Lecture 6
Verilog Language 2

Outline

- Verilog Language
  - Examples of three Verilog styles
  - Procedure Statement
  - Simulation

Important Verilog Constructs

- Gate-level primitives
  - Built-in
  - User defined
- Module Instantiation Statement
  - Used in Structural Style code

  Structural style example: and g1(e, a, b)
- ASSIGN Statement

  Example: assign x = e ^y
  - Concurrent signal assignment

    Concurrent = happen at the same time (combinational
  - Used in both

    In both structural and gate-level
- INITIAL Statement (Block)
  - Used in Behavioral Style – for simulation purposes
- ALWAYS Statement (Block)
  - Used in Behavioral Style

Built-in Gate Level Primitives

- The following are provided as built-in:

and	nand	logical AND/NAND
or	nor	logical OR/NOR
xor	xnor	logical XOR/XNOR
buf	not	buffer/inverter
Bufif0	notif0	Tristate with low enable
bifif1	notif1	Tristate with high enable

Tristate buffers are allowed to connect to one single wire
Buffer require a control enabled signal

User Defined Primitive

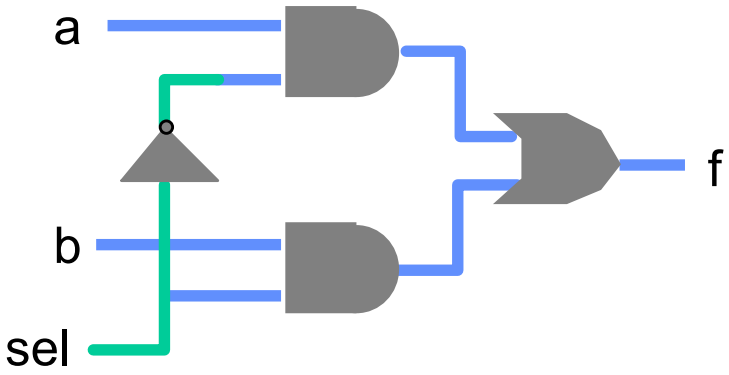
```
Primitive my_own_gate(out, a, b, c);
output out;
input a, b, c;
table
  00? : 1;
  000 : 0;
  ?00 : 0;
  11? : 1;
  1?1 : 1;
  ?11 : 1;
endtable
endprimitive
```

? : Don't Care

Module Instantiation for Structural Models

- Structural models
  - Are built from gate primitives and/or other modules
  - They describe the circuit using logic gates — much as you would see in an implementation of a circuit.
- Identify
  - Gate instances, wire names

Module Instantiation Using Primitives




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

      wire      f1, f2,nsel;
      and       g1 (f1, a, nsel),
                g2 (f2, b, sel);
      or        g3 (f, f1, f2);
      not       g4 (nsel, sel);
endmodule
```

Module Instantiation Using Other Modules Previously Defined

- There is a Flip Flop module with clock, data, q, qBar, reset, preset ports:
- Note the explicit matching of ports. Some ports can be left unused.
- Implicit listing of connections is also allowed, following the convention of position:
  - **d_ff d_ff_instance1(clk, d[0],q[0], , , ,);**

//qBar, reset, and preset ports are left unused

```
module d_ff_two(clk,d,q);
        input clk;
        input [1:0] d;
        output [1:0] q;
        d_ff d_ff_instance1(.clk (clk), .d  (d[0]), .q  (q[0]));
        d_ff d_ff_instance2(.clk (clk), .d  (d[1]), .q  (q[1]));

endmodule
```

ASSIGN (Concurrent) Statement

- Operation happens concurrently, ordering does not matter;
- Used for combinational logic
  - Structural Primitive/Module Instantiations could also be considered as a type of concurrent statement
- Continuous assignments: used in RTL design

```
assign e=A & B;
```

Continuous Assignment

```
wire [8:0] sum;
wire [7:0] a, b;
wire carryin;

assign sum = a + b + carryin;
```

Define bus widths

Continuous assignment: permanently sets the value of sum to be a+b+carryin
Recomputed when a, b, or carryin changes

Describing Combinational Logic

```
module mux21 (mux_out, in1, in2, sel)
input in1, in2, sel;
output  mux_out;

assign mux_out = (sel==0) ? in0: in1;

endmodule
```

```
module some_logic (d1_out, d2_out, in1, in2)
input [3:0] in1, in2;
output [3:0] d1_out, d2_out;

assign d1_out = in1 & in2;
assign d2_out = ~in1 | in2;

endmodule
```

Procedural Statement

- Procedural Statements
  - Used to describe sequential circuits
  - Define “Sequential operation”, order matters
- Two main code structures
  - Initial Block
  - Always Block

Initial and Always Blocks

- Basic components for behavioral modeling

```
initial
  begin
    … imperative statements
…
  end
```

Runs when simulation starts
Terminates when control reaches the end
Good for providing stimulus

```
always
  begin
    … imperative statements
…
  end
```

Runs when simulation starts
Restarts when control reaches the end
Good for modeling/specifying hardware

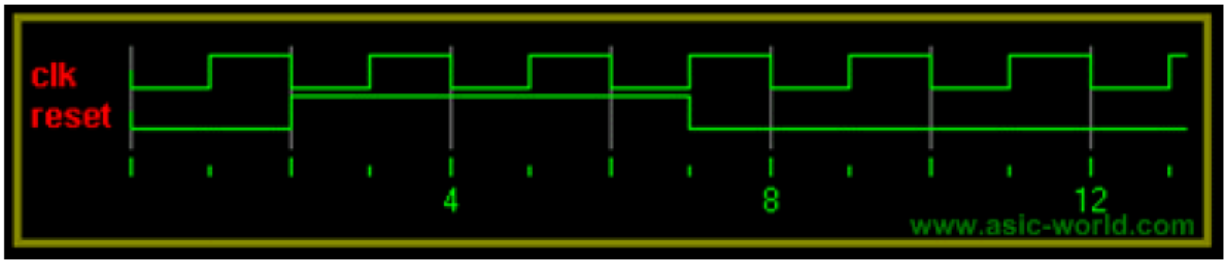
Initial Statement

- Inserted into code to support simulation
  - Use it to define a sequence of operations to set up the initial conditions of signals
- By default executed once **when simulation starts**
- **Terminates when last statement in the block is reached**
- Not synthesizable
- Can also use multiple initial statements for events operating in parallel

Initial Examples

```
module clk_gen();
reg clk, reset;
initial begin
        clk = 0;
        reset = 0;
        #2 reset = 1;
        #5 reset = 0;
End
always
begin
#1 clk = !clk;
end
```

```
module stimulus_gen;
reg a, b;
initial begin
        a = 0;
        b = 0;
        #5 a = 1;
        #5 b = 1;
        #5 a= 0;
         #5 b = 0;
end
endmodule
```



Always Statement

- One of the most important constructs in Verilog to describe behavior
- Two main places to use
  - Define an iterative sequential event
    - Used in a testbench
  - Define sequential circuits, e.g. flip-flops
- Can also describe combinational logic, if used carefully in limited context

Always for Generating Repetitive Events

```
module testbench ()
initial begin
        clk= 0;
        d = 0;
end

always begin
        #5 clk= ~clk;
end
always begin
        #10 d = ~d;
end
endmodule
```

Only execute once

Uses a **Delay Statement - #5 -** to control timing of when the always block will restart

Executes every 5ns (Creates a clock with 10ns period)

Executes every 10ns

Always for Sequential Logic

- Sequential logic only updates values upon clock transition
  - Described using the sensitivity list feature of the always statement
  - always statement restarts upon the described event statement in the sensitivity list

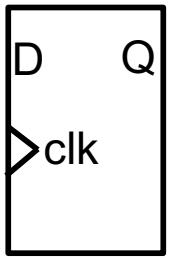
Three main ways to list:

```
always @(posedge clk) //special construct for sequential circuits
always @(signal1, signal2) //explicit list of signals
always @(*) Wildcard
```

Always for Sequential Logic

Simple flip-flop behavior

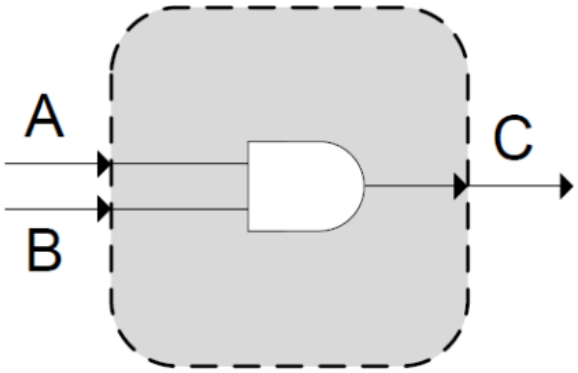
```
module dff(D, clk, Q);
        input D, clk;
        output reg Q;
        always @(posedge clk)
                Q = D;
endmodule
```



Always for combinational logic

```
always @(A, B) begin
        C = A & B;
end
```

Sensitivity list
List all signals you want to be involved

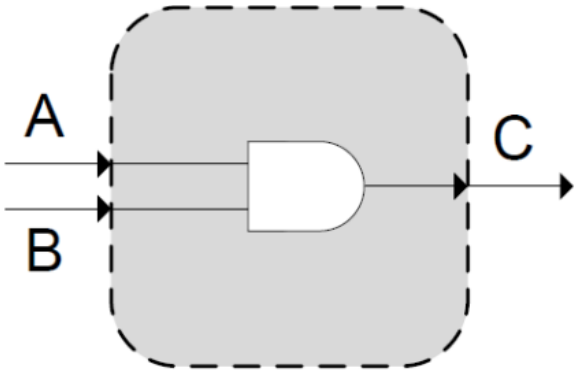


AND gate

Always for combinational logic

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

Sensitivity list
Wildcard Statement *
Compiler identifies all
Signals that are “read”



AND gate

Procedural Assignment

- Inside an initial or always block:

```
sum = a + b + cin;
```

- Similar to the **assign** statement, but functions within the sequential execution context
- RHS evaluated and assigned to LHS before next statement executes
- RHS may contain wires and regs
  - Two possible sources for data
- LHS must be a reg

Blocking vs. Non-blocking Assignments

- Verilog has two types of procedural assignments:

Blocking Assignment – Non-Blocking Assignment

C = A & B; //Simple blocking assignment

// C updated immediately

C = **#Time** A & B; //Blocking assignment with delay

//Execution halted for "Time" and then C updated

Blocking vs. Non-blocking Assignments

- Fundamental problem:
  - In a synchronous sequential circuit, all flip-flops sample simultaneously as they wake up with the clock tick
  - Assignment inside the always @(posedge clk) block run in some undefined sequence

Blocking vs. Non-blocking Assignments

- Non-Blocking Assignment

```
always @(posedge clk)
C  <= A & B;  //Non-blocking assignment
D <= !C;

// Right Hand Side will be evaluated, but the value of C will not be
// updated. Execution will move onto the next statement (it will use
// C's old value) C is updated before the next activation on the
// always block (i.e., by next clock tick)
```

Interaction Among Blocking Statements

```
reg d1, d2, d3, d4;

always @(posedge clk) d2 = d1;
always @(posedge clk) d3 = d2;
always @(posedge clk) d4 = d3;

//These run in some order, but it
// is not deterministic
//will d3 take on d2's old value of
// its new value as updated by d1?
```

```
reg d1, d2, d3, d4;

always @(posedge clk)
d1= ext_in;
d2 = d1;
d3 = d2;
d4 = d3;

//all registers contain same
// value (ext_in) each clock tick
```

Interaction among Non-Blocking Statements

- A sequence of non-blocking assignments do not communicate

```
a = 1;
b = a;
c = b;
```

Blocking assignments:
a = b = c = 1

```
a <= 1;
b <= a;
c <= b;
```

Non-blocking assignments:
a = 1
b = old value of a
c = old value of b

Non-blocking Assignments

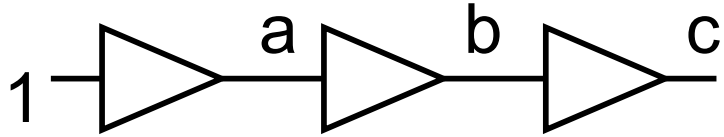
```
reg d1, d2, d3, d4;
always @(posedge clk) d1 <= ext_in;
always @(posedge clk) d2 <= d1;
always @(posedge clk) d3 <= d2;
always @(posedge clk) d4 <= d3;
```

Nonblocking rule: RHS evaluated when assignment runs

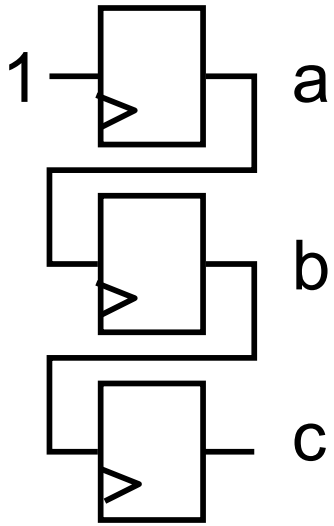
LHS updated only after all events for the current instant have run

Blocking looks like chain of wires, Non-blocking Looks Like Chain of Registers

```
a = 1;
b = a;
c = b;
```




```
a <= 1;
b <= a;
c <= b;
```



Imperative Statements:
If Statement
Case Statement

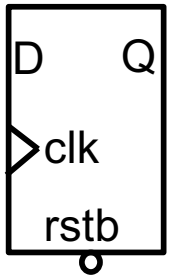
```
if (select == 1)
      y = a;
else
      y = b;
```

```
case (op)
  2'b00: y = a + b;
  2'b01: y = a – b;
  2'b10: y = a ^ b;
  default: y = 'hxxxx;
endcase
```

Always for Sequential Logic with If Statement

Flip Flop with Negative Edge Asynchronous Reset

```
module dff(D, clk, rstb, Q);
        input D, clk, rstb;
        output Q;
        reg Q;
        always @(negedge rstb or posedge clk)
                if (!rstb) begin
                        Q = 0;
                end
                else begin
                        Q = D;
                end
endmodule
```



Always for Sequential Logic

- Use “if…else” for conditional logic

```
module my_counter();
        reg [3:0] counter;
        wire clk, rstb, enable, up_en, down_en;
        always @ (posedge clk or negedge rstb)
                if (rstb == 1’b0) begin
                        counter = 4’b0000;
                end
                else if (enable == 1’b1 && up_en == 1’b1) begin
                        counter = counter + 1’b1;
                end
                else if (enable == 1’b1 && down_en == 1’b1) begin
                        counter = counter – 1’b1;
                end
                else  begin
                        counter = counter;
                end
endmodule
```

For Loops

- A increasing sequence of values on an output

```
reg [3:0] i, output;

for ( i = 0 ; i <= 15 ; i = i + 1 ) begin
  output = i;
  #10;
end
```

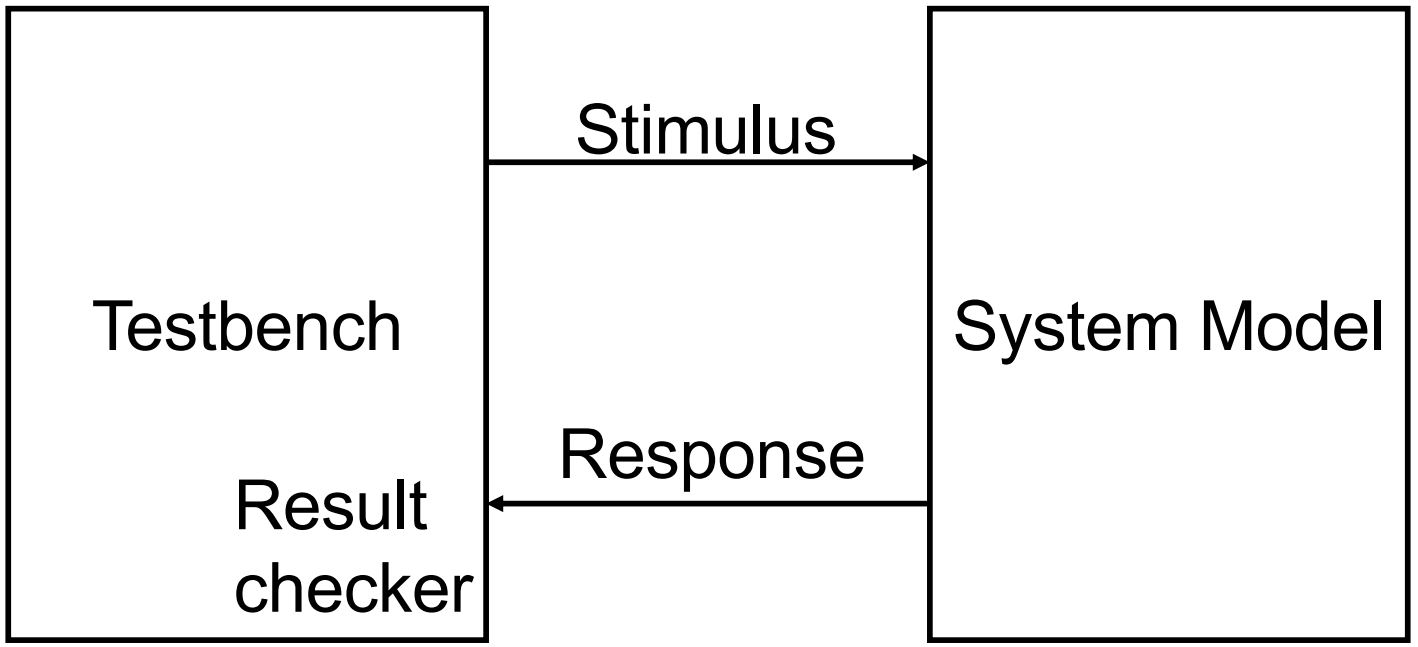
While Loops

- A increasing sequence of values on an output

```
reg [3:0] i, output;

i = 0;
while (i <= 15) begin
  output = i;
  #10 i = i + 1;
end
```

Simulating Verilog Code

- Testbench: a Verilog module to provide stimuli
- Instantiate circuits under test in testbench
- Pair of stimuli generator and circuit under test run simultaneously



Writing Testbenches

```
module test;
reg a, b, sel;

mux m(y, a, b, sel);

initial begin
  $monitor($time,, "a = %b b=%b sel=%b y=%b",
                   a, b, sel, y);
  a = 0; b= 0; sel = 0;
  #10 a = 1;
  #10 sel = 1;
  #10 b = 1;
end

end module
```

Inputs to device under test

Device under test

$monitor is a built-in event driven "printf" function

Stimulus generated by sequence of assignments and delays

More than modeling hardware

- $monitor — give it a list of variables. When one of them changes, it prints the information. Can only have one of these active at a time.
  e.g. …
  - $monitor ($time,,, "a=%b, b=%b, sum=%b, cOut=%b",a, b, sum, cOut);

extra commas to print spaces

%b is binary (also, %h, %d and others)

  - The above will print:
    2 a=0, b=0, sum=0, cOut=0<return>

newline automatically included

- $display() — sort of like printf()
  - $display ("Hello, world — %h", hexvalue)

display contents of data item called "hexvalue" using hex digits (0-9,A-F)

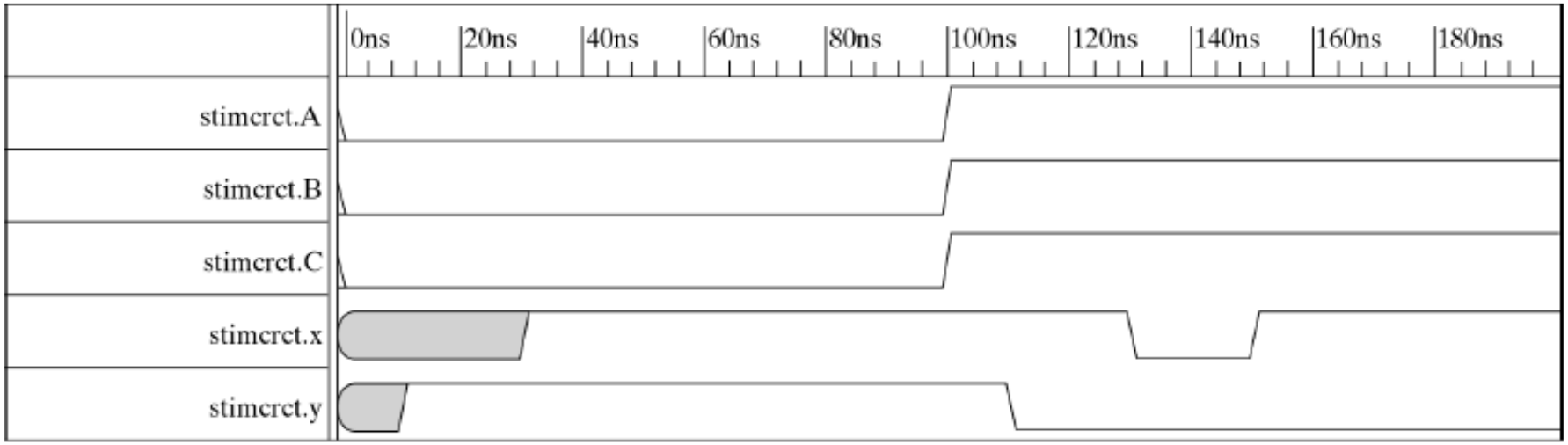
Testbench Stimulus

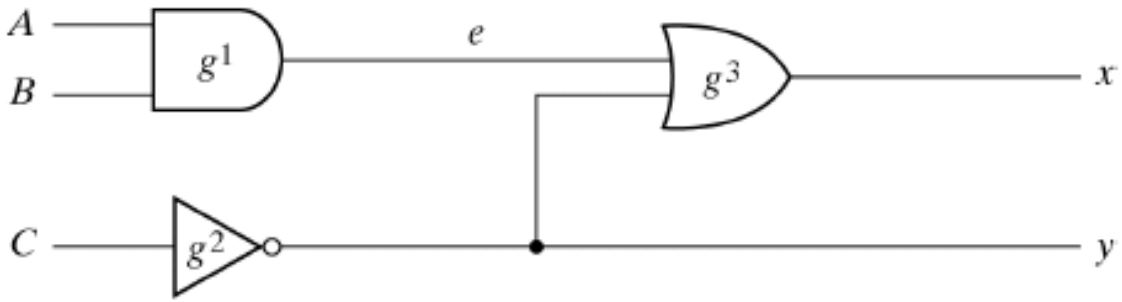
```
module circuit_under_test (A,B,C,x,y);
      input A,B,C;
      output x,y;
      wire e;
      and # (30) g1(e,A,B);
      or # (20) g3(x,e,y);
      not # (10) g2(y,C);
endmodule
```

Specify time delay of circuit (not synthesizable, only meaningful for the simulator)

```
`timescale 1ns / 100ps
module stim_crct;
    reg A,B,C;
    wire x,y;
    circuit_under_test c1(A,B,C,x,y);
    initial
          begin
          A = 1'b0; B = 1'b0; C = 1'b0;
          #100
          A = 1'b1; B = 1'b1; C = 1'b1;
          #100 $finish;
    end
endmodule
```

Testbench Stimulus





Discrete-event Simulation

- Basic idea: simulator is active only when it needs to react upon an event (when something changes)
- Main data structure used by simulators
  - Event Queue
  - Contains events labeled with the time stamp at which they are to be executed
- Basic simulation paradigm
  - Execute every event in the queue for the current simulated time
  - These events may change system state and may trigger to schedule new events in the future
  - When there are no events left at the current time instance, advance simulated time to the soonest time stamp of event in the queue

Simulation Behavior

- Non-preemptive, no priorities
- A process must explicitly request a context switch
- Events at a particular time stamp are unordered

Two Types of Events

- Evaluation events compute functions of inputs
  - Right hand side of assignments
- Update events change outputs
  - Left hand side of assignments, output ports of modules
- Taking into account # delays, non-blocking assignments

Update event writes new value of a and schedules any evaluation events that are sensitive to a change on **a**

a <= b + c

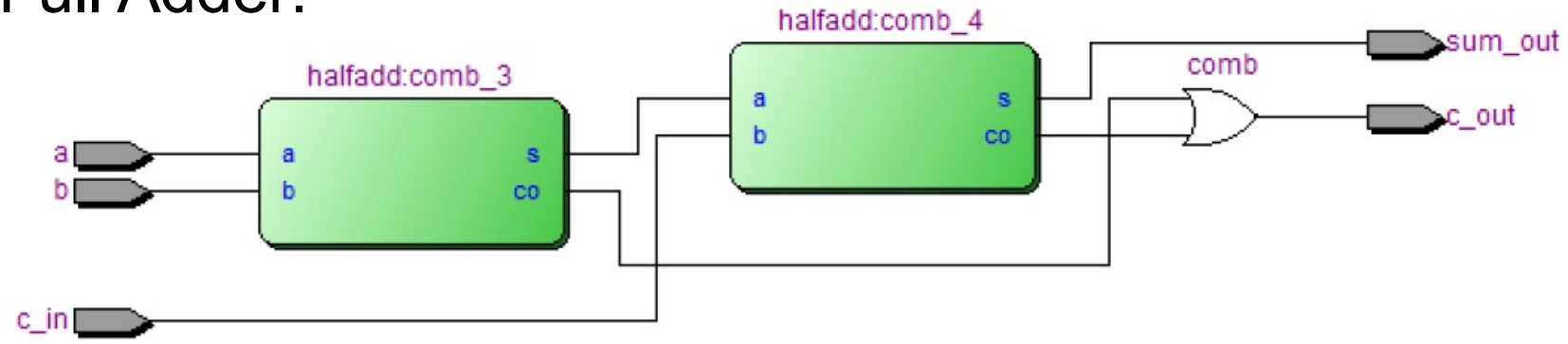
Evaluation event reads values of b and c, adds them, and schedules an update event

Simulation Behavior

- Concurrent processes (initial, always) run until they stop at one of the following
- #18
  - Schedule process to resume 18 time units from current time stamp
- Wait (BooleanStatement)
  - Resume when Statement becomes true
- @(a, b, y)
  - Resume when a, b, or y changes
- @(posedge clk)
  - Resume when clk changes from 0 to 1

Example: Adder

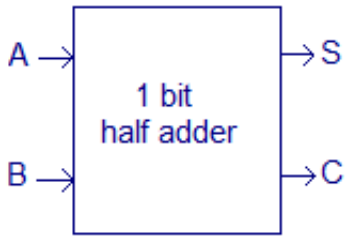
Full Adder:



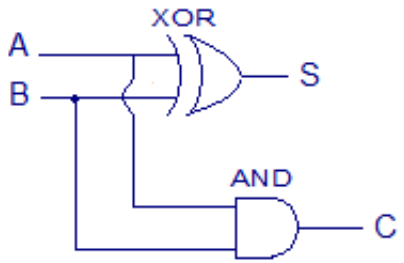
Half Adder:

Inputs		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Truth table



Schematic



Realization

Example: Adder

```
// Half Adder
//
module halfadd (a, b,
s, co);
        input a, b;
        output s, co;

        assign s = a ^ b;
        assign co = a & b;
endmodule
```

```
// Full Adder from Half Adders
//
module fulladd (a, b, c_in, sum_out, c_out);
        input a, b, c_in;
        output sum_out, c_out;
        wire s1, c1, c2;

        halfadd ha1(a, b, s1, c1);
        halfadd ha2(s1, c_in, sum_out, c2);

        assign c_out = c1 | c2;
endmodule
```

Putting it all together

- Computation encapsulated inside a **module**
  - Within a module computation can be described by three methods
    - Components instances - Structural
    - Direct/concurrent assignments - RTL
    - Procedures
      - **initial**: only used for simulation purposes in testbench codes
      - **always**: can be used to describe any behavior

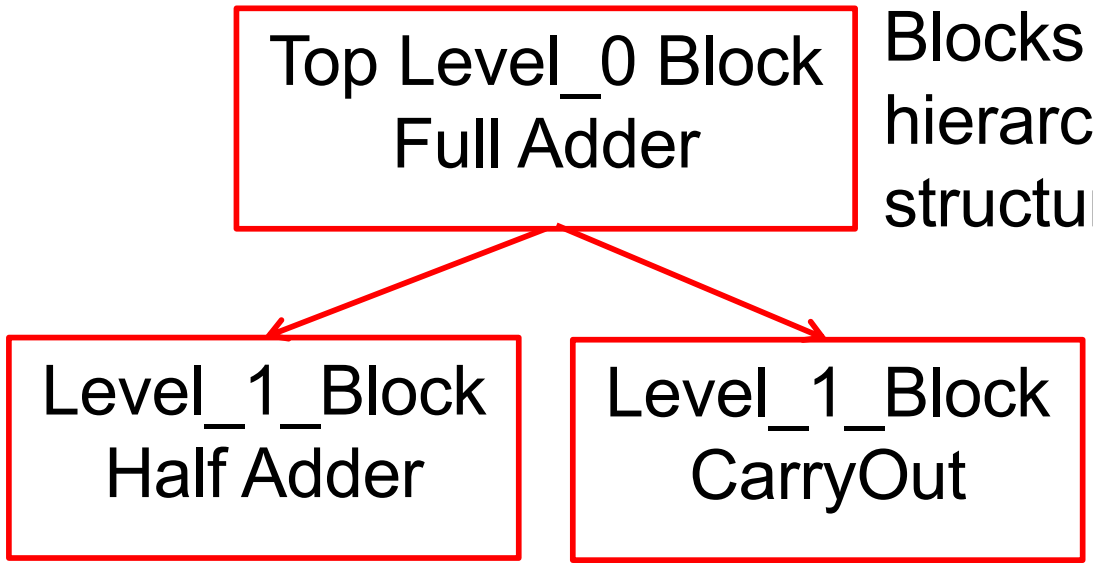
Putting it all together

- Computation encapsulated inside a **module**
  - Within a module computation can described by three methods
    1. Components instances - Structural
    2. Direct/concurrent assignments - RTL
    3. Procedures
       - **initial**: only used for simulation purposes in testbench codes
       - **always**: can be used to describe any behavior

1. Structural: Using component instances

```
// Full Adder from Half Adders and Additional Logic Pieces
//
module fulladd (a, b, c_in, sum_out, c_out);
        input a, b, c_in;
        output sum_out, c_out;
        wire s1, c1, c2;

        halfadd ha1(a, b, s1, c1);
        halfadd ha2(s1, c_in, sum_out, c2);
        carryout_block co1 (c1, c2, c_out);

endmodule
```

How do you create a project like that?

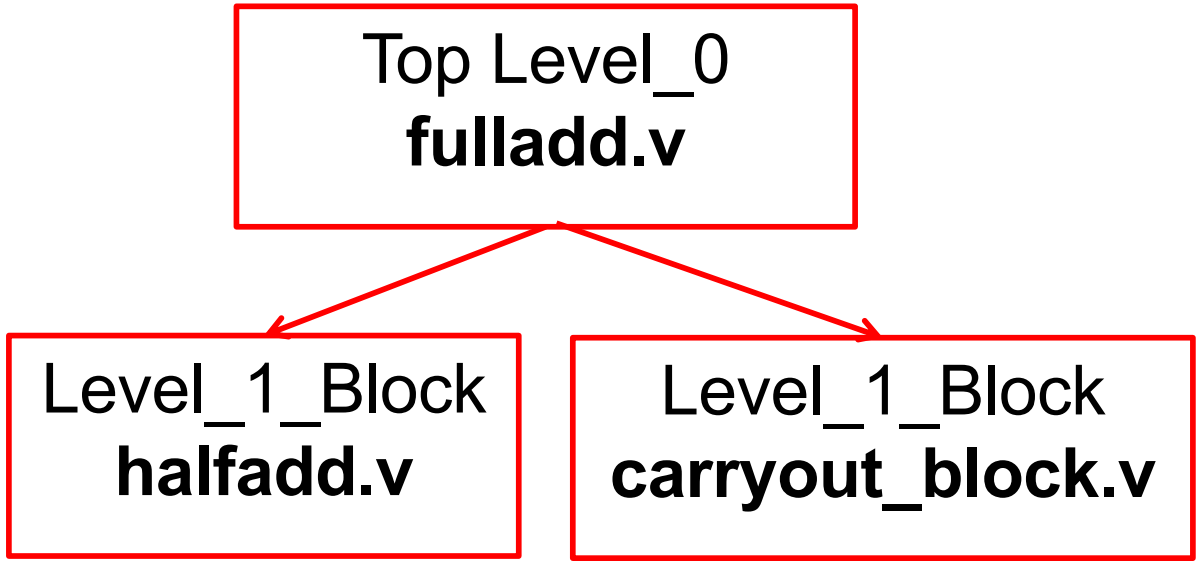


Blocks at the very top of hierarchy are described structurally

Need to describe the function of Half Adder:
All blocks at the bottom of the hierarchy must have a behavioral or RTL description

Need to describe the function of CarryOut:
All blocks at the bottom of the hierarchy must have a behavioral or RTL description

How do you create a project like that?



Can be composed of three verilog files:
carryout_block.v, halfadd.v, fulladd.v
In addition, each design file can have its dedicated testbench verilog file for its simulation

2. RTL Description Halfadd.v

```
// Half Adder
//
module halfadd (a, b,
s, co);
        input a, b;
        output s, co;

        assign s = a ^ b;
        assign co = a & b;
endmodule
```

RTL Description Carryout_block.v

```
// CarryOut
//
Module carryout_block (c1, c2, co);
        input c1, c2;
        output co;

        assign co = c1 & c2;
endmodule
```

Fulladd.v

```
// Full Adder from Half Adders
//
module fulladd (a, b, c_in, sum_out, c_out);
        input a, b, c_in;
        output sum_out, c_out;
        wire s1, c1, c2;

        halfadd ha1(a, b, s1, c1);
        halfadd ha2(s1, c_in, sum_out, c2);

        carryout_block cout1 (c1, c2, c_out);

endmodule
```

Putting it all together

- Computation encapsulated inside a **module**
  - Within a module computation can described by three methods
    1 Components instances - Structural
    2 Direct/concurrent assignments - RTL
    3 Procedures
      - **initial**: only used for simulation purposes in testbench codes
      - **always**: can be used to describe any behavior

Halfadd.v

```
// Half Adder
//
module halfadd (a, b, s, co);
        input a, b;
        output s, co;

        assign s = a ^ b;
        assign co = a & b;
endmodule
```

This style was used to describe the half adder in RTL style:

Signature statement is **assign**

Boolean or arithmetic operators directly applied on signal values on the right hand side

Putting it all together

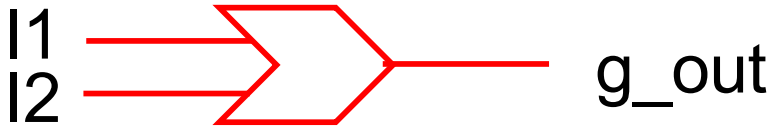
- Computation encapsulated inside a **module**
  - Within a module computation can described by three methods
    1 Components instances - Structural
    2 Direct/concurrent assignments - RTL
    3 Procedures
      - **initial**: only used for simulation purposes in testbench codes
      - **always @**: can be used to describe any behavior

Always procedure

- You can package statements of both RTL and Behavioral Style inside the always@ procedure
  - RTL, as a sequence of concurrent assignments
    - corresponding to combinational logic
    - Boolean or arithmetic operators on the right hand side of assignments
  - Behavioral, sequential statements
    - Corresponding to combinational or sequential logic
    - if-else statement

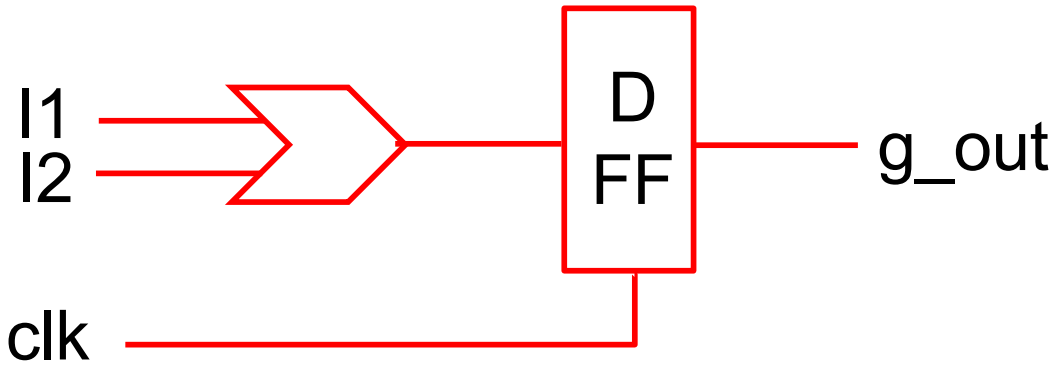
Always procedure with RTL statements

```
module pure_gate (I1, I2, g_out);
  input I1, I2;
output g_out;

always @(I1, I2)  begin
  g_out <= I1 OR I2;
  end;
end module;
```



Always procedure with behavioral statements

```
Module seq_logic (I1, I2, clk, g_out);
  input I1, I2, clk;
  output g_out;

always @(posedge clk)  begin
   g_out <= I1 OR I2;
end;
end module;
```



Simulation Environment

- Build a Verilog testbench
- Use simulator to simulate waveforms

Simulation Environment

- Simulation RTL vs Gate Netlist
  - Ideally both should give the same results
  - RTL
    - Only the high level behavior of your design
    - No delay from transition
    - No gate level implementation and signals
  - Gate netlist (synthesized from RTL)
    - Show delay from gates
    - May be different from RTL if synthesis has issue
    - Should be run to verify correctness of RTL