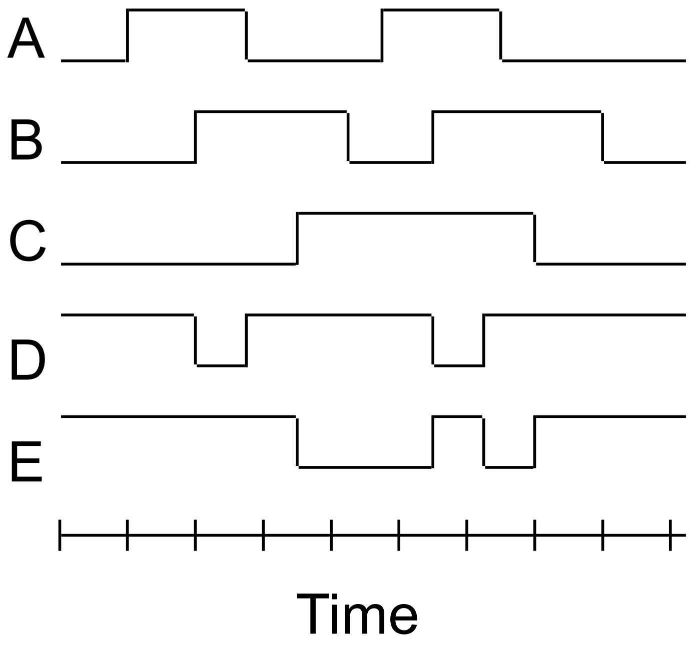
- **Outline**
  - Logic Simulation
  - Logic Design Description
  - Logic Models

- **Goal**
  - Understand logic simulation problem
  - Understand logic models

What is Logic Simulation?

- Simulate temporal behavior of logic design
- Logic design description
  - netlist, network
  - components
    - e.g. AND, OR, RAM, Pentium
  - component interconnections
- Logic models
  - component behavior
  - interconnect behavior
  - signal values
- Timing models
  - component behavior
  - interconnect behavior
  - signal delays



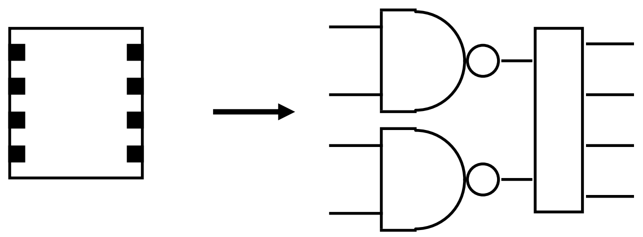


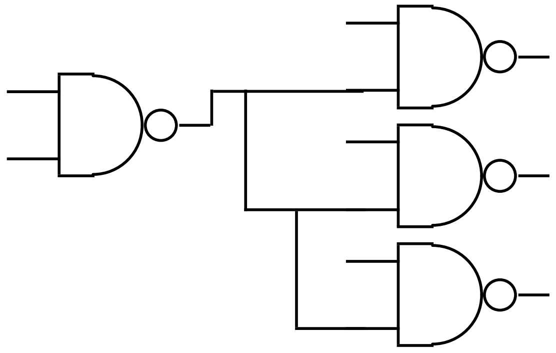
Logic Simulation Goals

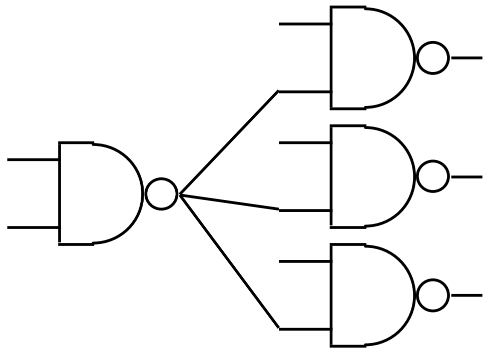
- Functional correctness
  - circuit does what you want
  - validate by using lots of input stimuli
- Performance
  - circuit runs fast enough
  - no hazards or races
  - validate using lots of stimuli
- Test generation
  - simulate faulty circuits
  - does input stimulus cause faulty output?

Logic Design Description

- **Components**
  - modules, cells,...
  - primitive - e.g. AND, OR, NOT
  - predefined - from library
    - functional behavior
    - timing behavior
  - composite - user-defined
    - subnetwork
    - hierarchy
- **Component connections**
  - wiring - nets
  - attachment points - pins, ports, terminals
  - can include wiring structure
    - fan-in, fan-out
    - parasitics

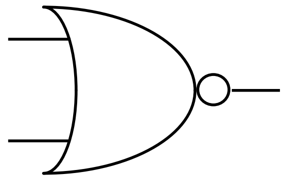


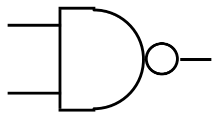




Logic Models

- **Logical Primitive**
  - Boolean logic operations
    - AND, OR, NAND, NOR, NOT, XOR, XNOR
  - often special-cased in simulator for speed
- **Behavioral Model**
  - finite state machine
    - outputs are function of inputs, next state
  - often supplied by model library vendors
    - implementation is secret
- **Subnetwork**
  - composed of primitives, behavioral models, subnetworks
  - use hierarchy, regularity in logic design

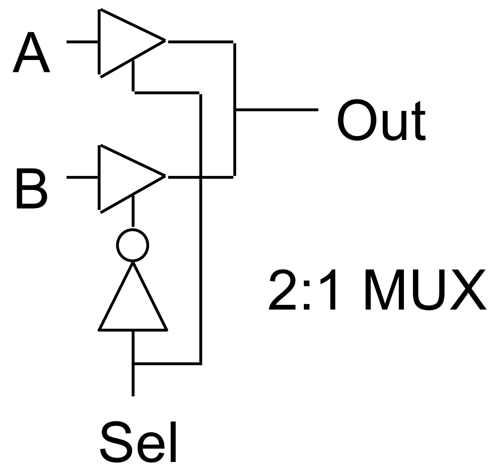


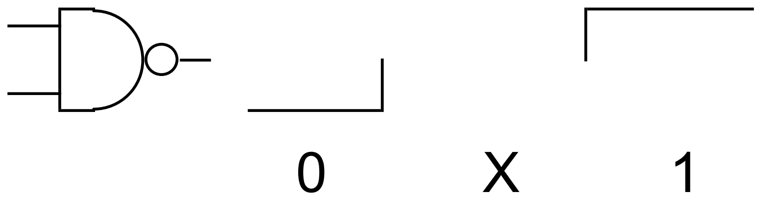


$C = f(A,B)$

Logic Values

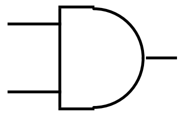
- Component output values - states
- 2-state simulation
  - 0 - logical 0
  - 1 - logical 1
- 3-state simulation
  - add X state
    - unknown, uninitialized, intermediate
- 5-state simulation
  - add rising, falling states
- Other states
  - Z - high impedance - for buses
  - U - unknown, but 0 or 1
    - X for intermediate value
  - D, D' - fault signals - for fault simulation
  - more states => closer to analog simulation





Logic Model Implementation

- **Truth table**
  - list of all input/output choices
  - fast - table lookup evaluation
    - use 0, 1, 2 for 3 states
  - impractical for many inputs, logic values
- **Latch**
  - truth table with knowledge of previous state
- **Logic equations**
  - can be compiled and executed fast
  - must support value system
- **Behavioral description**
  - HLL description of input/output relationship
    - hardware description language - Verilog, VHDL
    - general-purpose language
  - usually precompiled for speed



	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

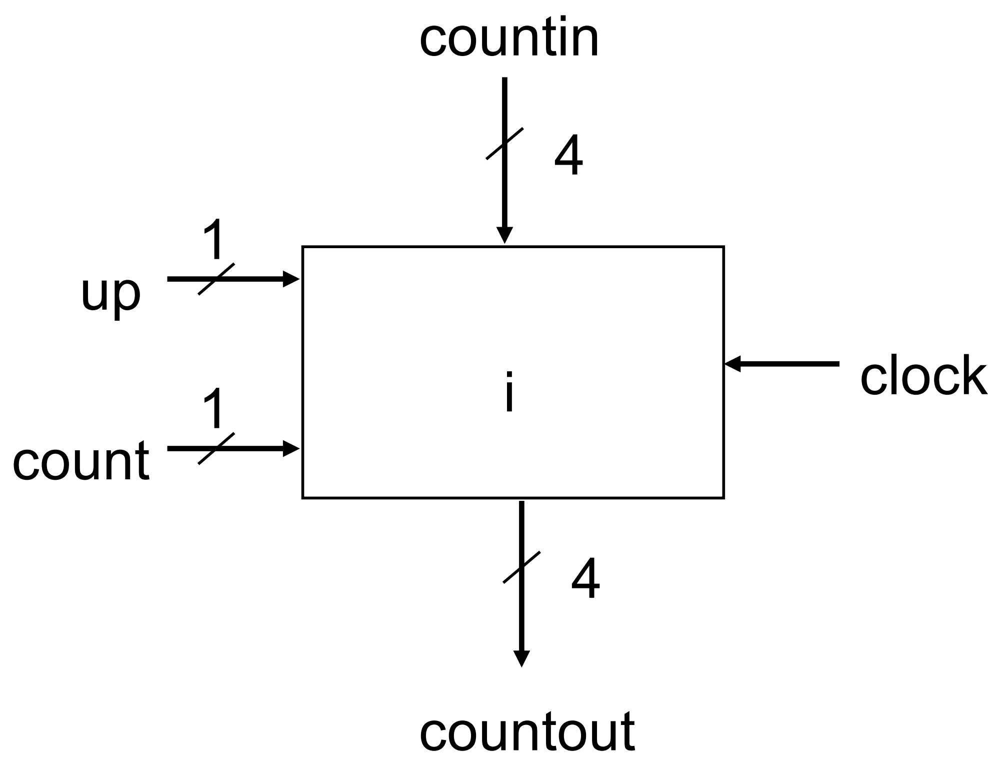
```
if (gate == and2)
    out = in1 && in2;
else if (gate == or2)
    out = in1 || in2;
```

Hardware Description Languages

- **Special-purpose languages**
  - special constructs for hardware description
  - timing, concurrency, bit vectors, etc.
  - can usually include structural description
  - examples
    - Verilog
    - VHDL
- **Standard programming languages**
  - add data types and objects for hardware description
  - examples
    - C
    - C++
    - Matlab

Example

- **4-bit Up/Down Counter**
  - 4-bit input "countin"
  - 4-bit output "countout"
  - up/down control "up"
  - count control "count"
  - internal state "i"
  - implied clock
- **Behavior**
  - up = 1 => count up
  - up = 0 => count down
  - count = 1 => count
  - count = 0 => load



Verilog Example

```
/* ----- Programmable 4-bit up/down counter ----- */

module counter(countin, countout, up, count, clk);

  input countin[4], up, count, clk;
  output countout[4];
  reg countout[4];

  always @(posedge clk)
    begin
      if (count == 0)
        countout = countin;
      else if (up == 1)
        countout = countout + 1;
      else
        countout = countout – 1;
    end
endmodule
```

C Example

```
/* ----- Programmable 4-bit up/down counter ----- */

unsigned int counter_private;

counter(unsigned int countin,
        unsigned int up,
          unsigned int count,
        unsigned int *countout)
{
    while (1) {
      if (count) {
          if (up) counter_private ++;
          else    counter_private --;
      else
          counter_private = countin;

      *countout = counter_private;
    }
}
```

HDL vs. Programming Language

- **HDL strengths**
  - concurrency
  - bit manipulations
  - module representations
  - timing constraints
  - structural representation support
- **Programming Language strengths**
  - data structures
  - language support
- **Mostly VHDL and Verilog today**
  - most vendors support both languages

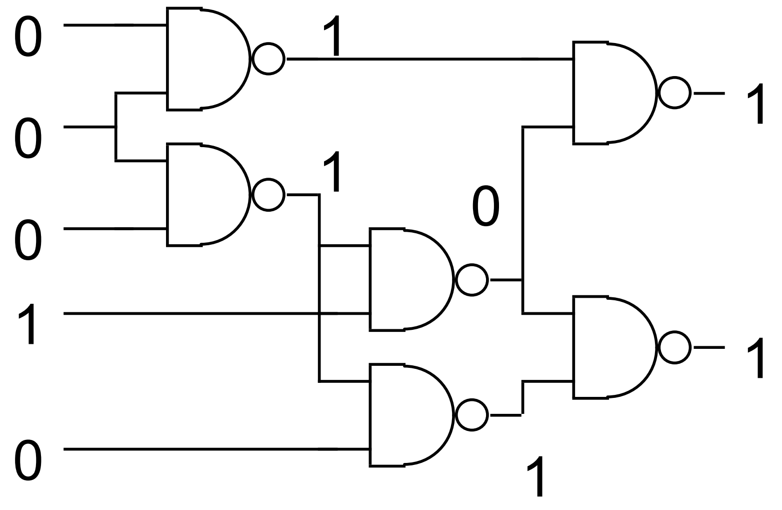
Timing

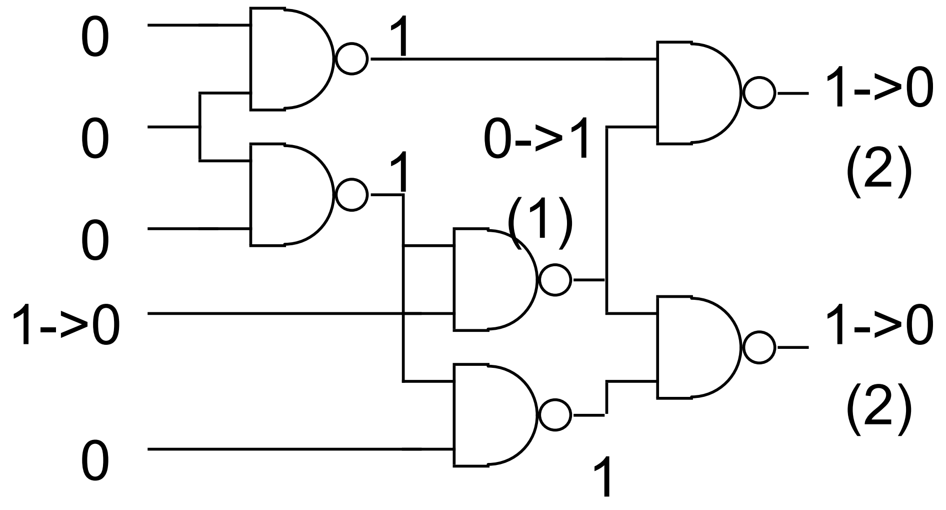
- **Outline**
  - Timing Models
  - Simulation Algorithms

- **Goal**
  - Understand timing models
  - Understand simulation algorithms

Component Timing Models

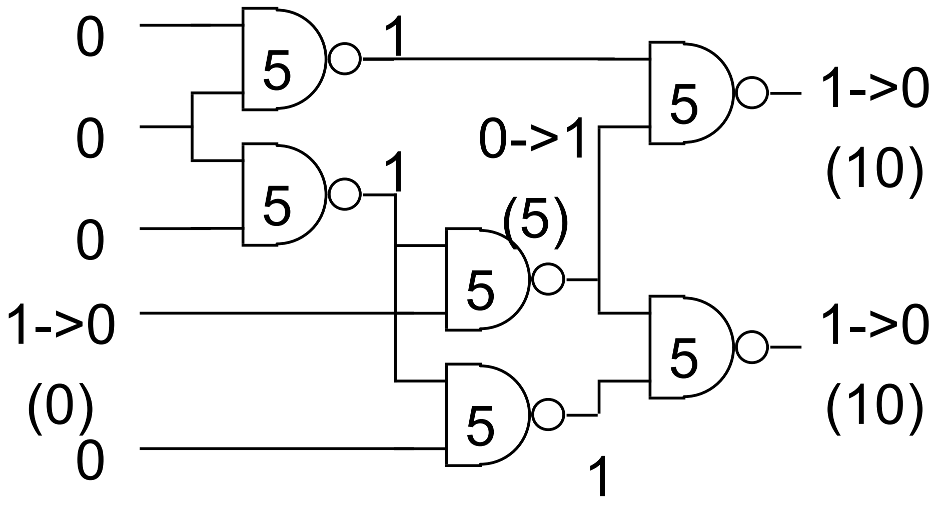
- **Zero delay**
  - no delay from gate inputs to outputs - a boolean equation
  - good when just considering functional behavior
    - e.g. combinational fault simulation
- **Unit delay**
  - gate delay is one unit
  - appropriate when all gates are same delay
    - e.g. gate array, ignores wiring load



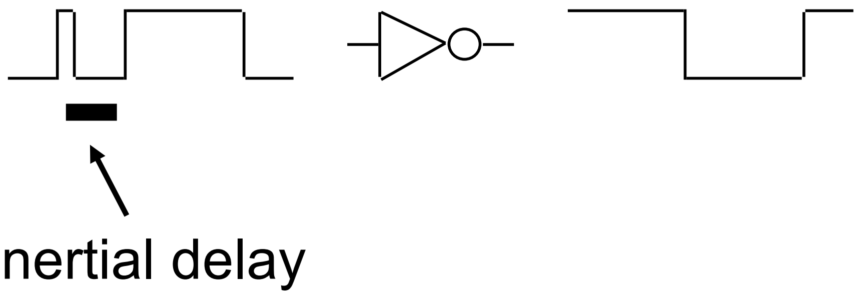


Component Timing Models

- **Propagation delay**
  - fixed delay from inputs to outputs
  - can vary with fan-out, output load
  - delay = time or multiple unit delays
  - varies among gate types
    - XOR vs. inverter
  - varies between gate instances
    - manufacturing variations
  - make output unknown for interval
    - rise/fall, min/max delay



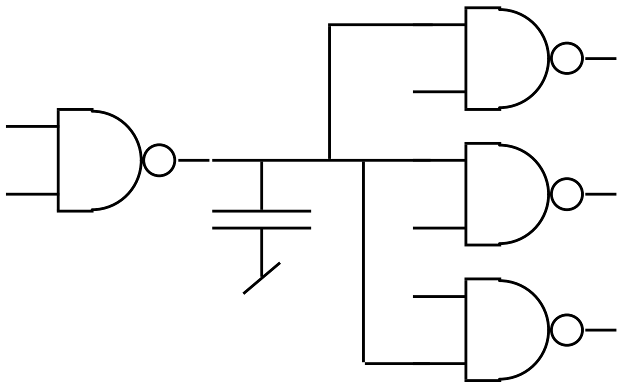
- **Inertial delay**
  - delay time before gate responds
  - input value must be present long enough for the gate to respond
  - gates are low-pass filters



Component Timing Models

- **Output Loading Delay**
  - gate delay is function of load it drives
  - wiring capacitance
  - type of gate
    - e.g. small vs. driver
  - number of fan-outs
  - types of fan-outs
  - depends on interconnect delay model
- **Compute prior to simulation**

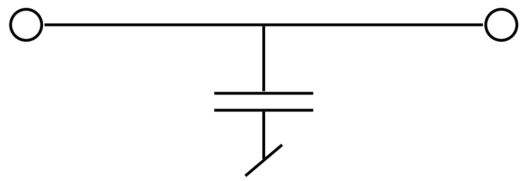
PropDelay = f(load)



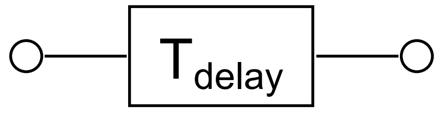
```
PropDelay = UnloadedDelay[gatetype]
          + C_L*LoadFactor[gatetype]
          + fanoutcnt*FanFactor[gatetype]
```

Interconnect Timing Models

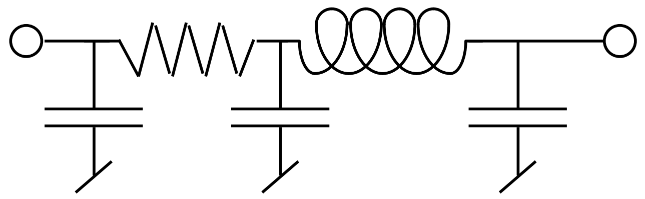
- Isochronic
  - zero delay
  - interconnect acts as capacitor
  - all delay assigned to driving gate



- Wave propagation
  - transmission line
  - fixed delay
    - function of wire “length”
  - can model as different propagation delay times for different fan-outs



- Signal diffusion
  - distributed RLC parasitics
  - inertial delay
  - usually combine with driver timing model



Simulation Algorithms

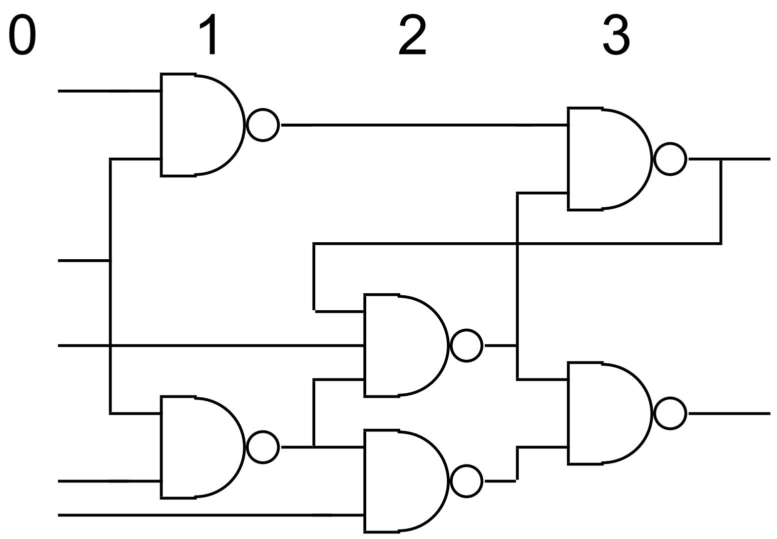
- **Compiled simulation**
  - logic equations compiled to code
  - execute code - fast
  - works well with zero and unit delay models
  - difficult with general timing models
  - no use of latency, multi-rate behavior
- **Event-driven simulation**
  - input changes cause evaluation of model, scheduling of output change event
    - use timing models to determine new event times
  - output change evaluated at scheduled time
  - advantage
    - real circuits have 10-30% activity
    - dormant parts of circuit are not evaluated

Compiled Simulation

- Algorithm
  - mark feedback paths
  - levelize circuit - topological sort
  - generate evaluation code
    - by level in circuit
  - compile and link with control and I/O
  - execute
- Issues
  - compilation time vs. execution time
    - use only on big problems
  - which gate inputs/outputs to watch
    - compilation can eliminate nodes
  - simple delay models only

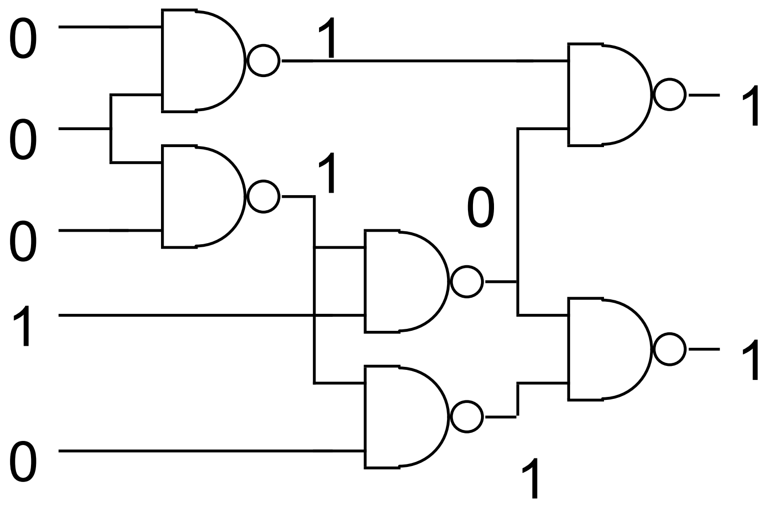
Circuit Level Assignment

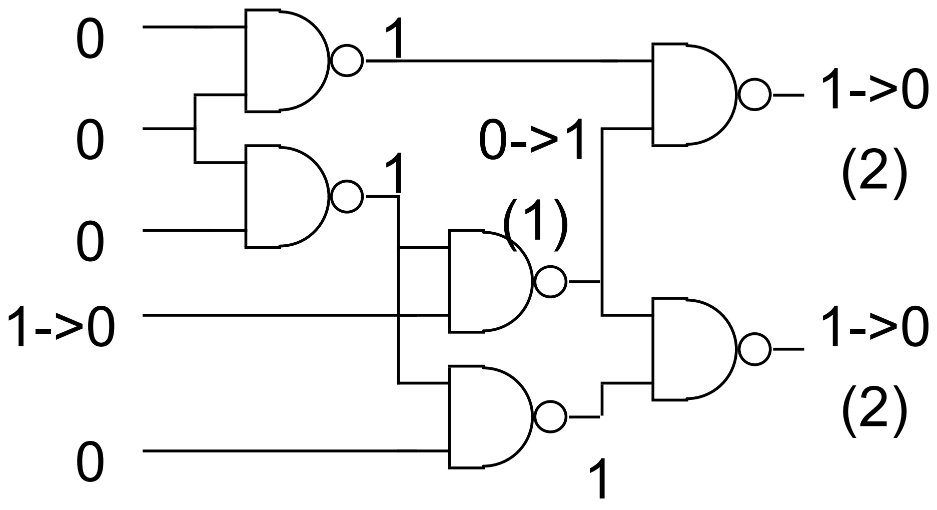
- **Topologically order circuit**
  - assign gate and its outputs to a level
  - primary inputs = level 0
  - for each gate
    - level = max(nonfeedback input levels) + 1
  - use breadth-first search
  - sort gates by level
- **Marking feedback paths**
  - feedback == cycle in directed graph
  - gates are vertices
  - gate outputs are directed edges
  - during breadth-first search
    - if hit visited node, then feedback



Zero and Unit Delay Simulation

- **Zero-delay simulation**
  - evaluate gates in levelized order
  - for each gate update output from inputs
    - inputs already computed from previous level
  - feedback oscillation improperly modeled
- **Unit-delay simulation**
  - evaluate gates in levelized order
  - for each gate update output from inputs
  - output time is max(controlling input change times)+1



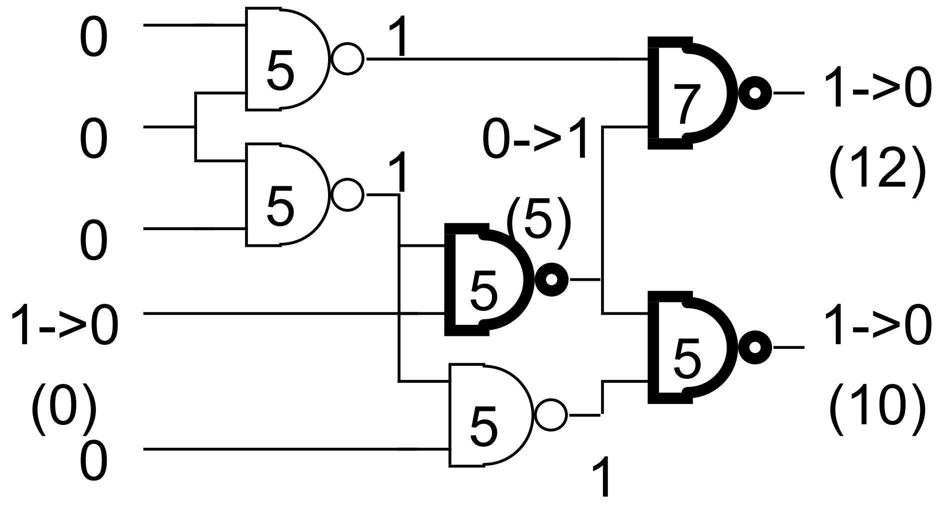


Event Driven Simulation

- **Outline**
  - Event-driven simulation algorithms
  - Event-driven simulation implementation
- **Goal**
  - Understand event-driven simulation
  - Understand simulator implementation
- **Reading**
  - Gate-Level Simulation by D'Abreu
  - Types of Simulators by Trimberger

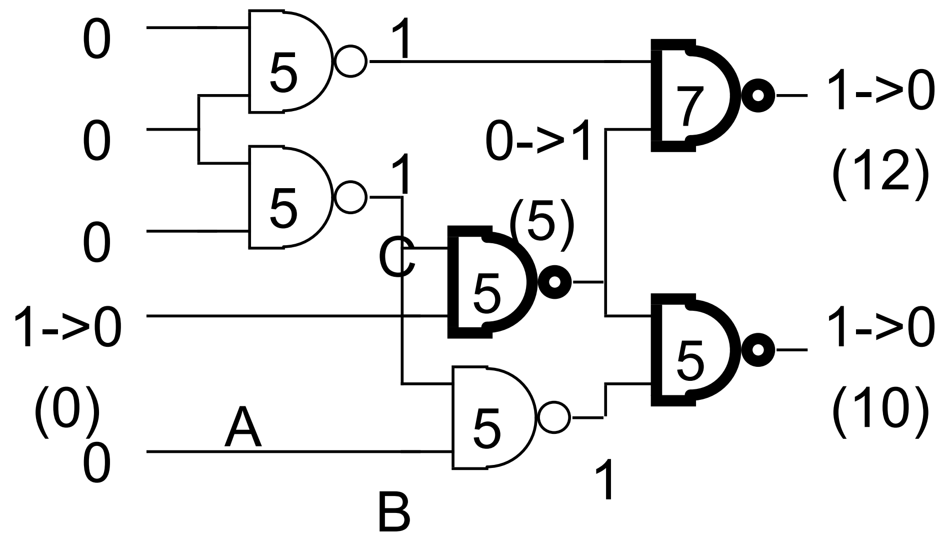
Event-Driven Logic Simulation

- **Evaluate gate when inputs change**
  - use logic model to compute new output value
  - use timing model to compute when output will change
- **Schedule an output change event**
  - store the event on a time-sorted event queue
- **Process events from the queue**
  - output change evaluated at scheduled time
  - causes new events to be scheduled



Event-Driven Logic Simulation

1. t=X: Schedule PI:1->0 at t=0
2. t=0: PI changes 1->0
   - Evaluate A, schedule A:0->1 at t=5
4. t=5: A changes 0->1
   - Evaluate B, schedule B:1->0 at t=10
   - Evaluate C, schedule C:1->0 at t=12
5. t=10: B changes 1->0, output
6. t=12: C changes 1->0, output



Simulation Algorithm

```
while (HaveEvents())
  event = NextEvent();                        /* time-sorted order */
  currenttime = event->time;                  /* update global time */
  event->gate->output = event->output         /* change gate output */
  print output if it is primary output;

  for (all gates g in event->gate fan-out list)
    newoutput = EvalGate(g);                  /* new gate output */
    newtime = currenttime + g->delay;         /* when it changes */
    ScheduleEvent(g, newoutput, newtime);
```

Simulator Initialization

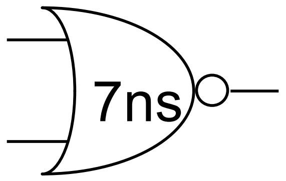
- Set all gate outputs to X (3-state logic)
- Start simulation from primary inputs
  - inputs from outside world
- 0/1 values are set as inputs propagate
- Problems
  - feedback and memory increase initialization time
  - some states may never initialize
- Solution
  - real circuits have initialization logic
  - reading before writing memory is an error

Event Scheduling

- Only schedule event on a gate output if it:
  - occurs after the last pending event and has a different value
    - otherwise creates useless work
  - occurs before first pending event
    - remove next event if same value, now obsolete
  - occurs between two events and is different than previous one
    - remove next event if same value, now obsolete
  - has different value than current output, and no pending events
    - otherwise creates useless work
- Note: typically 0, 1, or 2 pending events on a gate

A: 0->1 at 3ns

B: 1->0 at 3ns



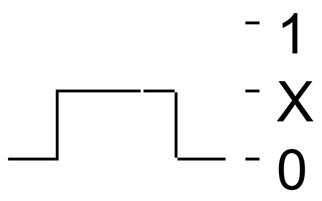
B first, then A:

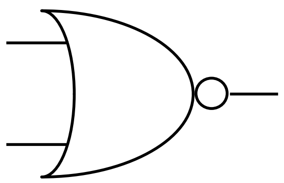
1 at 10ns - discard

0 at 10ns - keep

Event Scheduling

- **Note: cases 2 & 3 cannot happen for pure propagation delay model**
  - events always arrive in time-sorted order
  - new event must come >= last pending event
  - can happen for more complex timing models
- **Inertial delay**
  - remove pending event pair if new event caused them to be spaced < inertial delay and return to original value
  - for pure propagation model, just add time check in case 1 against last pending event
  - wait for all events at time T -don't delete until you are sure





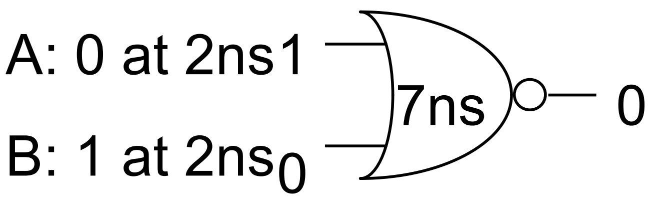
7ns prop. delay
3ns inertial delay

Pending: 1: 0 at 1ns
2: 1 at 5ns

New: 0 at 6ns - discard, remove #2
0 at 9ns - add to queue

Optimizations

- **Problem**
  - multiple events at time T can cause multiple gate evaluations
  - up to N evaluations for N inputs
- **Solution**
  - for all events at time T put gates to be evaluated on a list
  - evaluate all gates at same time, scheduling events
- **Problem**
  - multiple events at time T cause multiple printed output
- **Solution**
  - wait until time advances to print primary outputs
  - print them only if any primary outputs changed



1. Evaluate gate using event A
2. Schedule event C: 1 at 9ns
3. Evaluate gate using event B
4. Delete event C

Ambiguity – Is Your Fault

- **Verilog Example**

```
always @(posedge clk) begin
        a = 0;
end

always @(posedge clk) begin
        a = 1;
end
```

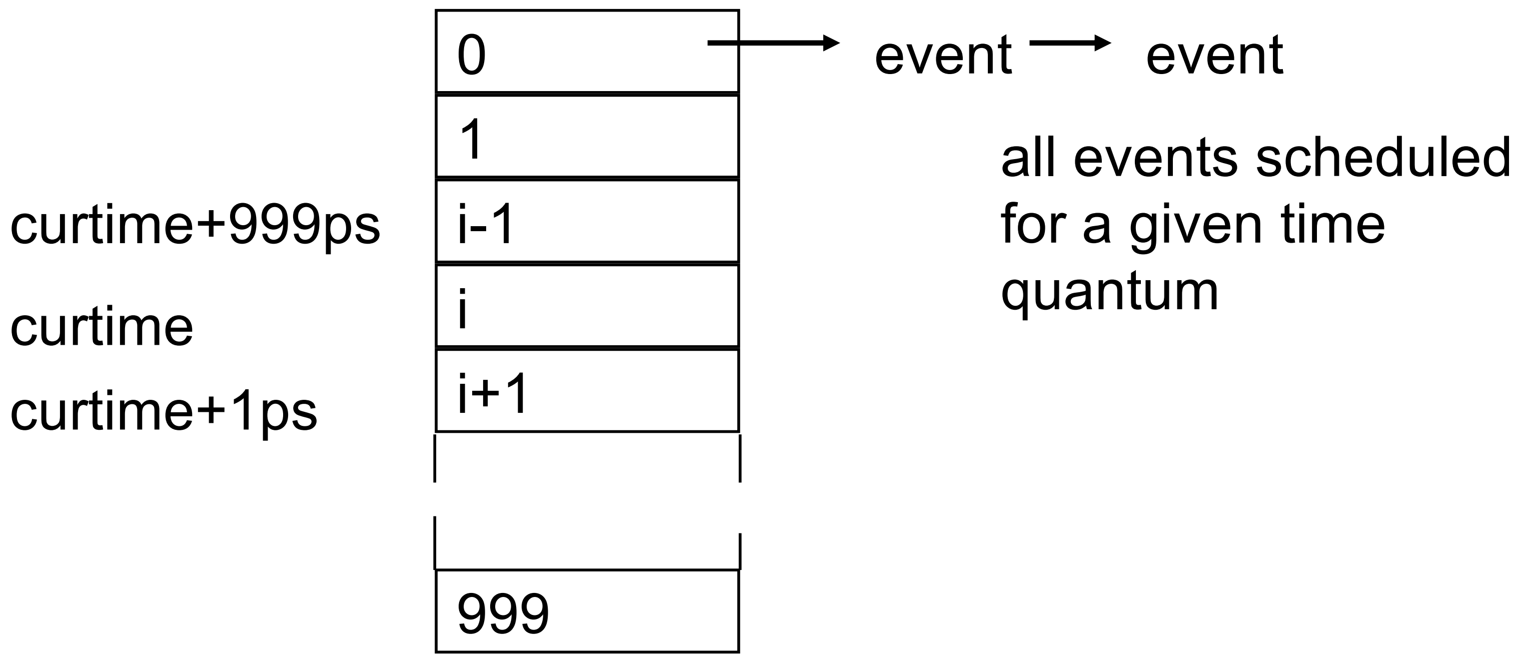
Simulation Control

- Visualization
  - timing diagrams
  - back annotation of schematic
- Inputs
  - timing diagrams
  - vectors - especially for synchronous circuits
- Probing
  - examine events at a node
  - straightforward for event-driven simulation
    - mark each node to save value
    - can force node to value
  - must make a primary output for compiled simulation
- Control
  - stop and start time sequence
  - insert “breakpoint” events, like debugger

Event Queue Implementation

- Events must be processed in time order
  - event queue must sort by time
- Must be able to delete events
  - cancel obsolete events on a gate
- Implementations
  - priority queue
    - O(logN) time to insert/delete for N-item queue
    - many implementations - AVL tree, heap, etc.
    - problem: N is often large
  - bucket queue - time wheel
    - divide time into time quanta Q, e.g. 1ps
    - circular buffer of N entries 1 quantum in size
    - can store events Q*N into future
    - events beyond buffer go into unsorted far list
    - O(1) time to insert, nearly so for delete

Time Wheel



time wheel of *time quantums*

Wheel only needs to be big enough to hold most variation in gate delays

Time Wheel Operation

- **Insertion**

```
if (eventtime - curtime >= WHEELLENGTH*quantum)
  insert event into far list
else
  insert at wheel[eventtime % (WHEELLENGTH*quantum)]
```

- **Deletion**

```
i = curtime % (WHEELLENGTH*quantum)
while (wheel[i] == NULL)
  if (i == WHEELLENGTH-1)
    i = 0; timebase += (WHEELLENGTH*quantum);
    for all events in far list
      if (eventtime - timebase < WHEELLENGTH*quantum) insert
        event into wheel
  else i++
remove first event from wheel[i] list and return it
```

Fault Simulation

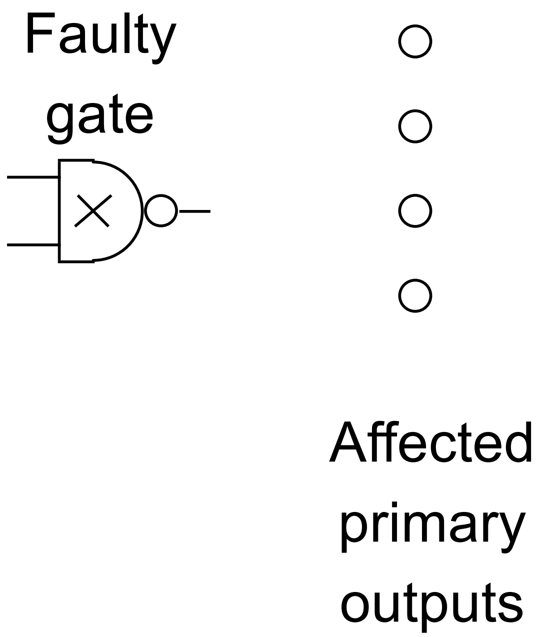
- **Outline**
  - Fault Simulation
  - Fault Models
  - Parallel Fault Simulation
  - Concurrent Fault Simulation

- **Goal**
  - Understand fault simulation problem
  - Understand fault simulation methods

Fault Simulation

- Simulate behavior of faulty logic design
  - inject faults into logic circuit
  - run logic simulation to determine faulty behavior

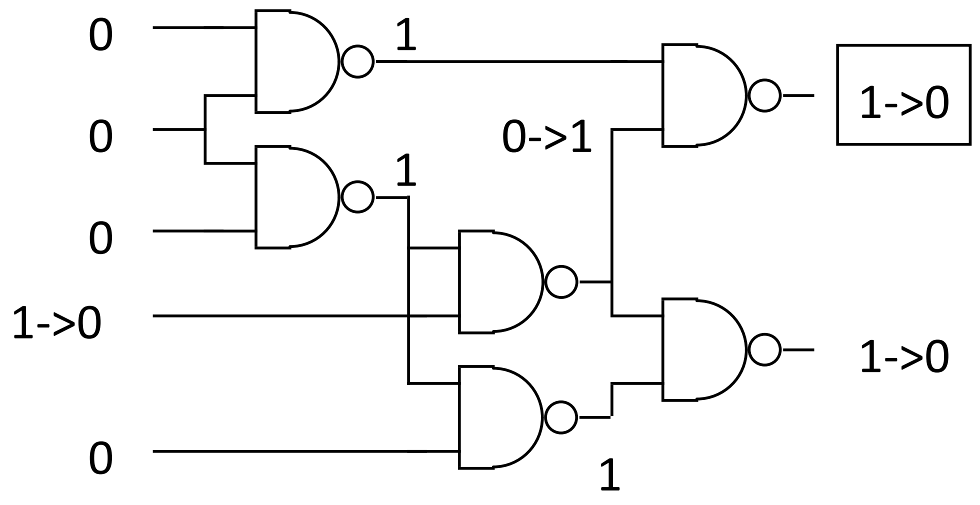
- Goals
  - test generation
    - does vector cause fault to be detected at primary outputs?
  - fault coverage
    - what fraction of faults are detected by test set?
  - fault analysis
    - does fault modify circuit behavior?
    - what parts of circuit are affected?
    - use in defect and fault tolerant design



Example

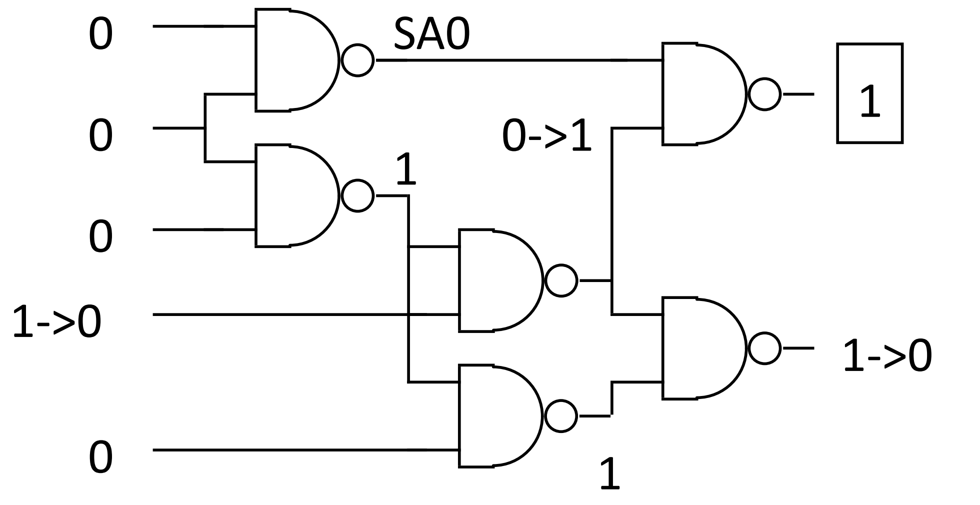
Good

Circuit



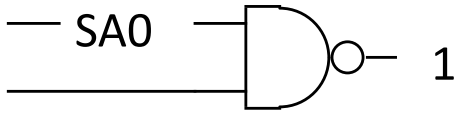
Faulty

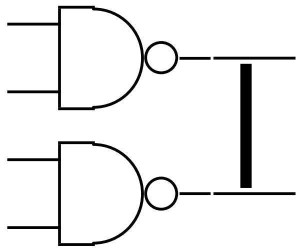
Circuit

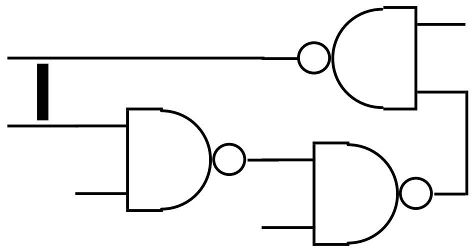


Fault Models

- **Fault model**
  - logic-level abstraction of circuit fault
  - permanent faults caused during manufacturing
  - transient and permanent faults during operation
- **Stuck-at model**
  - gate inputs or outputs stuck at 0 or 1
  - developed for TTL, but still used
  - convenient
    - logical values, single node influence
- **Bridging model**
  - short between two or more nodes
  - logical behavior difficult
    - nodes fight, both affected
    - feedback and oscillation
  - cannot consider all possible node pairs

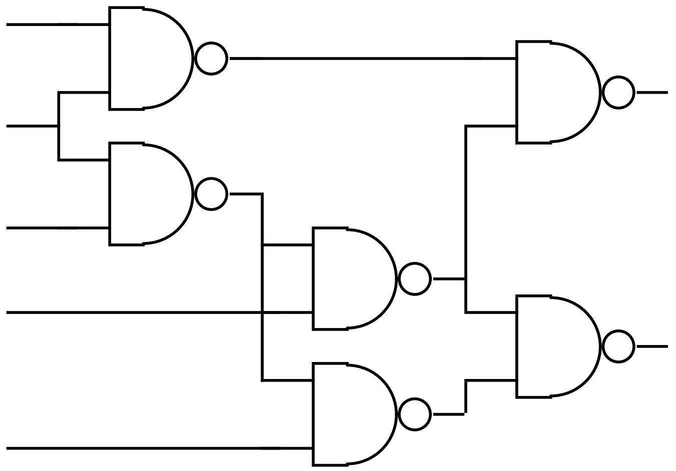






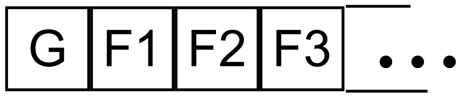
Fault Simulation

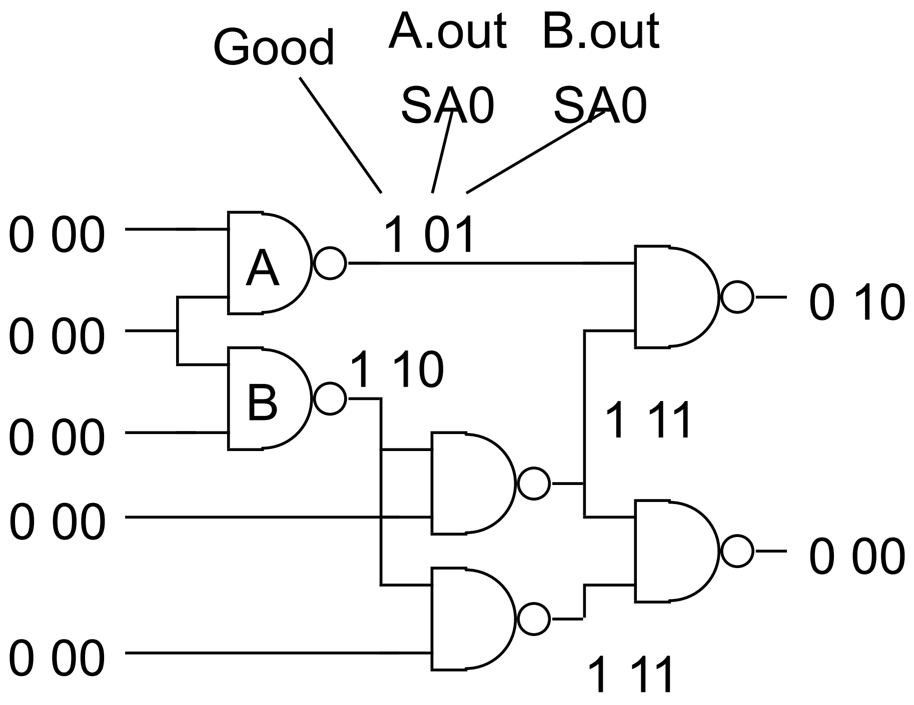
- **Problem**
  - fault simulation can be expensive
  - Example: C7552
    - 7552 possible faults to simulate
    - 243 test vectors
    - 1.8M test vector simulations
- **Observation**
  - stuck-at faults do not change netlist structure
  - only node values and gate functions
- **Observation**
  - faulty behavior only occurs in gate's cone of influence
    - transitive closure of fan-out list



Parallel Fault Simulation

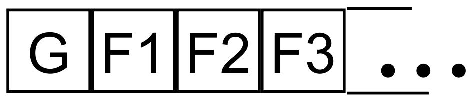
- Simulate all faults in parallel on same netlist structure
  - each gate output "value" is really a vector of values
  - good output and all faulty outputs





Parallel Fault Simulation

- Compute gate output with bit-wise operations
  - AND, OR, NOT operate on all bits in data word
  - each bit is a separate good or faulty circuit
  - execute 32-64 faults in parallel
  - very high parallelism on MPP
    - 64k faults in parallel on CM-2
- Problems
  - 0, 1, X requires 2 bits to represent
    - choose right encodings
    - 0 = 00
    - 1 = 11
    - X = 10
  - not all functions available as opcodes
    - use sequence of bitwise ops



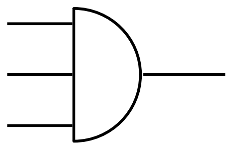
```
1   1   0   0
bitwise AND
1   0   1   0
      =
1   0   0   0
```

```
00   00   00   11   11   0
       bitwise AND
00   11   01   11   01   0
            =
00   00   00   11   01   0
```

Concurrent Fault Simulation

- Only track differences in fault cone of influence
  - cone can be very small
- Only propagate value differences
  - behavior differences caused by faults
  - for 3-value logic, can only be 3 possible behaviors at gate output
    - faulty and good behavior are same at a gate output
    - several faulty behaviors are same at a gate output
  - several faulty netlist behaviors are collapsed to one behavior

Good	F79	F383	F993
0	0	[1]	0
1	[0]	1	[0]
1	1	1	1



Good	F79	F383	F993
0	0	[1]	0

$\longrightarrow$

Good, F79, F993	F383
0	[1]

in1: 0 - Good, F79, F993; [1] - F383

in2: 1 - Good, F383; [0] - F79, F993 $\longrightarrow$ intersect 6 lists to get 2 lists on output $\nearrow$

in3: 1 - Good, F79, F383, F993

Parallel Pattern Single-Fault Simulation

- **Apply test patterns to fault until detected**
  - one fault at a time
  - apply test patterns with parallel simulation
- **Advantages**
  - up to 400X faster since most faults detected in first few vectors
  - avoid list overhead of concurrent fault simulation