Digital System Design

Verilog® HDL
Useful Modeling Techniques

Maziar Goudarzi

Today Program

- Procedural Continuous Assignment
- Overriding Parameters
- Conditional Compilation and Execution
- Useful System Tasks

Procedural Continuous Assignment

Useful Modeling Techniques

Procedural Continuous Assignment

- Overrides, for a certain time, the effect of regular assignments to a variable.
- Two types
  - `assign/deassign`
    - Works only on register data types
  - `force/release`
    - Works on both register and net data types
- Note:
  - Not synthesizable. Use only for modeling and simulation

Procedural Continuous Assignment (cont'd)

- `assign/deassign`
  - Keywords
    - `assign`: overrides regular procedural assignments
      - LHS: reg or concatenation of regs. No nets. No arrays. No bit-select or part-select
    - `deassign`: re-enables regular procedural assignments
  - After `deassign`:
    - Last value remains on the register until a new procedural assignment changes it.

```
// Negative edge-triggered D-flipflop with asynchronous reset
module edge_dff(q, qbar, d, clk, reset);

// Inputs and outputs
output q,qbar;
input d, clk, reset;
reg q, qbar; //declare q and qbar are registers

always @(negedge clk) //assign value of q & qbar at active edge of clock.
begin
        q = d;
        qbar = ~d;
end

always @(reset) //Override the regular assignments to q and qbar
              //whenever reset goes high. Use of procedural continuous
                 //assignments.
        if(reset)
      begin  //if reset is high,override regular assignments to q with
            //the new values, using procedural continuous assignment.
               assign q = 1'b0;
               assign qbar = 1'b1;
        end
        else
        begin   //If reset goes low, remove the overriding values by
                //deassigning the registers. After this the regular
             //assignments q = d and qbar = ~d will be able to change
             //the registers on the next negative edge of clock.
               deassign q;
               deassign qbar;
        end

endmodule
```

Procedural Continuous Assignment (cont'd)

- `force/release`
  - Keywords:
    - `force`: overrides all procedural/continuous/ procedural continuous assignments
    - `release`: re-enables other assignments
    - Hence, assignments in priority order:
      1. `force`
      2. `assign` (procedural continuous)
      3. Procedural/continuous assignments

force/release on reg variables

```
module stimulus;
…
…
//instantiate the d-flipflop   ,
edge_dff dff(Q, Qbar, D, CLK, RESET);
…
…
initial
begin
    //these statements force value of 1 on dff.q between time 50 and
     //100, regardless of the actual output of the edge_dff.
    #50 force dff.q = 1'b1; //force value of q to 1 at time 50.
    #50 release dff.q;  //release the value of q at time 100.
end
…
…
endmodule
```

`force/release` on nets

```
module top;
…
…
assign out = a & b & c; //continuous assignment on net out
…
initial
    #50 force out = a | b & c;
    #50 release out;
end
…
…
endmodule
```

- Net value immediately returns to its normal `assign`ed value when released

Overriding Parameters

Useful Modeling Techniques

Overriding Parameters

- Two methods
  - `defparam` statement
  - Module instance parameter value assignment
- `defparam` statement
  - Keyword: `defparam`
  - Syntax:

```
defparam <parameter_hierarchical_name>=<value>;
```

```
//Define a module hello_world
module hello_world;
parameter id_num = 0; //define a module identification number = 0

initial //display the module identification number
        $display("Displaying hello_world id number = %d", id_num);
endmodule


//define top-level module
module top;
//change parameter values in the instantiated modules
//Use defparam statement
defparam w1.id_num = 1, w2.id_num = 2;

//instantiate two hello_world modules
hello_world w1();
hello_world w2();

endmodule
```

```
Displaying hello_world id number = 1
Displaying hello_world id number = 2
```

Overriding Parameters (cont'd)

- Module instance parameter values
  - Parameters are overridden when the module is instantiated
  - Syntax:

```
<module_name> #(<param_vals>) <instance_name>;
```

```
//define top-level module
module top;

//instantiate two hello_world modules; pass new parameter values
hello_world #(1) w1; //pass value 1 to module w1
hello_world #(2) w2; //pass value 2 to module w2

endmodule
```

Example with multiple parameters

```
//define module with delays
module bus_master;
parameter delay1 = 2;
parameter delay2 = 3;
parameter delay3 = 7;
…
<module internals>
…
endmodule

//top-level module; instantiates two bus_master modules
module top;

//Instantiate the modules with new delay values
bus_master #(4, 5, 6) b1(); //b1: delay1 = 4, delay2 = 5, delay3 = 6
bus_master #(9,4) b2(); //b2: delay1 = 9, delay2 = 4, delay3 = 7(default)

endmodule
```

Conditional Compilation and Execution

Useful Modeling Techniques

Conditional Compilation

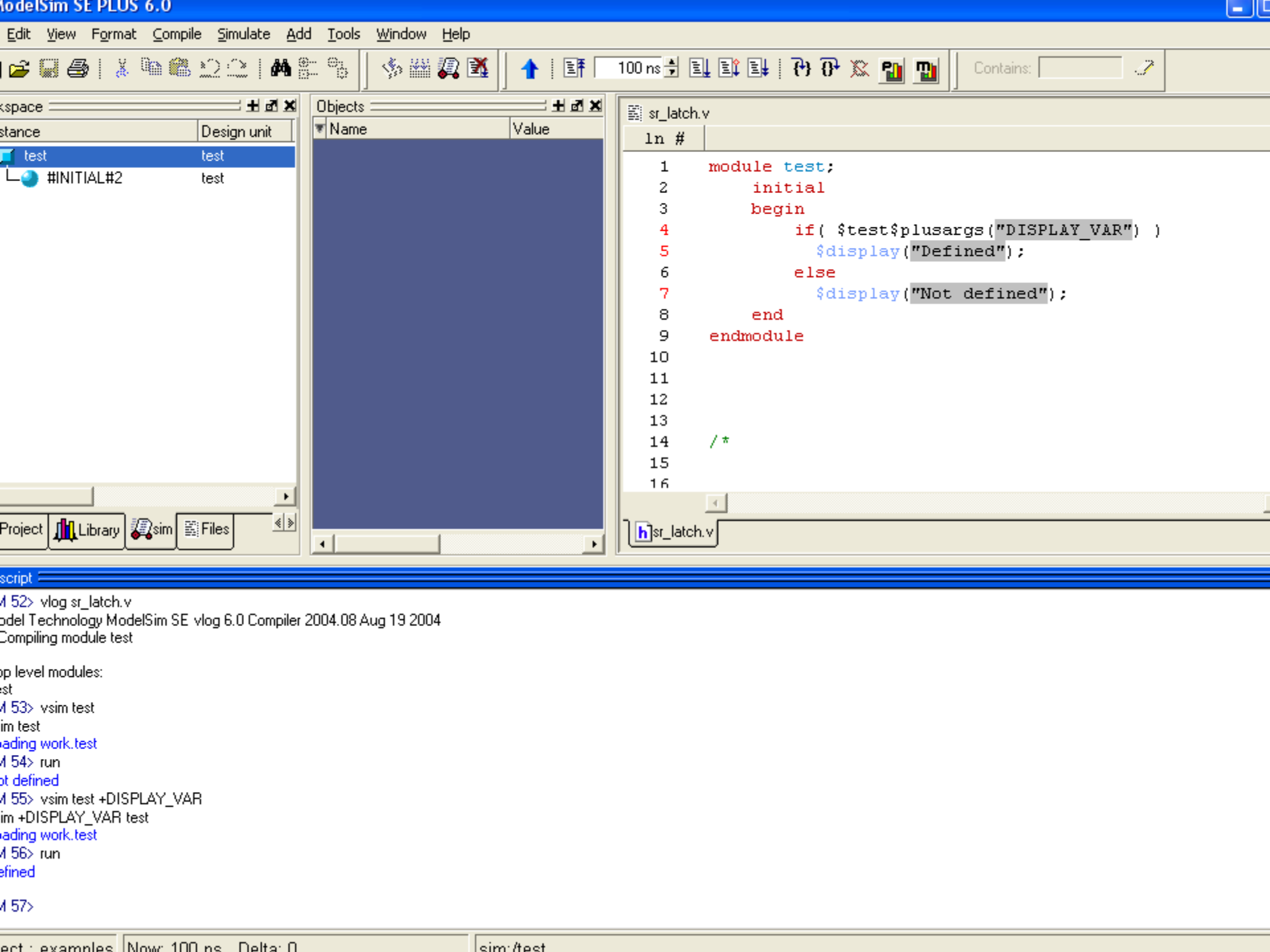
- Usage:
  - To compile some part of code under certain conditions
- Keywords:
  - `` `ifdef, `else, `endif ``
  - `` `define `` to define the flag

```
//Conditional Compilation

//Example 1
`ifdef TEST //compile module test only if text macro TEST is defined
module test;
…
…
endmodule
`else //compile the module stimulus as default
module stimulus;
…
…
endmodule
`endif //completion of `ifdef statement

//Example 2
module top;

bus_master b1(); //instantiate module unconditionally
`ifdef ADD_B2
   bus_master b2(); //b2 is instantiated conditionally if text macro
                    //ADD_B2 is defined
`endif

endmodule
```

Conditional Execution

- Usage:
  - To execute some part of code when a flag is set at runtime
  - Used only in behavioral modeling
- Keywords:
  - `$test$plusargs`
- Syntax:
  - `$test$plusargs( <argument_to_check> )`

```
//Conditional execution
module test;
…
…
initial
begin
    if($test$plusargs("DISPLAY_VAR"))
        $display("Display = %b ", {a,b,c} ); //display only if flag is set
    else
        $display("No Display"); //otherwise no display
end
endmodule
```


ModelSim SE PLUS 6.0
Edit View Format Compile Simulate Add Tools Window Help
100 ns
Contains:
kspace
stance Design unit
test test
#INITIAL#2 test
Project Library sim Files
Objects
Name Value
sr_latch.v
ln #
1 module test;
2 initial
3 begin
4 if($test$plusargs("DISPLAY_VAR"))
5 $display("Defined");
6 else
7 $display("Not defined");
8 end
9 endmodule
10
11
12
13
14 /*
15
16
sr_latch.v
script
M 52> vlog sr_latch.v
odel Technology ModelSim SE vlog 6.0 Compiler 2004.08 Aug 19 2004
Compiling module test
op level modules:
est
M 53> vsim test
im test
ading work.test
M 54> run
ot defined
M 55> vsim test +DISPLAY_VAR
im +DISPLAY_VAR test
ading work.test
M 56> run
efined
M 57>
ect : examples
Now: 100 ns Delta: 0
sim:/test

Useful System Tasks

Useful Modeling Techniques

Useful System Tasks
File Output

- Opening a file
  - Syntax:

    `<file_handle> = $fopen( "<file_name>" );`
  - `<file_handle>` is a 32 bit value, called *multi-channel descriptor*
  - Only 1 bit is set in each descriptor
  - Standard output has a descriptor of 1 (Channel 0)

```
//Multichannel descriptor
integer handle1, handle2, handle3; //integers are 32-bit values

//standard output is open; descriptor = 32'h0000_0001 (bit 0 set)
initial
begin
   handle1 = $fopen("file1.out"); //handle1 = 32'h0000_0002 (bit 1 set)
   handle2 = $fopen("file2.out"); //handle2 = 32'h0000_0004 (bit 2 set)
   handle3 = $fopen("file3.out"); //handle3 = 32'h0000_0008 (bit 3 set)
end
```

Useful System Tasks
File Output (cont'd)

- Writing to files
  - `$fdisplay, $fmonitor, $fstrobe`
  - `$strobe, $fstrobe`
    - The same as `$display, $fdisplay`, but executed after all other statements schedule in the same simulation time
  - Syntax:

    `$fdisplay(<handle>, p1, p2,…, pn);`
- Closing files

  `$fclose(<handle>);`

Example: Simultaneously writing to multiple files

```
//All handles defined in Example 9-7
//Writing to files
integer desc1, desc2, desc3; //three file descriptors
initial
begin
    desc1 = handle1 | 1; //bitwise or; desc1 = 32'h0000_0003
    $fdisplay(desc1, "Display 1");//write to files file1.out & stdout

    desc2 = handle2 | handle1; //desc2 = 32'h0000_0006
    $fdisplay(desc2, "Display 2");//write to files file1.out & file2.out

    desc3 = handle3 ; //desc3 = 32'h0000_0008
    $fdisplay(desc3, "Display 3");//write to file file3.out only
end
```

Strobe

```
//Strobing
always @(posedge clock)
begin
a = b;
c = d;
end
always @(posedge clock)
$strobe("Displaying a = %b, c = %b", a, c); // display values at posedge
```

Useful System Tasks
Random Number Generation

- Syntax:

  `$random;`

  `$random(<seed>);`

- Returns a 32 bit random value

```
//Generate random numbers and apply them to a simple ROM
module test;
integer r_seed;
reg [31:0] addr;//input to ROM
wire [31:0] data;//output from ROM
…
…
ROM rom1(data, addr);

initial
   r_seed = 2; //arbitrarily define the seed as 2.

always @(posedge clock)
   addr = $random(r_seed); //generates random numbers
…
<check output of ROM against expected results>
…
…
endmodule
```

```
reg [23:0] rand1, rand2;
rand1 = $random % 60; //Generates a random
//number between -59 and 59
rand2 = {$random} % 60; //Addition of
//concatenation operator to $random generates a
//positive value between0 and 59.
```

Digital System Design

Verilog® HDL
Behavioral Modeling (2)

Today program

- Behavioral Modeling
  - Timing controls
  - Other features

Behavioral Modeling

Timing Controls in Behavioral Modeling

Introduction

- No timing controls $\Rightarrow$ No advance in simulation time
- Three methods of timing control
  - delay-based
  - event-based
  - level-sensitive

Delay-based Timing Controls

- ***Delay*** ≡ Duration between encountering and executing a statement
- Delay symbol: #
- Delay specification syntax:

<delay>

::= #<*NUMBER*>

||= #<*identifier*>

||= #<*mintypmax_exp*> <,<*mintypmax_exp*>>*)

Delay-based Timing Controls (cont'd)

- Types of delay-based timing controls
  1. Regular delay control
  2. Intra-assignment delay control
  3. Zero-delay control

Delay-based Timing Controls (cont'd)

- Regular Delay Control
  - Symbol: non-zero delay before a procedural assignment
  - Used in most of our previous examples

```
//define parameters
parameter latency = 20;
parameter delta = 2;
//define register variables
reg x, y, z, p, q;

initial
begin
        x = 0; // no delay control
       #10 y = 1; // delay control with a number. Delay execution of
                                  // y = 1 by 10 units

      #latency z = 0; //Delay control with identifier.Delay of 20 units
        #(latency + delta) p = 1; // Delay control with expression

       #y x = x + 1; // Delay control with identifier. Take value of y.

        #(4:5:6) q = 0; // Minimum, typical and maximum delay values.
                    //Discussed in gate-level modeling chapter.
end
```

Delay-based Timing Controls (cont'd)

- Intra-assignment Delay Control
  - Symbol: non-zero delay to the right of the assignment operator
  - Operation sequence:
    1. Compute the right-hand-side expression at the current time.
    2. Defer the assignment of the above computed value to the LHS by the specified delay.

Delay-based Timing Controls (cont'd)

- Intra-assignment delay examples

```
//define register variables
reg x, y, z;

//intra assignment delays
initial
begin
        x = 0; z = 0;
       y = #5 x + z; //Take value of x and z at the time=0, evaluate
                  //x + z and then wait 5 time units to assign value
                 //to y.

end

//Equivalent method with temporary variables and regular delay control
initial
begin
        x = 0; z = 0;
        temp_xz = x + z;
       #5 y = temp_xz; //Take value of x + z at the current time and
              //store it in a temporary variable. Even though x and z
                 //might change between 0 and 5,
                 //the value assigned to y at time 5 is unaffected.
end
```

Delay-based Timing Controls (cont'd)

- Zero-Delay Control
  - Symbol: #0
  - Different *initial/always* blocks in the same simulation time
    - Execution order non-deterministic
  - Zero-delay ensures execution after all other statements
    - Eliminates race conditions
  - Multiple zero-delay statements
    - Non-deterministic execution order

Delay-based Timing Controls (cont'd)

- Zero-delay control examples

```
initial
begin
    x = 0;
    y = 0;
end

initial
begin
    #0 x = 1; //zero delay control
    #0 y = 1;
end
```

Event-based Timing Control

- Event
  - Change in the value of a register or net
  - Used to trigger execution of a statement or block (reactive behavior/reactivity)
- Types of Event-based timing control
  1. Regular event control
  2. Named event control
  3. Event OR control
  4. Level-sensitive timing control (next section)

Event-based Timing Control (cont'd)

- Regular event control
  - Symbol: `@(<event>)`
  - Events to specify:
    - `posedge sig`
      - Change of `sig` from any value to 1 or from 0 to any value
    - `negedge sig`
      - Change of `sig` from any value to 0 or from 1 to any value
    - `sig`
      - Any chage in `sig` value

Event-based Timing Control (cont'd)

- Regular event control examples

```
@(clock) q = d; //q = d is executed whenever signal clock changes value
@(posedge clock) q = d; //q = d is executed whenever signal clock does
                        //a positive transition ( 0 to 1,x or z,
                        // x to 1, z to 1 )
@(negedge clock) q = d; //q = d is executed whenever signal clock does
                        //a negative transition ( 1 to 0,x or z,
                        //x to 0, z to 0)
q = @(posedge clock) d; //d is evaluated immediately and assigned
                        //to q at the positive edge of clock
```

Event-based Timing Control (cont'd)

- Named event control
  - You can *declare* (name) an event, and then *trigger* and *recognize* it.
  - Verilog keyword for declaration: *event*

```
event calc_finished;
```

  - Verilog symbol for triggering: `->`

```
->calc_finished
```

  - Verilog symbol for recognizing: `@()`

```
@(calc_finished)
```

Event-based Timing Control (cont'd)

- Named event control examples

```
//This is an example of a data buffer storing data after the
//last packet of data has arrived.

event received_data; //Define an event called received_data

always @(posedge clock) //check at each positive clock edge
begin
        if(last_data_packet) //If this is the last data packet
                ->received_data; //trigger the event received_data
end

always @(received_data) //Await triggering of event received_data
                        //When event is triggered, store all four
```

```
                        //packets of received data in data buffer
                    //use concatenation operator { }
    data_buf = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[3]};
```

Event-based Timing Control (cont'd)

- Event OR control
  - Used when need to trigger a block upon occurrence of any of a set of events.
  - The list of the events: sensitivity list
  - Verilog keyword: `or`

```
//A level-sensitive latch with asynchronous reset
always @( reset or clock or d)
                                //Wait for reset or clock or d to change
begin
        if (reset)              //if reset signal is high, set q to 0.
                q = 1'b0;
        else    if(clock)       //if clock is high, latch input
                q = d;
end
```

Level-sensitive Timing Control

- Level-sensitive vs. event-based
  - event-based: wait for triggering of an event (change in signal value)
  - level-sensitive: wait for a certain condition (on values/levels of signals)
- Verilog keyword: `wait()`

```
always
  wait(count_enable) #20 count=count+1;
```

Behavioral Modeling

Other Features

Contents

- Sequential and Parallel Blocks
- Special Features of Blocks

Sequential and Parallel Blocks

- Blocks: used to group multiple statements
- Sequential blocks
  - Keywords: `begin end`
  - Statements are processed in order.
  - A statement is executed only after its preceding one completes.
    - Exception: non-blocking assignments with intra-assignment delays
  - A delay or event is relative to the simulation time when the previous statement completed execution

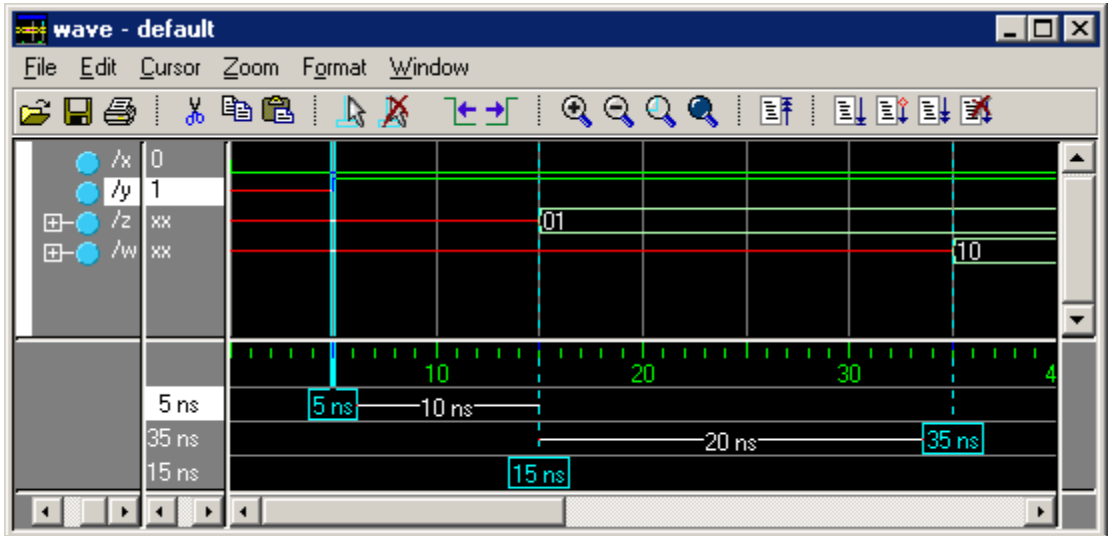
Sequential and Parallel Blocks (cont'd)

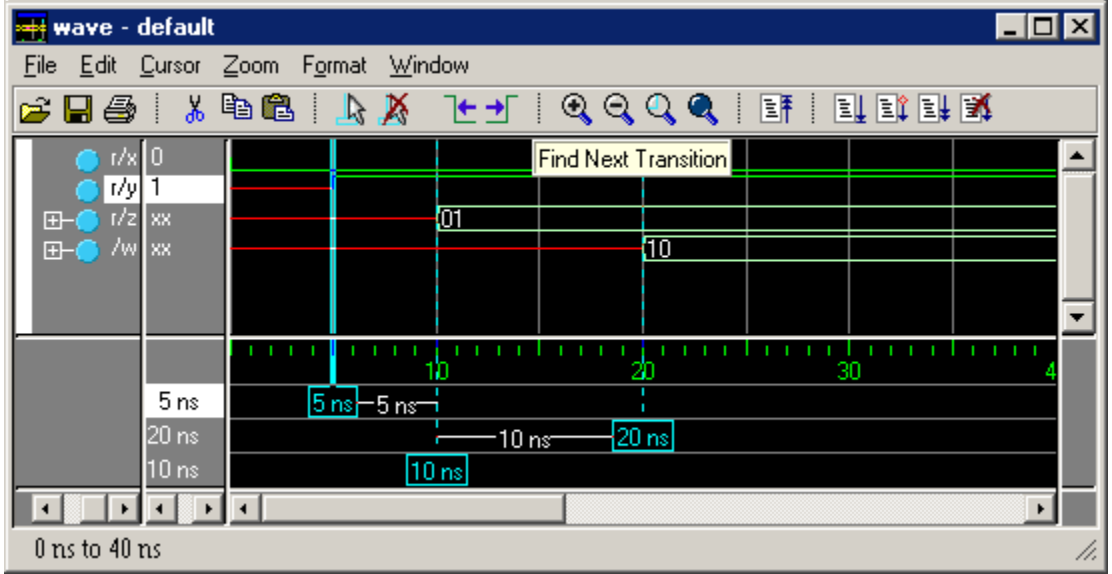
- Parallel Blocks
  - Keywords: `fork, join`
  - Statements in the blocks are executed concurrently
  - Timing controls specify the order of execution of the statements
  - All delays are relative to the time the block was entered
    - The written order of statements is not important

Sequential and Parallel Blocks (cont'd)

```
initial
begin
    x=1'b0;
    #5   y=1'b1;
    #10  z={x,y};
    #20  w={y,x};
end
```

```
initial
fork
    x=1'b0;
    #5   y=1'b1;
    #10  z={x,y};
    #20  w={y,x};
join
```





Sequential and Parallel Blocks (cont'd)

- Parallel execution $\Rightarrow$ Race conditions may arise

```
initial
begin
        x=1'b0;
        y=1'b1;
        z={x,y};
        w={y,x};
end
```

- `z, w` can take either `2'b01, 2'b10` or `2'bxx, 2'bxx` depending on simulator

Special Features of Blocks

- Contents
  - Nested blocks
  - Named blocks
  - Disabling named blocks

Special Features of Blocks (cont'd)

- Nested blocks
  - Sequential and parallel blocks can be mixed

```
initial
begin
  x=1'b0;
  fork
       #5 y=1'b1;
       #10 z={x,y};
  join
  #20  w={y,x};
end
```

Special Features of Blocks (cont'd)

- Named blocks
  - Syntax:

```
begin: <the_name>          fork: <the_name>
  …                              …
end                        join
```

  - Advantages:
    - Can have local variables
    - Are part of the design hierarchy.
    - Their local variables can be accessed using hierarchical names
    - Can be disabled

Special Features of Blocks (cont'd)

```
module top;
   initial
   begin : block1
     integer i;   //hiera. name: top.block1.i
   …
   end

   initial
   fork : block2
   reg i;   //hierarchical name: top.block2.i
   …
   join
endmodule
```

Special Features of Blocks (cont'd)

- Disabling named blocks
  - Keyword: `disable`
  - Action:
    - Similar to `break` in C/C++, but can disable any named block not just the inner-most block.

Special Features of Blocks (cont'd)

```
module find_true_bit;

reg [15:0] flag;
integer i;

initial
begin
   flag = 16'b
    0010_0000_0000_0000;
   i = 0;
   begin: block1
   while(i < 16)
   begin
      if (flag[i])
      begin
       $display("Encountered a
            TRUE bit at element
            number %d", i);
       disable block1;
      end // if
      i = i + 1;
      end // while
   end // block1
   end //initial
   endmodule
```

Behavioral Modeling

Examples

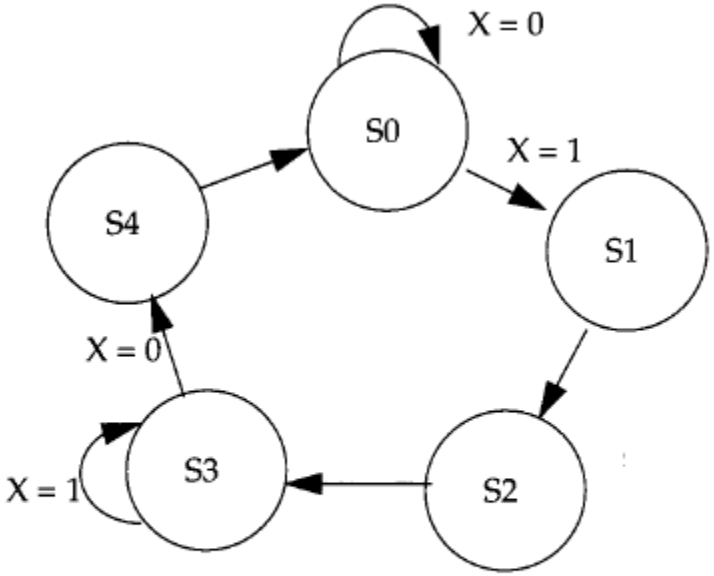
4-to-1 Multiplexer

```
// 4-to-1 multiplexer. Port list is taken exactly from
// the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

  // Port declarations from the I/O diagram
  output out;
  input i0, i1, i2, i3;
  input s1, s0;
  reg out; //output declared as register

  //recompute the signal out if any input signal changes.
  //All input signals that cause a recomputation of out to
  //occur must go into the always @(...)
  always @(s1 or s0 or i0 or i1 or i2 or i3)
  begin
    case ({s1, s0})
      2'b00: out = i0;
      2'b01: out = i1;
      2'b10: out = i2;
      2'b11: out = i3;
      default: out = 1'bx;
    endcase
  end
endmodule
```

4-bit Counter

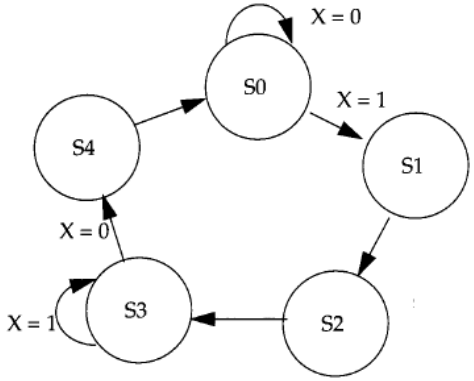
```
//Binary  counter
module counter(Q , clock, clear);

  // I/O ports
  output [3:0] Q;
  input clock, clear;
  //output defined as register
  reg [3:0] Q;

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

endmodule
```

Traffic Signal Controller



State	Signals
S0	*Hwy = G Cntry = R*
S1	*Hwy = Y Cntry = R*
S2	*Hwy = R Cntry = R*
S3	*Hwy = R Cntry = G*
S4	*Hwy = R Cntry = Y*

```
`define TRUE        1'b1
`define FALSE       1'b0
`define RED         2'd0
`define YELLOW 2'd1
`define GREEN       2'd2

//State definition                      HWY       CNTRY
`define S0                  3'd0  //GREEN       RED
`define S1                  3'd1  //YELLOW      RED
`define S2                  3'd2  //RED         RED
`define S3                  3'd3  //RED         GREEN
`define S4                  3'd4  //RED         YELLOW

//Delays
`define Y2RDELAY    3         //Yellow to red delay
`define R2GDELAY    2         //Red to Green Delay
```



State	Signals
S0	*Hwy = G Cntry = R*
S1	*Hwy = Y Cntry = R*
S2	*Hwy = R Cntry = R*
S3	*Hwy = R Cntry = G*
S4	*Hwy = R Cntry = Y*

```
module sig_control (hwy, cntry, X, clock, clear);

//I/O ports
output [1:0] hwy, cntry; //2 bit output for 3 states of signal GREEN, YELLOW, RED;
reg [1:0] hwy, cntry;    //declare output signals are registers

input X;  //if TRUE, indicates that there is car on  the country road, otherwise FALSE

input clock, clear;

//Internal state variables
reg [2:0] state;
reg [2:0] next_state;
```

```
//Signal controller starts in S0 state
initial begin
    state = `S0;
    next_state = `S0;
    hwy = `GREEN;
    cntry = `RED;
end

always @(posedge clock) //state changes only at positive edge of clock
    state = next_state;

always @(state) //Compute values of main signal and country signal
begin
    case(state)
          `S0:      begin
                            hwy = `GREEN;
                            cntry = `RED;
                    end
          `S1:      begin
                            hwy = `YELLOW;
                            cntry = `RED;
                    end
          `S2:      begin
                            hwy = `RED;
                            cntry = `RED;
                    end
          `S3:      begin
                            hwy = `RED;
                            cntry = `GREEN;
                    end
          `S4:      begin
                            hwy = `RED;
                            cntry = `YELLOW;
                    end
    endcase
end
```

```
//State machine using case statements
always @(state or clear or  X)
begin
    if(clear)
          next_state = `S0;
    else
          case (state)
                    `S0: if( X)
                              next_state = `S1;
                         else
                              next_state = `S0;
                    `S1: begin //delay some positive edges of clock
                              repeat(`Y2RDELAY) @(posedge clock) ;
                              next_state = `S2;
                         end
                    `S2: begin //delay some positive edges of clock
                              repeat(`R2GDELAY) @(posedge clock)
                              next_state = `S3;
                         end
                    `S3: if( X)
                              next_state = `S3;
                         else
                              next_state = `S4;
                    `S4: begin //delay some positive edges of clock
                              repeat(`Y2RDELAY) @(posedge clock) ;
                              next_state = `S0;
                         end
                   default: next_state = `S0;
          endcase
end

endmodule
```

Digital System Design

Verilog® HDL
Behavioral Modeling (1)

Today program

- Behavioral Modeling
  - Concepts
  - Constructs

Introduction

- The move toward higher abstractions
  - Gate-level modeling
    - Netlist of gates
  - Dataflow modeling
    - Boolean function assigned to a net
  - Now, behavioral modeling
    - A sequential algorithm (quite similar to software) that determines the value(s) of signal(s)

Structured Procedures

- Two basic structured procedure statements

  `always`

  `initial`

  - All behavioral statements can appear only inside these blocks
  - Each always or initial block has a separate activity flow (concurrency)
  - Start from simulation time 0
  - Cannot be nested

Structured Procedures: `initial` statement

- Starts at time 0
- Executes only once during a simulation
- Multiple `initial` blocks, execute in parallel
  - All start at time 0
  - Each finishes independently
- Syntax:

```
initial
begin
  // behavioral statements
end
```

Structured Procedures: `initial` statement (cont'd)

- Example:

```
module stimulus;
  reg x, y, a, b, m;

  initial
    m= 1'b0;

  initial
  begin
    #5 a=1'b1;
    #25 b=1'b0;
  end

 initial
  begin
    #10 x=1'b0;
    #25 y=1'b1;
  end

  initial
     #50 $finish;
endmodule
```

time	statement executed
0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
30	b = 1'b0;
35	y = 1'b1;
50	$finish;

Structured Procedures: `always` statement

- Start at time 0
- Execute the statements in a looping fashion
- Example

```
module clock_gen;
  reg clock;

  // Initialize clock at time zero
  initial
    clock = 1'b0;

  // Toggle clock every half-cycle (time period =20)
  always
    #10 clock = ~clock;

  initial
    #1000 $finish;
endmodule
```

Can we move this to the `always` block?

What happens if such a `$finish` is not included?

Procedural Assignments

- Assignments inside `initial` and `always`
- Are used to update values of `reg`, `integer`, `real`, or `time` variables
  - The value remains unchanged until another procedural assignment updates it
  - In contrast to continuous assignment (Dataflow Modeling, previous chapter)

Procedural Assignments (cont'd)

- Syntax
  - *<lvalue> = <expression>*
  - *<lvalue>* can be
    - `reg`, `integer`, `real`, `time`
    - A bit-select of the above (e.g., `addr[0]`)
    - A part-select of the above (e.g., `addr[31:16]`)
    - A concatenation of any of the above
  - *<expression>* is the same as introduced in dataflow modeling
  - What happens if the widths do not match?
    - LHS wider than RHS => RHS is zero-extended
    - RHS wider than LHS => RHS is truncated (Least significant part is kept)

Behavioral Modeling Statements: Conditional Statements

- Just the same as `if-else` in C
- Syntax:

```
if (<expression>) true_statement;

if (<expression>) true_statement;
  else  false_statement;

if (<expression>) true_statement1;
  else if (<expression>) true_statement2;
  else if (<expression>) true_statement3;
  else default_statement;
```

- *True* is `1` or non-zero
- *False* is `0` or ambiguous (`x` or `z`)
- More than one statement: `begin end`

Conditional Statements (cont'd)

- Examples:

```
if (!lock) buffer = data;

if (enable) out = in;

if (number_queued < MAX_Q_DEPTH)
begin
  data_queue = data;
  number_queued = number_queued +1;
end
else $display("Queue full! Try again.");

if (alu_control==0)
  y = x+z;
else if (alu_control==1)
  y = x-z;
else if (alu_control==2)
  y = x*z;
else
  $display("Invalid ALU control signal.");
```

Behavioral Modeling Statements: Multiway Branching

- Similar to `switch-case` statement in C
- Syntax:

```
case (<expression>)
   alternative1: statement1;
   alternative2: statement2;
      ...
   default: default_statement; // optional
endcase
```

- Notes:
  - <expression> is compared to the alternatives in the order specified.
  - Default statement is optional

Multiway Branching (cont'd)

- Examples:

```
reg [1:0] alu_control;
...
case (alu_control)
  2'd0: y = x + z;
  2'd1: y = x - z;
  2'd2: y = x * z;
  default: $display("Invalid ALU control signal.");
```

- Now, you write a 4-to-1 multiplexer.

Multiway Branching (cont'd)

- Example 2:

```
module mux4_to_1(out, i0, i1, i2, i3, s1, s0);
  output out;
  input i0, i1, i2, i3, s1, s0;
  reg out;

  always @(s1 or s0 or i0 or i1 or i2 or i3)
    case ({s1,s0})
      2'd0: out = i0;
      2'd1: out = i1;
      2'd2: out = i2;
      2'd3: out = i3;
    endcase
endmodule
```

Multiway Branching (cont'd)

- The `case` statements compares <expression> and alternatives bit-for-bit
  - `x` and `z` values should match

```
module demultiplexer1_to_4(out0, out1, out2, out3, in, s1, s0);
  output out0, out1, out2, out3;
  input in, s1, s0;
  always @(s1 or s0 or in)
    case( {s1, s0} )
      2'b00: begin ... end
      2'b01: begin ... end
      2'b10: begin ... end
      2'b11: begin ... end
      2'bx0, 2'bx1, 2'bxz, 2'bxx, 2'b0x, 2'b1x, 2'bzx:
             begin ... end
      2'bz0, 2'bz1, 2'bzz, 2'b0z, 2'b1z:
             begin ... end
      default: $display("Unspecified control signals");
    endcase
endmodule
```

Multiway Branching (cont'd)

- `casex` and `casez` keywords
  - `casez` treats all `z` values as "don't care"
  - `casex` treats all `x` and `z` values as "don't care"
- Example:

```
reg [3:0]
integer state;
casex(encoding)
   4'b1xxx: next_state=3;
   4'bx1xx: next_state=2;
   4'bxx1x: next_state=1;
   4'bxxx1: next_state=0;
   default: next_state=0;
endcase
```

Behavioral Modeling Statements: Loops

- Loops in Verilog
  - `while, for, repeat, forever`
- The `while` loop syntax:

```
while (<expression>)
    statement;
```

- Example:
  - Look at p. 136 of your book

Loops (cont'd)

- The `for` loop
  - Similar to C
  - Syntax:

```
for( init_expr; cond_expr; change_expr)
    statement;
```

  - Example:
    - Look at p. 137 of your book

Loops (cont'd)

- The `repeat` loop
  - Syntax:

```
repeat( number_of_iterations )
    statement;
```

  - The number is evaluated only when the loop is first encoutered
  - Example:
    - Look at p. 138 of your book

Loops (cont'd)

- The `forever` loop
  - Syntax:

```
forever
    statement;
```

  - Equivalent to `while(1)`
  - Example:
    - Look at pp. 139-140 of your book

Procedural Assignments (cont'd)

- The two types of procedural assignments
  - Blocking assignments
  - Non-blocking assignments
- Blocking assignments
  - are executed in order (sequentially)
  - Example:

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial begin
  x=0; y=1; z=1;
  count=0;
  reg_a= 16'b0; reg_b = reg_a;
  #15 reg_a[2] = 1'b1;
  #10 reg_b[15:13] = {x, y, z};
  count = count + 1;
end
```

All executed at time 0

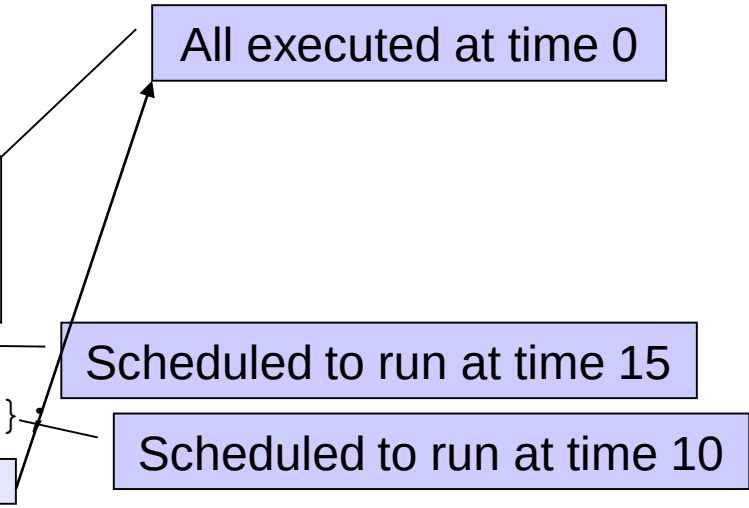
executed at time 15

All executed at time 25

Procedural Assignments (cont'd)

- Non-blocking assignments
  - The next statements are not blocked for this one
  - Syntax:
    - *<lvalue>* **<=** *<expression>*
  - Example:

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial begin
  x=0; y=1; z=1;
  count=0;
  reg_a= 16'b0; reg_b = reg_a;
  reg_a[2] <= #15 1'b1;
  reg_b[15:13] <= #10 {x, y, z};
  count <= count + 1;
end
```



Procedural Assignments (cont'd)

- Application of non-blocking assignments
  - Used to model concurrent data transfers
  - Example: Write behavioral statements to swap values of two variables
  - Another example

```
always @(posedge clock)
begin
   reg1 <= #1 in1;
   reg2 <= @(negedge clock) in2 ^ in3;
   reg3 <= #1 reg1;
end
```

The old value of reg1 is used

Procedural Assignments (cont'd)

- Race condition
  - When the final result of simulating two (or more) concurrent processes depends on their order of execution
- Example:

```
always @(posedge clock)
   b = a;
always @(posedge clock)
   a = b;
```

```
always @(posedge clock)
begin
  temp_b = b;
  temp_a = a;
  b = temp_a;
  a = temp_b;
end
```

Procedural Assignments (cont'd)

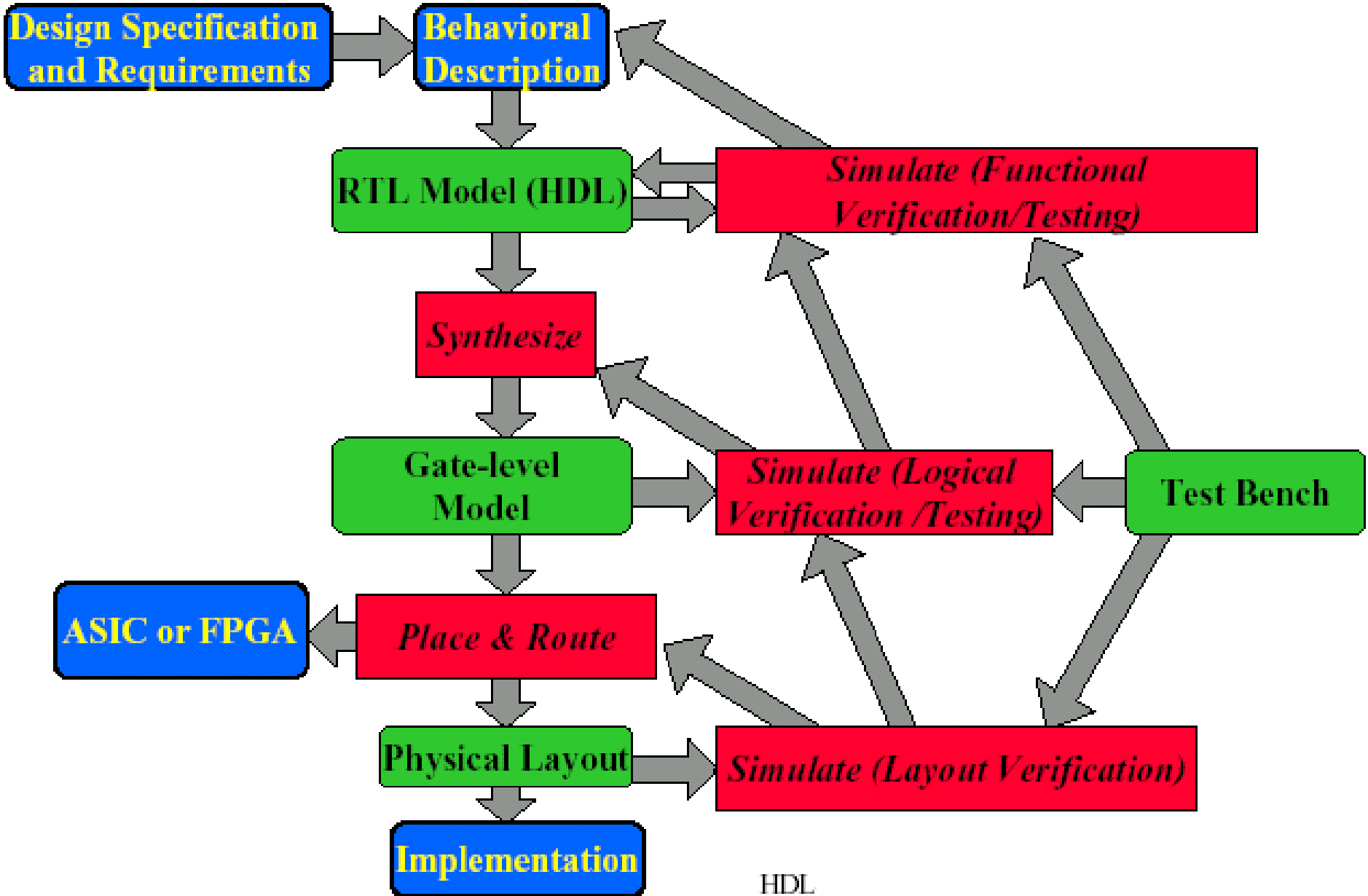
- Recommendation
  - Concurrent data transfers => race condition
  - Use non-blocking assignments wherever concurrent data transfers
  - Example: pipeline modeling
  - Disadvantage:
    - Lower simulation performance
    - Higher memory usage in the simulator

Verilog HDL -Introduction

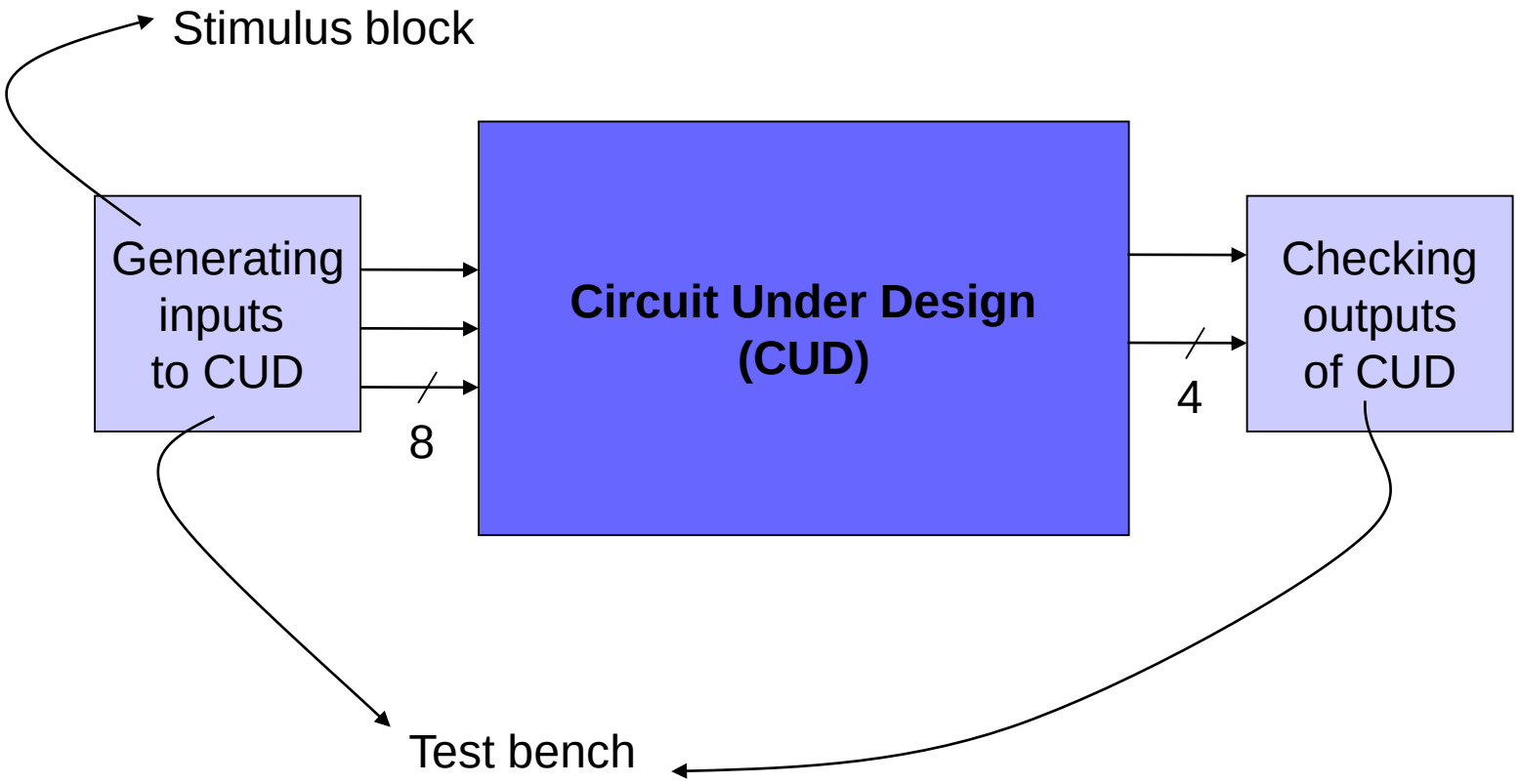
Typical Design Flow (in 1996)

1. Design specification
2. Behavioral description
3. RTL description
4. Functional verification and testing
5. Logic synthesis
6. Gate-level netlist
7. Logical verification and testing
8. Floor planning, automatic place & route
9. Physical layout

Design Methodology



Basics of Digital Design Using HDLs



Simulation- Test Bench Styles

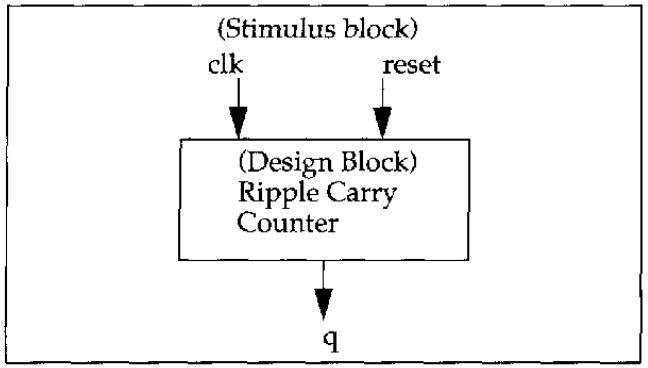


Figure 2-6 Stimulus Block Instantiates Design Block

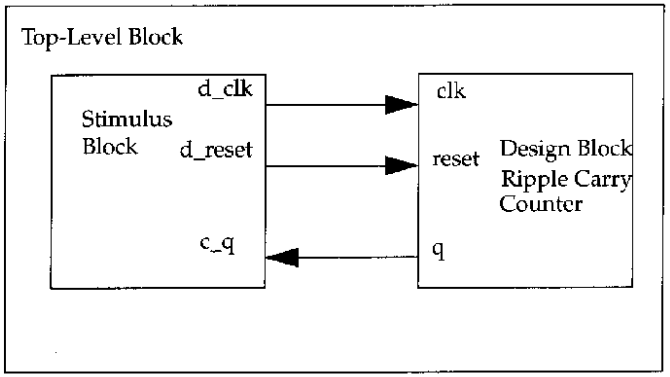


Figure 2-7 Stimulus and Design Blocks Instantiated in a Dummy Top-Level Module

Design Methodologies

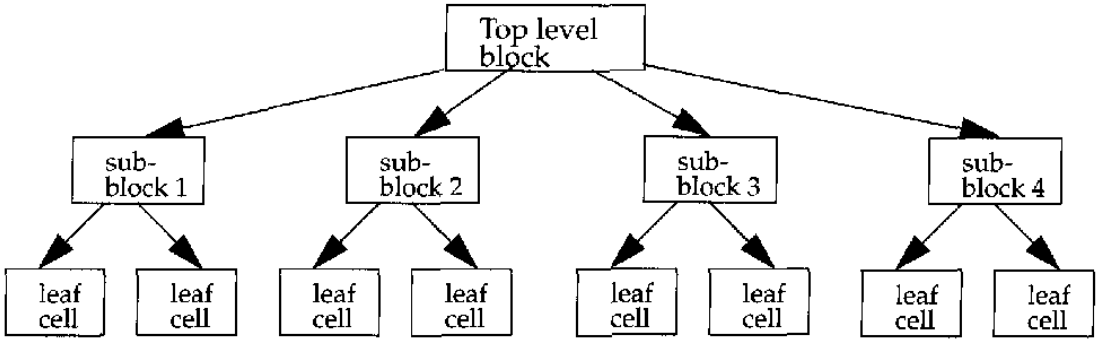


Figure 2-1 Top-down Design Methodology

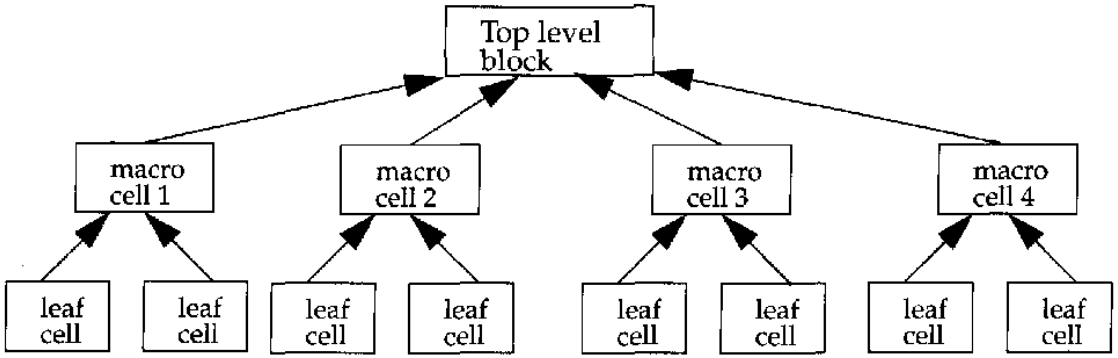


Figure 2-2 Bottom-up Design Methodology

4-bit Ripple Carry Counter

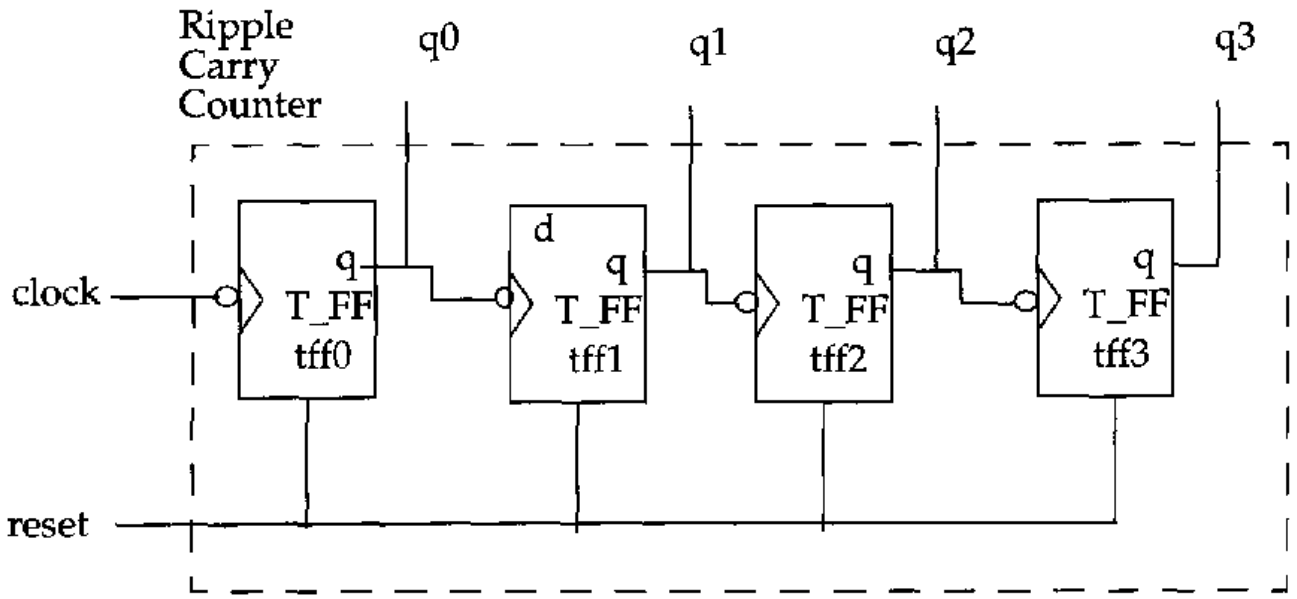


Figure 2-3 Ripple Carry Counter

T-flipflop and the Hierarchy

reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0

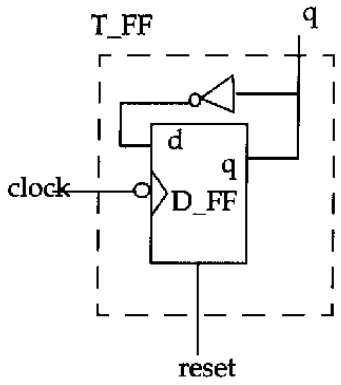


Figure 2-4 T-flipflop

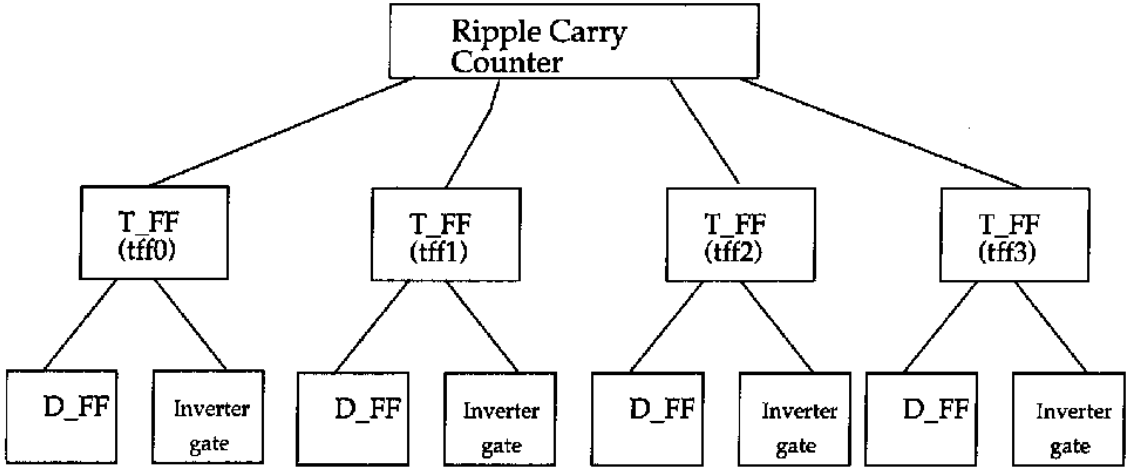
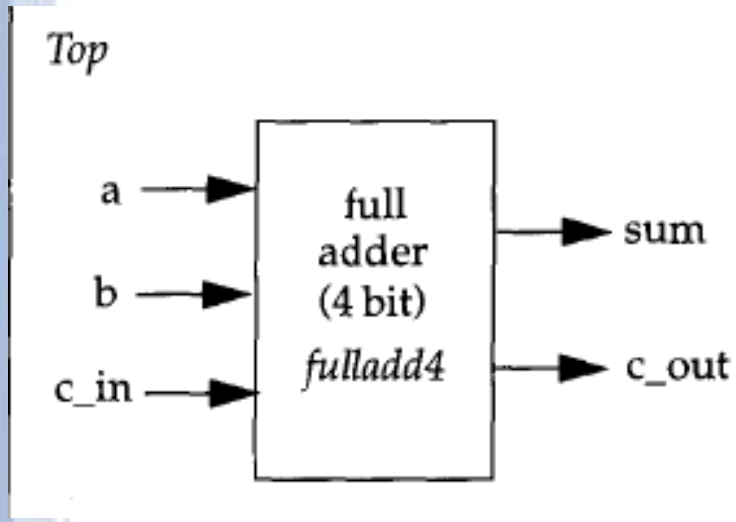


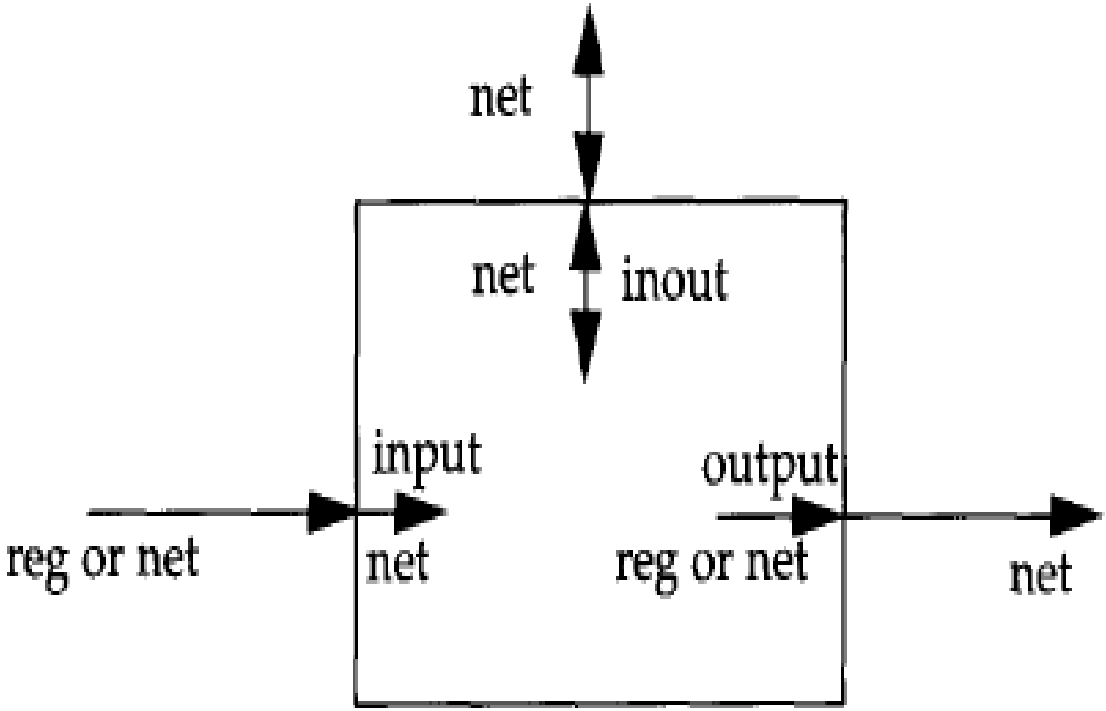
Figure 2-5 Design Hierarchy

Ports

- Ports provide interface for by which a module can communicate with its environment



Port connection rules



Connecting Ports

- Suppose we have a module

```
module fulladd4(sum, c_out, a, b, c_in);
```

```
module Top;

//Declare connection variables
reg [3:0]A,B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

    //Instantiate fulladd4, call it fa_ordered.
    //Signals are connected to ports in order (by position)
    fulladd4 fa_ordered(SUM, C_OUT, A, B, C_IN);
```

Module- Basic building block

```
module <module_name> (<module_terminal_list>);

...
<module internals>
...
...
endmodule
```

A module can be an element or collection of low level design blocks

Module

Module Name,
Port List, Port Declarations (if ports present)
Parameters(optional),

Declarations of **wire**s, **reg**s and other variables

Data flow statements (**assign**)

Instantiation of lower level modules

always and **initial** blocks. All behavioral statements go in these blocks.

Tasks and functions

endmodule statement

Modules (cont'd)

- Verilog supported levels of abstraction
  - Behavioral (algorithmic) level
    - Describe the algorithm used
    - Very similar to C programming

  - Dataflow level
    - Describe how data flows between registers and is processed
  - Gate levelInterconnect logic gates
  - Switch level
    - Interconnect transistors (MOS transistors)
- Register-Transfer Level (RTL)
  - Generally known as a combination of behavioral+dataflow that is synthesizable by EDA tools

Instance

- A module provides a template which you can create actual objects.
- When a module is invoked, Verilog creates a unique object from the template
- The process of creating a object from module template is called instantiation
- The object is called instance

Instances

```
module ripple_carry_counter(q, clk, reset);


 output [3:0] q;
 input clk, reset;


 //4 instances of the module TFF
 TFF tff0(q[0],clk, reset);
 TFF tff1(q[1],q[0], reset);
 TFF tff2(q[2],q[1], reset);
 TFF tff3(q[3],q[2], reset);


endmodule
```

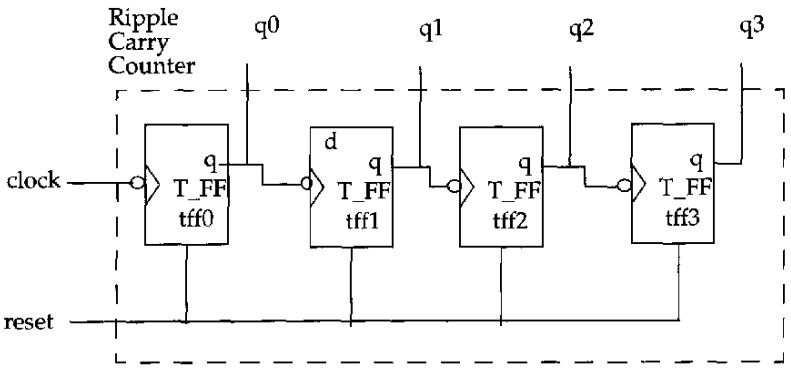


Figure 2-3 Ripple Carry Counter

Instances (cont'd)

```
module TFF(q, clk, reset);
 output q;
 input clk, reset;
 wire d;
 DFF dff0(q, d, clk, reset);
 not n1(d, q); // not is a Verilog provided primitive.
endmodule
// module DFF with asynchronous reset
module DFF(q, d, clk, reset);
 output q;
 input d, clk, reset;
 reg q;

 always @(posedge reset or negedge clk)
    if (reset)
    q = 1'b0;
    else
    q = d;
endmodule
```

How to build and test a module

Construct a “test bench” for your design

- Develop your hierarchical system within a module that has input and output ports (called “design” here)
- Develop a separate module to generate tests for the module (“test”)
- Connect these together within another module (“testbench”)

```
module testbench ();
      wire      l, m, n;

design   d (l, m, n);
test      t (l, m);

initial begin
      //monitor and display
      …
```

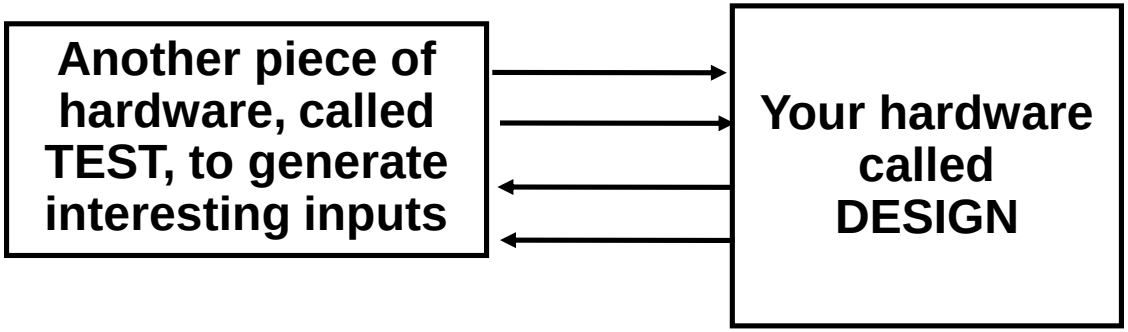
```
module design (a, b, c);
      input     a, b;
      output   c;
…
```

```
module test (q, r);
output   q, r;

initial begin
      //drive the outputs with signals
      …
```

Another view of this

- 3 chunks of verilog, one for each of:

TESTBENCH is the final piece of hardware which connect DESIGN with TEST so the inputs generated go to the design you want to test...



Verilog Examples

Module testAdd generated inputs for module halfAdd and displayed changes. Module halfAdd was the *design*

```
module tBench;
wire      su, co, a, b;

halfAdd ad(su, co, a, b);
testAdd tb(a, b, su, co);
endmodule
```

```
module halfAdd (sum, cOut, a, b);
output   sum, cOut;
input     a, b;

xor  #2   (sum, a, b);
and #2   (cOut, a, b);
endmodule
```

```
module testAdd(a, b, sum, cOut);
input     sum, cOut;
output   a, b;
reg  a, b;

initial begin
      $monitor ($time,,
         "a=%b, b=%b, sum=%b, cOut=%b",
          a, b, sum, cOut);
      a = 0; b = 0;
      #10 b = 1;
      #10 a = 1;
      #10 b = 0;
      #10 $finish;
end
endmodule
```

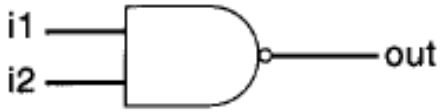
Gate Level Modeling

- A logic circuit can be designed by use of logic gates.
- Verilog supports basic logic gates as predefined *primitives.* These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition.

```
and     or      xor
nand    nor     xnor
```

Gate gate_name(out,in1,in2…)

and



nand

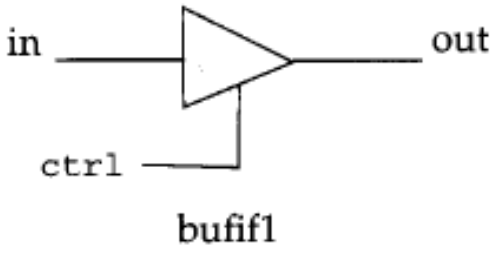
```
wire OUT, IN1, IN2;

// basic gate instantiations.
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);
or or1(OUT, IN1, IN2);
nor nor1(OUT, IN1, IN2);
xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);
```

Buf/not gates

- *Buf/not* gates have one scalar input and one or more scalar outputs.

```
// basic gate instantiations.
buf b1(OUT1, IN);
not n1(OUT1, IN);

// More than two outputs
buf b1_2out(OUT1, OUT2, IN);

// gate instantiation without instance name
not (OUT1, IN); // legal gate instantiation
```

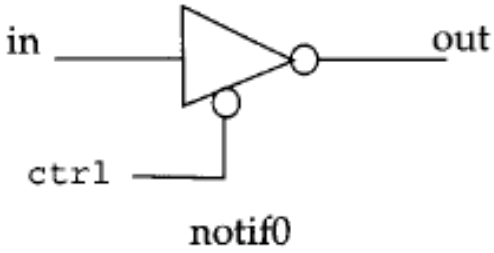
Bufif/notif



bufif1

bufif1	ctrl 0	1	x	z
in 0	z	0	L	L
1	z	1	H	H
x	z	x	x	x
z	z	x	x	x

notif0	ctrl 0	1	x	z
in 0	1	z	H	H
1	0	z	L	L
x	x	z	x	x
z	x	z	x	x

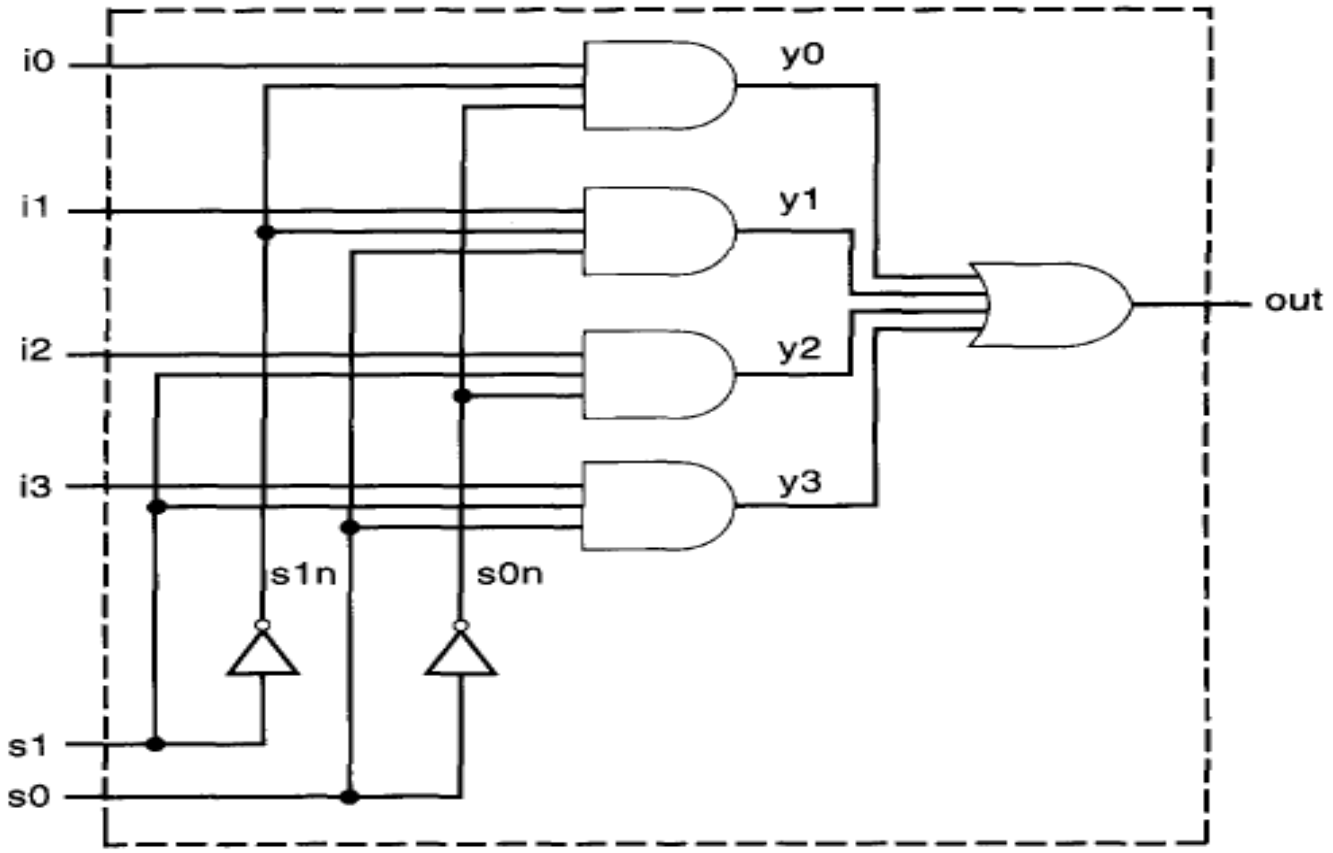


notif0

Instantiation of bufif gates

```
//Instantiation of bufif gates.
bufif1 b1 (out, in, ctrl);
bufif0 b0 (out, in, ctrl);

//Instantiation of notif gates
notif1 n1 (out, in, ctrl);
notif0 n0 (out, in, ctrl);
```

Design of 4:1 Multiplexer



Contd..

```
// Module 4-to-1 multiplexer. Port list is taken exactly from
// the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
```

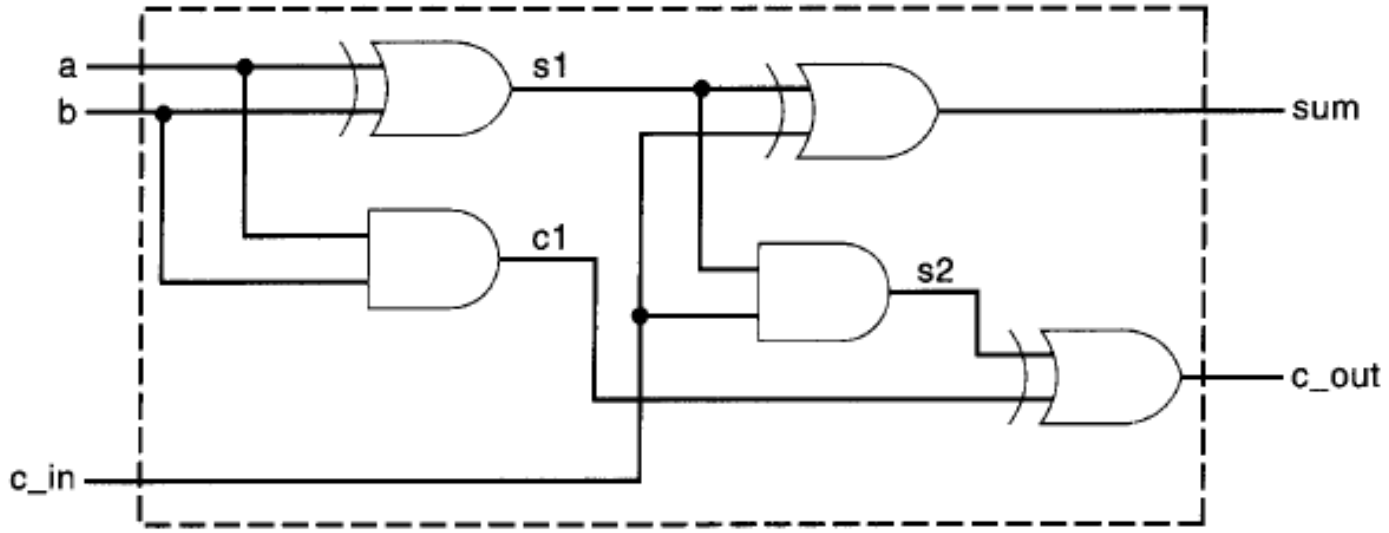
```
// Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;

// Gate instantiations

// Create s1n and s0n signals.
not (s1n, s1);
not (s0n, s0);

// 3-input and gates instantiated
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);

// 4-input or gate instantiated
or (out, y0, y1, y2, y3);

endmodule
```

Stimulus

```
// Define the stimulus module (no ports)
module stimulus;

// Declare variables to be connected
// to inputs
reg IN0, IN1, IN2, IN3;
reg S1, S0;

// Declare output wire
wire OUTPUT;

// Instantiate the multiplexer
mux4_to_1 mymux(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);
```

```
// Define the stimulus module (no ports)

// Stimulate the inputs
initial
begin
  // set input lines
  IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
  #1 $display("IN0= %b, IN1= %b, IN2= %b, IN3=
%b\n",IN0,IN1,IN2,IN3);

  // choose IN0
  S1 = 0; S0 = 0;
  #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

  // choose IN1
  S1 = 0; S0 = 1;
  #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

  // choose IN2
  S1 = 1; S0 = 0;
  #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

  // choose IN3
  S1 = 1; S0 = 1;
  #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
end

endmodule
```

4 bit full adder



$$sum = (a \oplus b \oplus cin)$$

$$cout = (a \cdot b) + cin \cdot (a \oplus b)$$

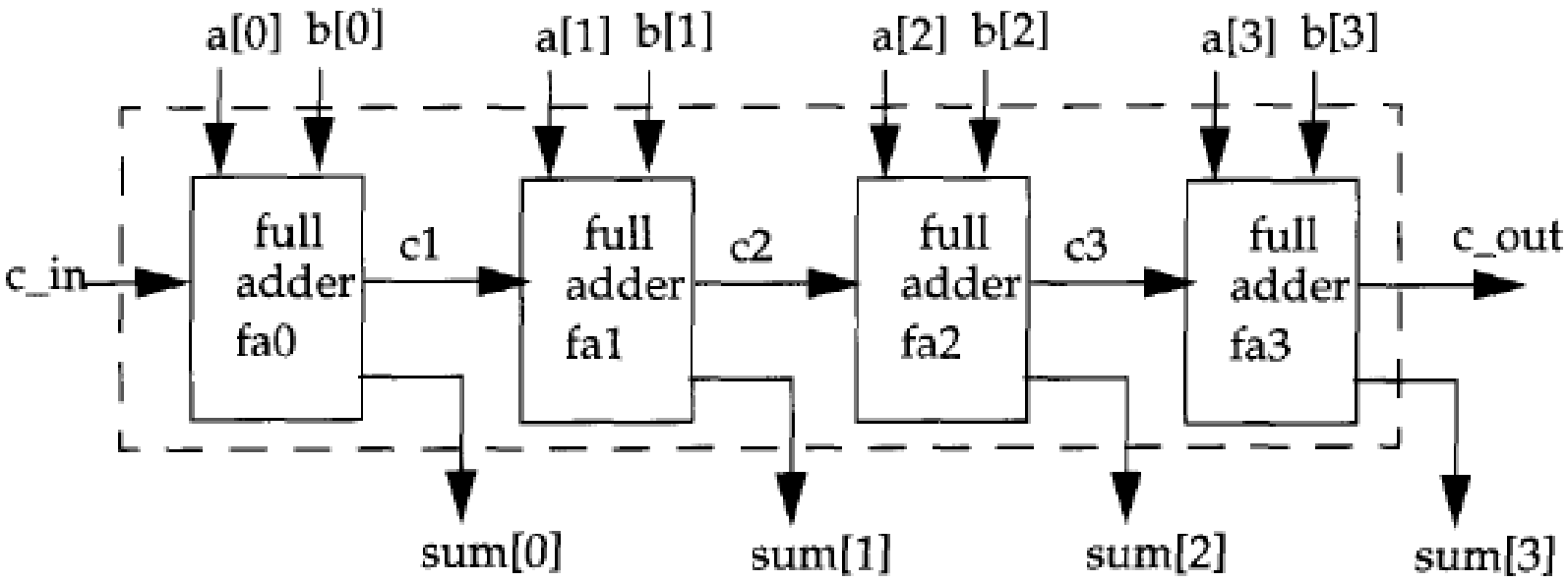
Declaration:

```
// Define a 1-bit full adder
module fulladd(sum, c_out, a, b, c_in);

// I/O port declarations
output sum, c_out;
input a, b, c_in;

// Internal nets
wire s1, c1, c2;
```

Code contd..

```
// Instantiate logic gate primitives
xor (s1, a, b);
and (c1, a, b);

xor (sum, s1, c_in);
and (c2, s1, c_in);

or  (c_out, c2, c1);

endmodule
```

4 bit adder using 1 bit adder



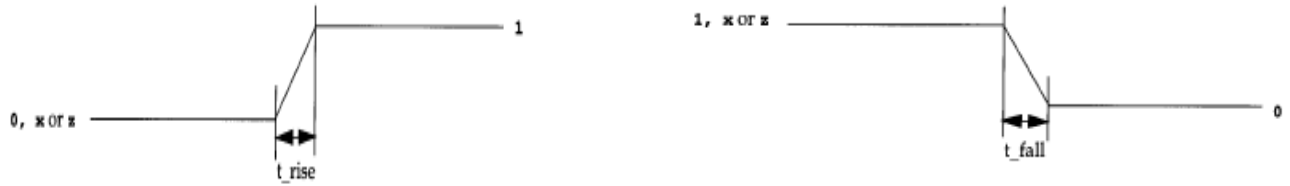
```
// Define a 4-bit full adder
module fulladd4(sum, c_out, a, b, c_in);

// I/O port declarations
output [3:0] sum;
output c_out;
input[3:0] a, b;
input c_in;

// Internal nets
wire c1, c2, c3;

// Instantiate four 1-bit full adders.
fulladd fa0(sum[0], c1, a[0], b[0], c_in);
fulladd fa1(sum[1], c2, a[1], b[1], c1);
fulladd fa2(sum[2], c3, a[2], b[2], c2);
fulladd fa3(sum[3], c_out, a[3], b[3], c3);

endmodule
```

Stimulus

```
// Define the stimulus (top level module)
module stimulus;

// Set up variables
reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

// Instantiate the 4-bit full adder. call it FA1_4
fulladd4 FA1_4(SUM, C_OUT, A, B, C_IN);
```

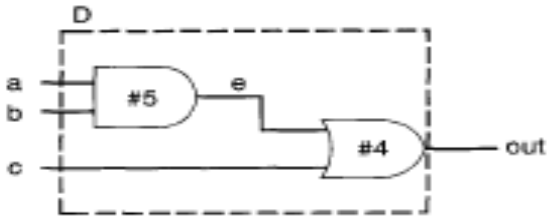
Gate Delays:

- Rise Delay: Delay associated with a o/p transition to 1 from any value.

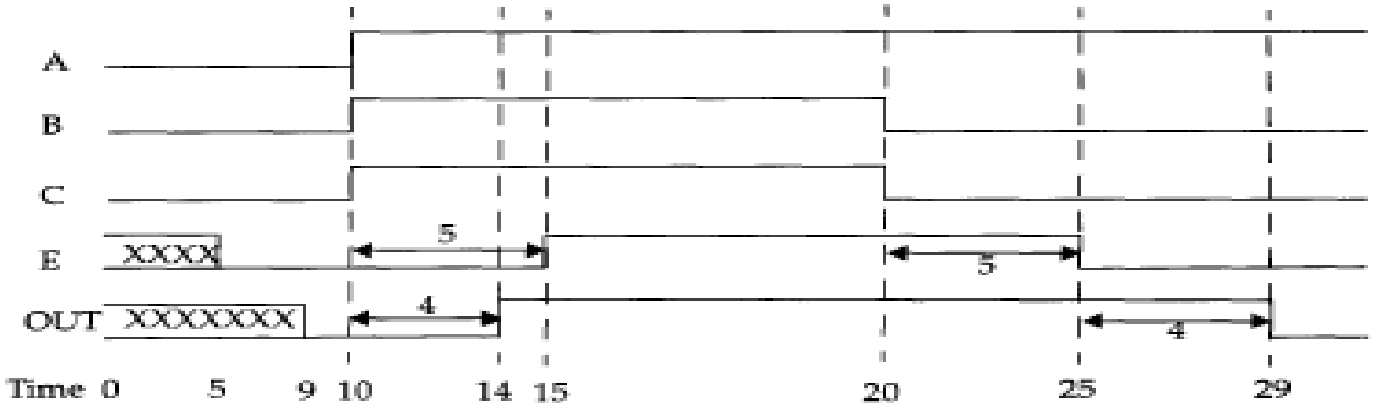


Fall Delay: Delay associated with o/p transition to 0 from any value.

Turn off Delay: Delay associate with o/p transition to Z from another value.




```
A= 1'b0; B= 1'b0; C= 1'b0;

#10 A= 1'b1; B= 1'b1; C= 1'b1;

#10 A= 1'b1; B= 1'b0; C= 1'b0;
```



Min value

The min vale is the minimum delay value that the designer expects the gate to have.

Type value

The type value is the typical delay value that the designer expects the gate to have.

Max value

The max value is the maximum delay value that the designer expects the gate to have.

```
//min delay=4
//type delay=5
//max delay=6
and #(4:5:6)  a1(out, i1, i2) ;

//min delay, rise=3 , fall =5
//type delay, rise=4 , fall =6
//max delay, rise=5 , fall =7
and #(3:4:5, 5:6:7) a2(out, i1, i2 ) ;

//min delay, rise=2 , fall =3 , turn-off =4
//type delay, rise=3 , fall =4 , trun-off=5
//max delay, rise=4 , fall =5 , trun-off=6

and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1, i2 ) ;
```

Dataflow Modeling

- In complex designs the number of gates is very large
- Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called *logic synthesis*

Continuous Assignment

```
//Syntax of assign statement in the simplest form
<continuous_assign>
        ::= assign <drive_strength>?<delay>? <list_of_assignments>;


// Continuous assign. out is a net. i1 and i2 are nets.
assign out = i1 & i2;

// Continuous assign for vector nets. addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers.
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

// Concatenation. Left-hand side is a concatenation of a scalar
// net and a vector net.
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

Rules:

- The left hand side of an assignment must always be a scalar or vector net
- It cannot be a scalar or vector register.
- Continuous assignments are always active.
- The assignment expression is evaluated as soon as one of the right-hand-side operands changes and the value is assigned to the left-hand-side net.

- The operands on the right-hand side can be registers or nets.

- Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value

- Implicit Continious Assignment

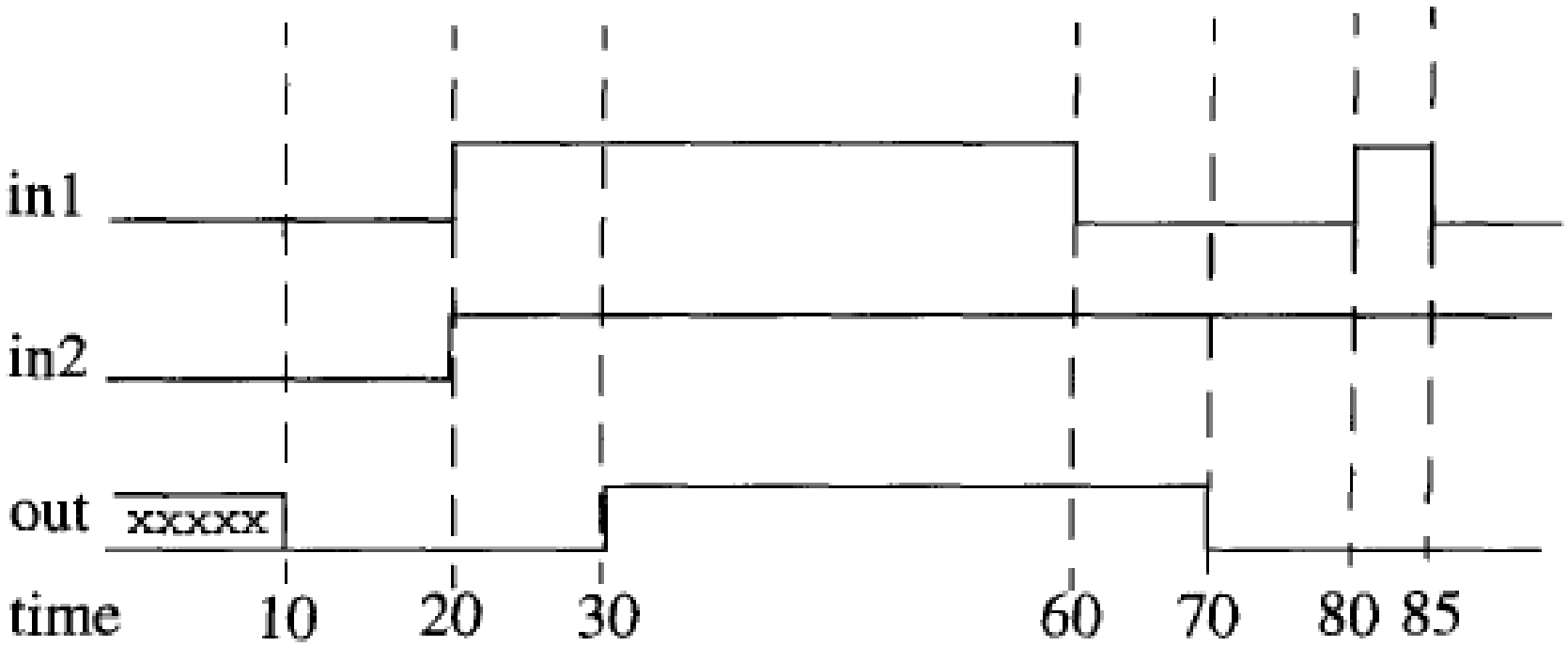
```
//Regular Continious Assignment
wire= out;
assign out = in1& in2;
//same effect is achieved by an implicit assignment
Wire out = in1& in2;
```

- Implicit Net Declaration

```
//Continious Assignment,out is a net
Wire i1, i2;
assign out = in1& in2;
```

Delay

```
assign #10 out = in1 & in2; // Delay in a continuous assign
```



Implicit Continuous Assignment Delay

```
wire #10 out = in1 & in2;
```

Net Declaration Delay

```
wire #10 out ;

assign  out = in1 & in2;
```

Operator Types

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two
Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Shift	>>	Right shift	two
	<<	Left shift	two
Concatenation	{ }	Concatenation	any number
Replication	{ { } }	Replication	any number
Conditional	?:	Conditional	three
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one

Conditional Operator

Usage: condition_expr ? true_expr : false_expr ;

```
//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
```

4:1 Multiplexer Example

```
// Module 4-to-1 multiplexer using data flow. logic equation
// Compare to gate-level model
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

//Logic equation for out
assign out =   (~s1 & ~s0 & i0)|
               (~s1 & s0 & i1) |
               (s1 & ~s0 & i2) |
               (s1 & s0 & i3) ;

endmodule
```

User Defined Primitives (UDPs)

- Keywords **and**, **or**, **not**, **xor**, etc. are System Primitives
- Can Define your Own Primitives (UDPs)
- Can do this in a variety of ways including Truth Tables
- Instead of **module/endmodule** use the keywords **primitive/endprimitive**
- Only one output and must be listed first
- Keywords **table** and **endtable** used
- Input values listed in order
- Output is always last entry

HDL Example 2

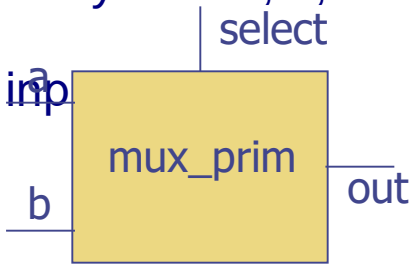
```
//User defined primitive(UDP)
primitive crctp (x,A,B,C);                    // user defined
   output x;
   input A,B,C;
//Truth table for x(A,B,C) = Minterms ( ? )
   table                                                  // truth table
//     A  B  C  :  x  (Note that this is only a comment)
       0   0   0  :  1;
       0   0   1  :  0;
       0   1   0  :  1;
       0   1   1  :  0;
       1   0   0  :  1;
       1   0   1  :  0;
       1   1   0  :  1;
       1   1   1  :  1;
   endtable
endprimitive
```

Primitives

- Pre-defined primitives
  - Total 26 pre-defined primitives
  - All combinational
  - Tri-state primitives have multiple output, others have single output
- User-Defined Primitives (UDP)
  - Combinational or sequential
  - Single output
- UDP vs. modules
  - Used to model cell library

Shorthand Notation

```
primitive mux_prim ( out, select, a, b );

   output      out;

   input       select, a, b;

   table

//select  a   b     :   out

    0      0   ?     :   0;      // ? => iteration of table entry over 0, 1, x.

    0      1   ?     :   1;      // i.e., don't care on the inp

    1      ?   0     :   0;

    1      ?   1     :   1;

    ?      0   0     :   0;
```

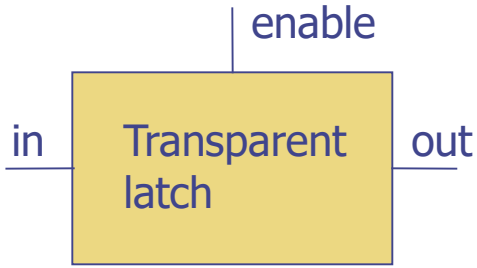


UDP: Sequential Behavior

- In table description, n+2 columns for n input
- n input columns + internal state column + output (next state) column
- Output port -> **reg** variable

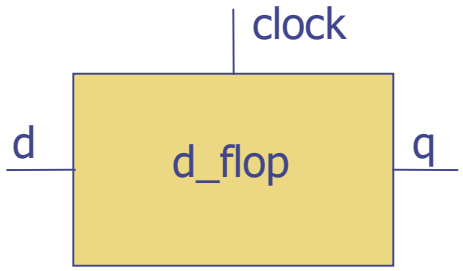
Level-sensitive Behavior

```
primitive transparent_latch(out, enable, in);
    output    out;
    input     enable, in;
    reg       out;
    table
//enable in   state    out/next_state
    1      1   :?    :    1;
    1      0   :?    :    0;
    0      ?   :?    :    -;    // '-' -> no change
    x      0   :0    :    -;
```



Edge-sensitive Behavior

```
primitive d_flop( q, clock, d );
    output   q;
    input    clock, d;
    reg      q;
    table
// clock  d    state    q/next_state
    (01)  0    :    ?    :    0;   // Parentheses indicate signal transition
    (01)  1    :    ?    :    1;   // Rising clock edge
    (0?)  1    :    1    :    1;
    (0?)  0    :    0    :    0;
    (?0)  ?    :    ?    :    -;   // Falling clock edge
```



Digital System Design

Verilog® HDL
Timing and Delays

Maziar Goudarzi

Today Program

- Delays and their definition and use in Verilog

Introduction:
Delays and Delay Back-annotation

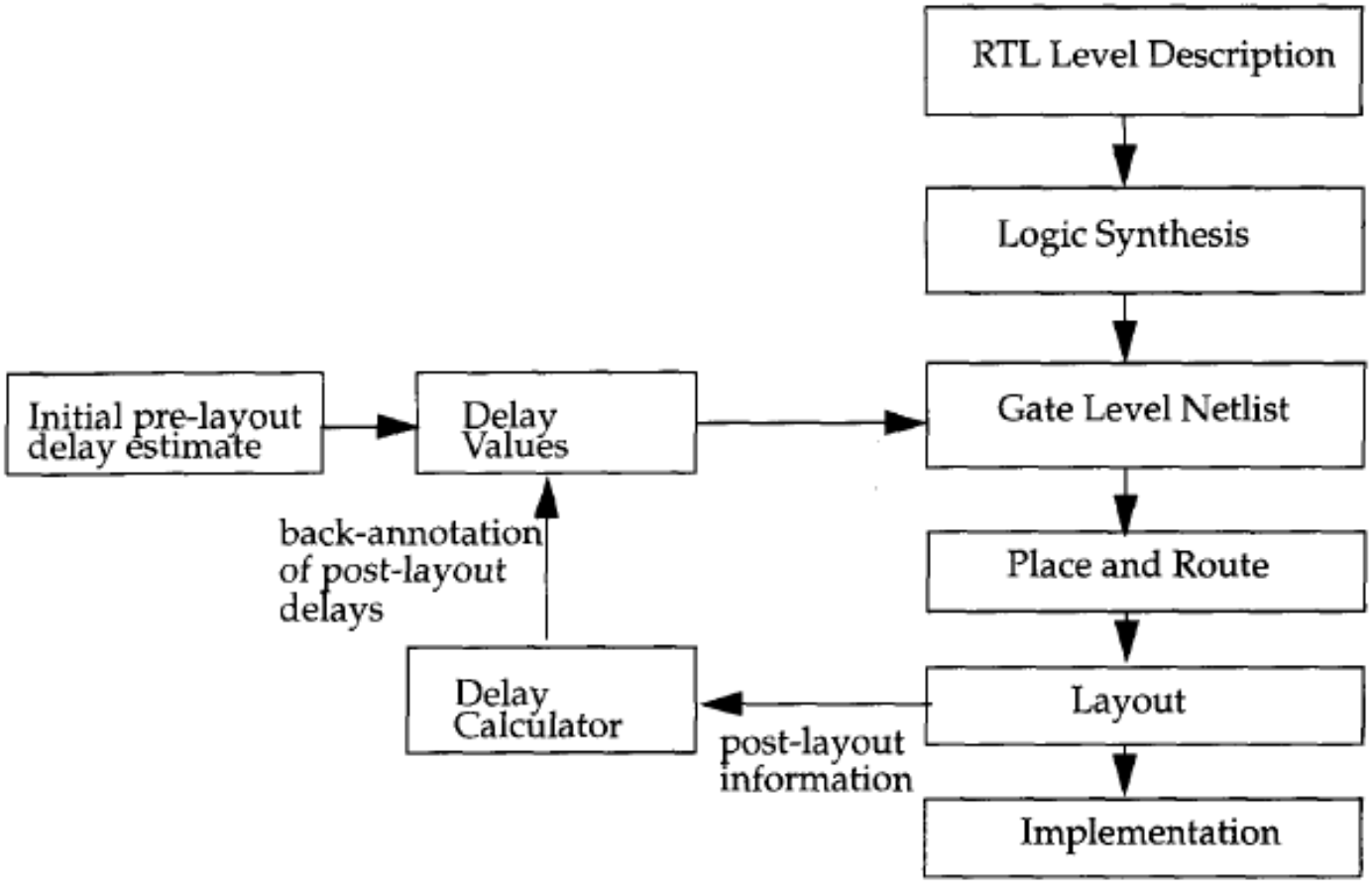


Figure 10-7 Delay Back-Annotation

Introduction (cont'd)

- Functional simulation vs. Timing simulation
- Delays are crucial in *REAL* simulations
  - Post-synthesis simulation
  - Post-layout simulation
    - FPGA counter-part: Post-P&R simulation
- Delay Models
  - Represent different physical concepts
  - Two most-famous models
    - Inertial delay
    - Transport delay (path delay)

Delay Models: Inertial Delay

- The inertia of a circuit node to change value
- Abstractly models the RC circuit seen at the node
- Different types
  - Input inertial delay
  - Output inertial delay

Delay Models: Transport Delay (Path Delay)

- Represents the propagation time of signals from module inputs to its outputs
- Models the internal propagation delays of electrical elements

Specifying Delays in Verilog

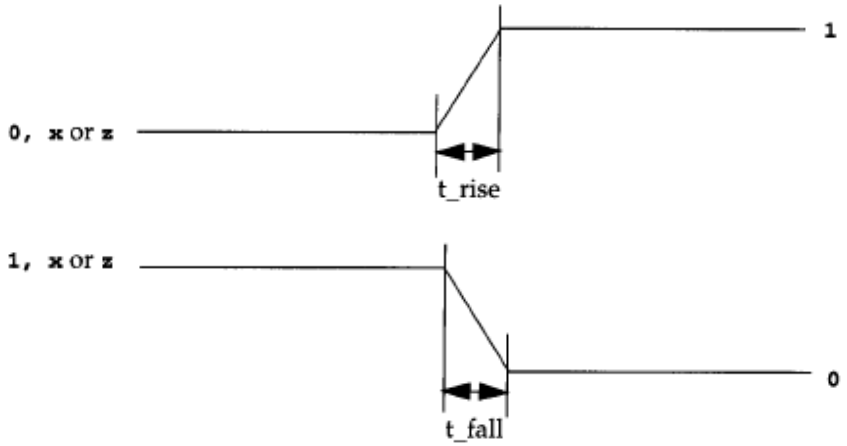
Delays in Verilog

Specifying Delays in Verilog

- Delays are shown by # sign in all Verilog modeling levels
- Supported delay types
  - Rise, Fall, Turnoff types
  - Min, Typ, Max values

Delay Types in Verilog

- Rise Delay
  - From any value to 1
- Fall Delay
  - From any value to 0
- Turn-Off Delay
  - From any value to z



- From any value to x
  - Minimum of the three delays
- Min/Typ/Max Delay values

Specifying Delays in Verilog (cont'd)

- Rise/Fall/Turnoff delay types (cont'd)
  - If no delay specified
    - Default value is zero
  - If only one value specified
    - It is used for all three delays
  - If two values specified
    - They refer respectively to rise and fall delays
    - Turn-off delay is the minimum of the two

Specifying Delays in Verilog (cont'd)

- Min/Typ/Max Values
  - Another level of delay control in Verilog
  - Each of rise/fall/turnoff delays can have min/typ/max values

```
not #(min:typ:max, min:typ:max, min:typ:max) n(out,in)
```

  - Only one of Min/Typ/Max values can be used in the entire simulation run
    - It is specified at start of simulation, and depends on the simulator used
    - ModelSim options

      ModelSim> vsim +mindelays
      ModelSim> vsim +typdelays
      ModelSim> vsim +maxdelays
    - Typ delay is the default

Specifying Delays in Verilog (cont'd)

- General syntax

```
#(rise_val, fall_val, turnoff_val)
#(min:typ:max, min:typ:max, min:typ:max)
```

- Gate-Level and Dataflow Modeling
  - All of the above syntaxes are valid
- Behavioral Modeling
  - Rise/fall/turnoff delays are not supported
  - Only `min:typ:max` values can be specified
    - Applies to all changes of values

Delays in Gate-Level Modeling

Delays in Verilog

Delays in Gate-Level Modeling

- The specified delays are *output-inertial* delays

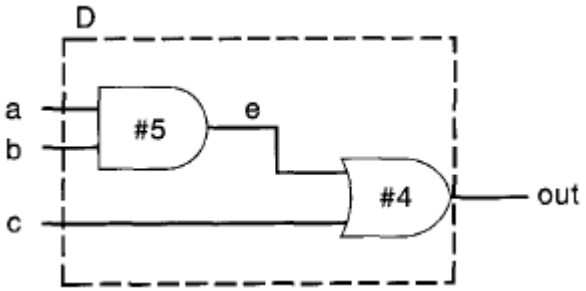
```
and #(rise_val, fall_val, turnoff_val) a(out,in1, in2)
not #(min:typ:max, min:typ:max, min:typ:max) b(out,in)
```

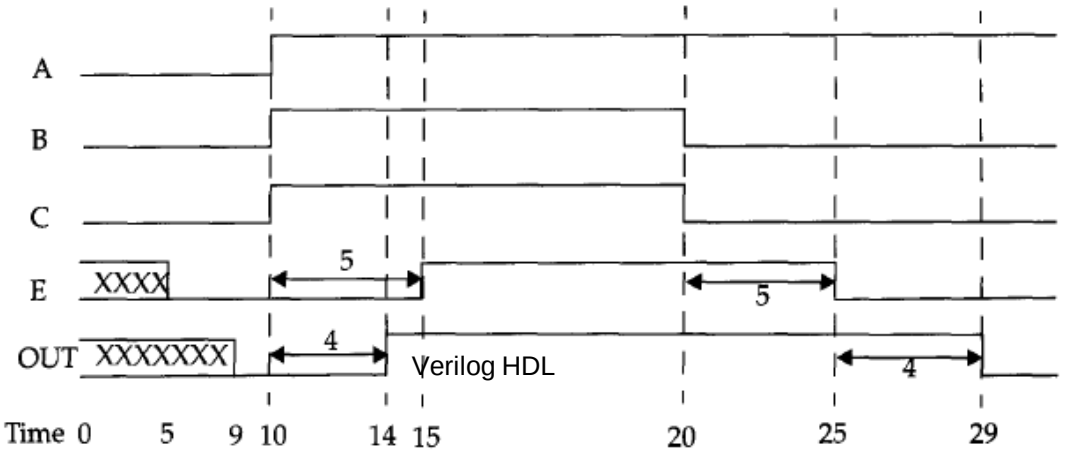
- Examples:

```
and #(5) a1(out, i1, i2);
and #(4, 6) a2(out, i1, i2);
and #(3, 4, 5) a2(out, i1, i2);
and #(1:2:3, 4:5:6, 5:6:7) a2(out, i1, i2);
```

Delays in Gate-Level Modeling (cont'd)

```
// Define a simple combination module called D
module D (out, a, b, c);

// I/O port declarations
output out;
input a,b,c;

// Internal nets
wire e;

// Instantiate primitive gates to build the circuit
and #(5) a1(e, a, b); //Delay of 5 on gate a1

or  #(4) o1(out, e,c); //Delay of 4 on gate o1

endmodule
```





Delays in Gate-Level Modeling (cont'd)

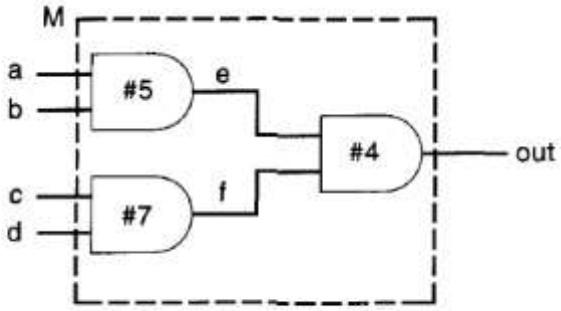


Figure 10-1 Distributed Delay

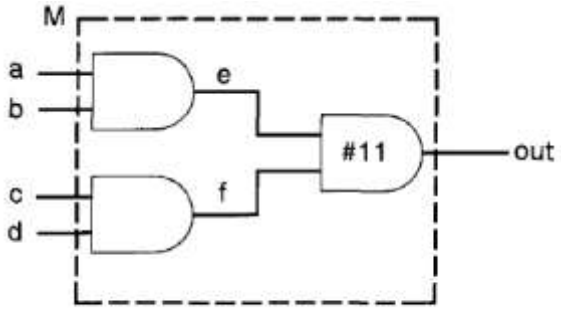


Figure 10-2 Lumped Delay

```
//Distributed delays in gate-level modules
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

//Delay is distributed to each gate.
and #5 a1(e, a, b);
and #7 a2(f, c, d);
and #4 a3(out, e, f);
endmodule
```

```
//Lumped Delay Model
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

and a1(e, a, b);
and a2(f, c, d);
and #11 a3(out, e, f);//delay only on the output gate
endmodule
```

Delays in Dataflow Modeling

Delays in Verilog

Delays in Dataflow Modeling

- As in Gate-Level Modeling the delay is *output-inertial delay*
- Regular assignment delay syntax

```
assign #delay out = in1 & in2;
```

- Implicit continuous assignment delay

```
wire #delay out = in1 & in2;
```

- Net declaration delay
  - Can also be used in Gate-Level modeling

```
wire #delay w;
```

Delays in Dataflow Modeling (cont'd)

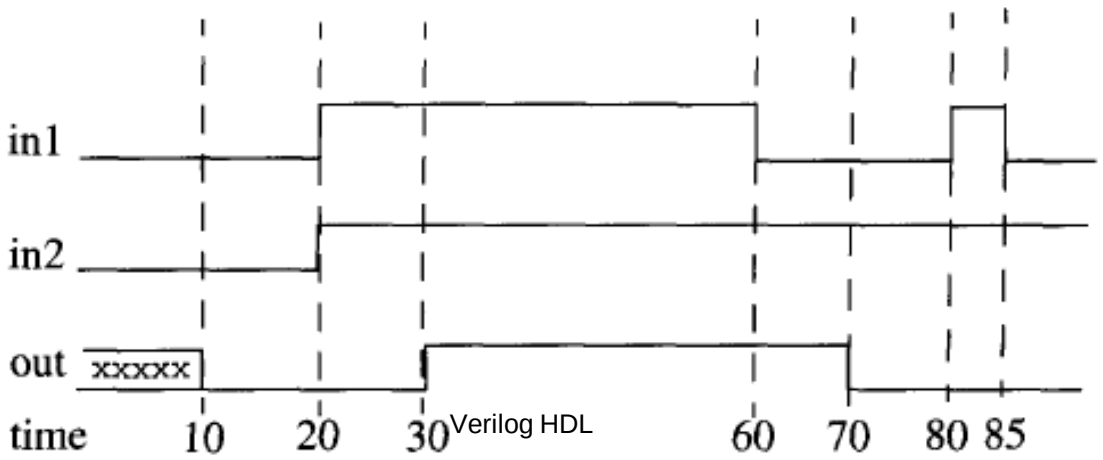
- Examples

```
wire #10 out = in1 & in2;
```

```
wire out;
assign #10 out = in1 & in2;
```

```
wire #10 out;
assign out = in1 & in2;
```

Note: pulses with a width less than the delay are not propagated to output



Delays in Dataflow Modeling (cont'd)

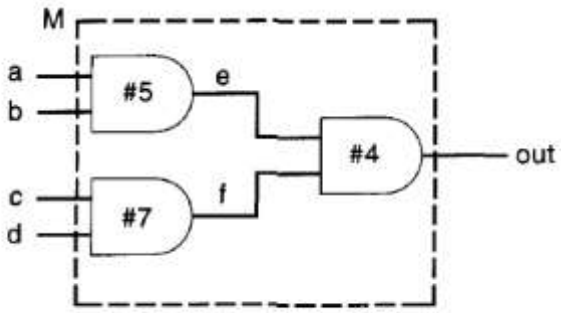


Figure 10-1 Distributed Delay

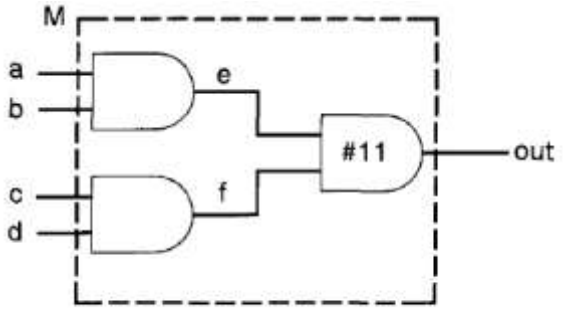


Figure 10-2 Lumped Delay

```
//Distributed delays in data flow definition of a module
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

//Distributed delay in each expression
assign #5 e = a & b;
assign #7 f = c & d;
assign #4 out = e & f;
endmodule
```

```
// Lumped delays in dataflow modeling
module M(out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

// Lumped delay model
assign e=a & b;
assign f=c & d;
assign #11 out = e & f;

endmodule
```

Delays in Behavioral Modeling

Delays in Verilog

Delay in Behavioral Modeling

- Only `min:typ:max` values can be set
  - i.e. rise/fall/turnoff delays are not supported
- Three categories
  - Regular delays
  - Intra-assignment delays
  - Zero delay

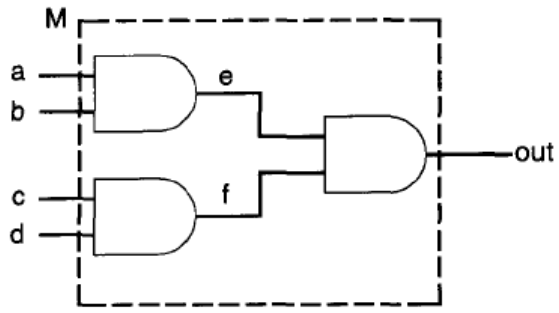
Path (Transport) Delays in Verilog

Delays in Verilog

Transport Delays in Verilog

- Also called
  - Pin-to-Pin delay
  - Path delay
- Gate-level, dataflow, and behavioral delays
  - Property of *the elements* in the module (white box)
    - Styles: Distributed or Lumped
- Path delay
  - A property of *the module* (black box)
  - Delay from any input to any output port

Transport Delays in Verilog (cont'd)



path a–e–out, delay = 9
path b–e–out, delay = 9
path c–f–out, de;ay = 11
path d–f–out, delay = 11

Example 10-3 Pin-to-Pin Delay

```
//Pin-to-pin delays
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

//Specify block with path delay statements
specify
    (a => out) = 9;
    (b => out) = 9;
    (c => out) = 11;
    (d => out) = 11;
endspecify

//gate instantiations
and a1(e, a, b);
and a2(f, c, d);
and a3(out, e, f);
endmodule
```

Transport Delays in Verilog (cont'd)

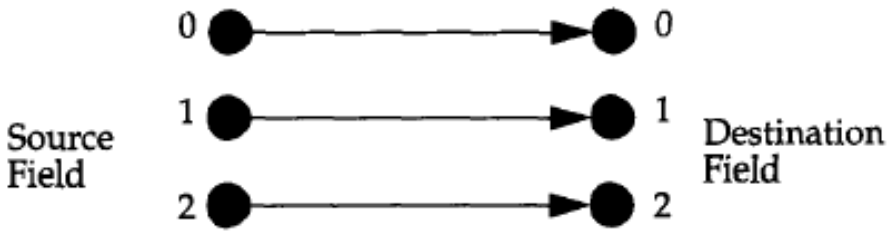
- `specify` block
  - Assign pin-to-pin delays
  - Define `specparam` constants
  - Setup timing checks in the design

`specify` blocks

- Parallel connection
  - Syntax:

```
specify
    (<src_field> => <dest_field>) = <delay>;
endspecify
```

  - `<src_field>` and `<dest_field>` are vectors of equal length
    - Unequal lengths, compile-time error

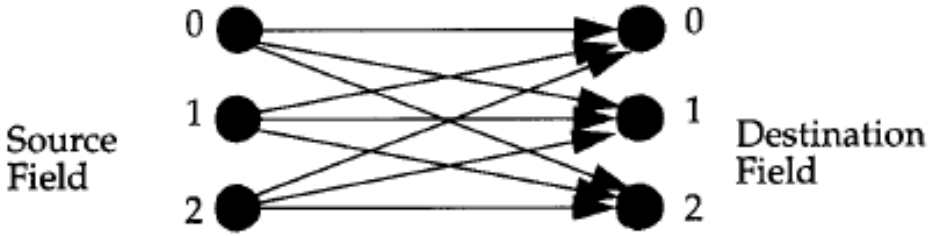


`specify` blocks (cont'd)

- Full connection
  - Syntax:

```
specify
    (<src_field> *> <dest_field>) = <delay>;
endspecify
```

  - No need to equal lengths in `<src_field>` and `<dest_field>`



`specify` blocks (cont'd)

```
//Full Connection
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

//full connection
specify
(a,b *> out) = 9;
(c,d *> out) = 11;
endspecify

and a1(e, a, b);
and a2(f, c, d);
and a3(out, e, f);
endmodule
```

```
//a[31:0] is a 32-bit vector and out[15:0] is a 16-bit vector
//Delay of 9 between each bit of a and every bit of out

specify
( a *> out) = 9; // you would need 32 X 16 = 352 parallel connection
                 // statements to accomplish the same result! Why?
endspecify
```

`specify` blocks (cont'd)

- `specparam` constants
  - Similar to `parameter`, but only inside `specify` block
  - Recommended to be used instead of hard-coded delay numbers

```
//Specify parameters using specparam statement
specify
    //define parameters inside the specify block
    specparam d_to_q = 9;
    specparam clk_to_q = 11;


    (d => q) = d_to_q;
    (clk => q) = clk_to_q;
endspecify
```

`specify` blocks (cont'd)

- Conditional path delays
  - Delay depends on signal values
  - Also called *State-Dependent Path Delay (SDPD)*

```
//Conditional Path Delays
module M (out, a, b, c, d);
output out;
input a, b, c, d;

wire e, f;

//specify block with conditional pin-to-pin timing
specify

//different pin-to-pin timing based on state of signal a.
if (a) (a => out) = 9;
if (~a) (a => out) = 10;

//Conditional expression contains two signals b , c.
//If b & c is true, delay = 9,
//otherwise delay = 13.
if (b & c) (b => out) = 9;
if (~(b & c)) (b => out) = 13;

//Use concatenation operator.
//Use Full connection
if ({c,d} == 2'b01)
//Conditional Path Delays
        (c,d *> out) = 11;
if ({c,d} != 2'b01)
        (c,d *> out) = 13;

endspecify

and a1(e, a, b);
and a2(f, c, d);
and a3(out, e, f);
endmodule
```

`specify` blocks (cont'd)
Rise, Fall, and Turn-off delays

```
//Specify one delay only. Used for all transitions.
specparam t_delay = 11;
(clk => q) = t_delay;

//Specify two delays, rise and fall
//Rise used for transitions 0->1, 0->z, z->1
//Fall used for transitions 1->0, 1->z, z->0
specparam t_rise = 9, t_fall = 13;
(clk => q) = (t_rise, t_fall);

//Specify three delays, rise, fall, and turn-off
//Rise used for transitions 0->1, z->1
//Fall used for transitions 1->0, z->0
//Turn-off used for transitions 0->z, 1->z
specparam t_rise = 9, t_fall = 13, t_turnoff = 11;
(clk => q) = (t_rise, t_fall, t_turnoff);
```

`specify` blocks (cont'd)
Rise, Fall, and Turn-off delays

```
//specify six delays.
//Delays are specified in order
//for transitions 0->1, 1->0, 0->z, z->1, 1->z, z->0. Order
//must be followed strictly.
specparam t_01 = 9, t_10 = 13, t_0z = 11;
specparam t_z1 = 9, t_1z = 11, t_z0 = 13;
(clk => q) = (t_01, t_10, t_0z, t_z1, t_1z, t_z0);

//specify twelve delays.
//Delays are specified in order
//for transitions 0->1, 1->0, 0->z, z->1, 1->z, z->0
//                0->x, x->1, 1->x, x->0, x->z, z->x.
//Order must be followed strictly.
specparam t_01 = 9, t_10 = 13, t_0z = 11;
specparam t_z1 = 9, t_1z = 11, t_z0 = 13;
specparam t_0x = 4, t_x1 = 13, t_1x = 5;
specparam t_x0 = 9, t_xz = 11, t_zx = 7;
(clk => q) = (t_01, t_10, t_0z, t_z1, t_1z, t_z0,
              t_0x, t_x1, t_1x, t_x0, t_xz, t_zx );
```

`specify` blocks (cont'd)
Min, Typ, Max delays

- Any delay value can also be specified as `(min:typ:max)`

```
//Specify three delays, rise, fall, and turn-off
//Each delay has a min:typ:max value
specparam t_rise = 8:9:10, t_fall = 12:13:14, t_turnoff = 10:11:12;
(clk => q) = (t_rise, t_fall, t_turnoff);
```

`specify` blocks (cont'd)

- Handling x transitions
  - Pessimistic approach
    - Transition to x: minimum possible time
    - Transition from x: maximum possible time

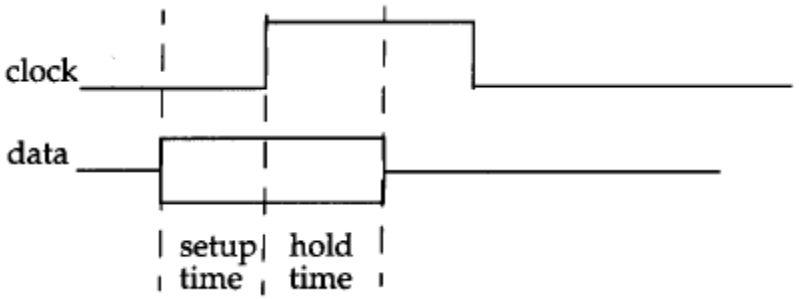
```
//Six delays specified .
//for transitions 0->1, 1->0, 0->z, z->1, 1->z, z->0.

specparam t_01 = 9, t_10 = 13, t_0z = 11;
specparam t_z1 = 9, t_1z = 11, t_z0 = 13;
(clk => q) = (t_01, t_10, t_0z, t_z1, t_1z, t_z0);
```

Transition	Delay Value
0->x	min(t_01, t_0z) = 9
1->x	min(t_10, t_1z) = 11
z->x	min(t_z0, t_z1) = 9
x->0	max(t_10, t_z0) = 13
x->1	max(t_01, t_z1) = 9
x->z	max(t_1z, t_0z) = 11

`specify` blocks (cont'd)

- Timing Checks
  - A number of system tasks defined for this
  - `$setup`: checks setup-time of a signal before an event
  - `$hold`: checks hold-time of a signal after an event
  - `$width`: checks width of pulses

`specify` blocks (cont'd)
Timing Checks

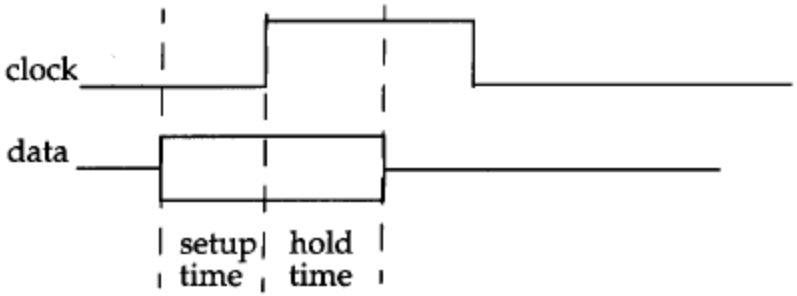


- `$setup` check
  - Syntax:

```
$setup(data_event, reference_event, limit);
```

```
//Setup check is set.
//clock is the reference
//data is being checked for violations
//Violation reported if T_posedge_clk - T_data < 3
specify
   $setup(data, posedge clock, 3);
endspecify
```

`specify` blocks (cont'd)
Timing Checks

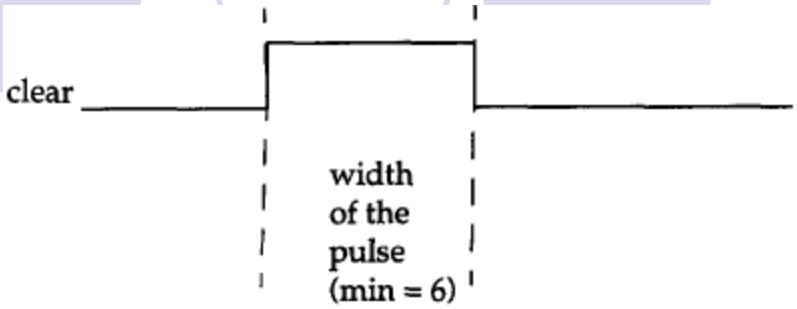


- `$hold` check
  - Syntax:

```
$hold(reference_event, data_event, limit);
```

```
//Hold check is set.
//clock is the reference
//data is being checked for violations
//Violation reported if T_data - T_posedge_clk < 5
specify
   $hold(posedge clear, data, 5);
endspecify
```

`specify` blocks (cont'd) Timing Checks



- `$width` check
  - Syntax:

```
$width(reference_event, limit);
```

```
//width check is set.
//posedge of clear is the reference_event
//the next negedge of clear is the data_event
//Violation reported if T_data - T_clk < 6
specify
    $width(posedge clock, 6);
endspecify
```

Today Summary

- Delays
  - Models
    - Inertial (distributed and lumped delay)
    - Transport (path/pin-to-pin delay)
  - Types
    - Rise/Fall/Turn-off
    - Min/Typ/Max Values
  - Delays in Verilog
    - Syntax and other common features
    - Gate-Level and Dataflow Modeling
    - Behavioral Modeling

Other Notes

- Homework 9
  - Chapter 10:
    - All exercises
    - Due date: Sunday, Day 11th

Digital System Design

Verilog® HDL
Tasks and Functions

Maziar Goudarzi

Today program

- Reusing code
  - Tasks and Functions

Introduction

- Procedures/Subroutines/Functions in SW programming languages
  - The same functionality, in different places
- Verilog equivalence:
  - *Tasks* and *Functions*
  - Used in behavioral modeling
  - Part of design hierarchy $\Rightarrow$ Hierarchical name

Contents

- Functions
- Tasks
- Differences between tasks and functions

Tasks and Functions

Functions

Functions

- Keyword: `function, endfunction`
- Can be used if the procedure
  - does not have any timing control constructs
  - returns exactly a single value
  - has at least one input argument

Functions (cont'd)

- Function Declaration and Invocation
  - Declaration syntax:

```
function <range_or_type> <func_name>;
  <input declaration(s)>
  <variable_declaration(s)>
  begin // if more than one statement needed
    <statements>
  end // if begin used
endfunction
```

Functions (cont'd)

- Function Declaration and Invocation
  - Invocation syntax:

```
<func_name> (<argument(s)>);
```

Functions (cont'd)

- Semantics
  - much like `function` in *Pascal*
  - An internal implicit `reg` is declared inside the function with the same name
  - The return value is specified by setting that implicit `reg`
  - `<range_or_type>` defines width and type of the implicit `reg`
    - `<type>` can be `integer` or `real`
    - default bit width is 1

Function Examples
Parity Generator

```
module parity;
reg [31:0] addr;
reg parity;

initial begin
   …
end

always @(addr)
begin
   parity = calc_parity(addr);
   $display("Parity calculated = %b",
        calc_parity(addr) );
end
```

```
function calc_parity;
input [31:0] address;
begin
   calc_parity = ^address;
end
endfunction

endmodule
```

Function Examples Controllable Shifter

```
module shifter;
`define LEFT_SHIFT      1'b0
`define RIGHT_SHIFT     1'b1
reg [31:0] addr, left_addr,
   right_addr;
reg control;

initial
begin
   …
end

always @(addr)
begin
  left_addr  =shift(addr, `LEFT_SHIFT);
  right_addr =shift(addr,`RIGHT_SHIFT);
end
```

```
function [31:0] shift;
input [31:0] address;
input control;
begin
  shift = (control==`LEFT_SHIFT)
   ?(address<<1) : (address>>1);
end
endfunction

endmodule
```

```
//Define a factorial with a recursive
function
module top;
...
// Define the function
function automatic integer factorial;
input [31:0] oper;
integer i;
begin
if (operand >= 2)
factorial = factorial (oper -1) * oper;
//recursive call
else
factorial = 1 ;
end
endfunction
```

```
// Call the function
integer result;
initial
begin
result = factorial(4); // Call
the factorial of 7
$display("Factorial of 4 is
%0d", result); //Displays 24
end
...
...
endmodule
```

Tasks and Functions

Tasks

Tasks

- Keywords: `task, endtask`
- Must be used if the procedure has
  - any timing control constructs
  - zero or more than one output arguments
  - no input arguments

Tasks (cont'd)

- Task declaration and invocation
  - Declaration syntax

```
task <task_name>;
    <I/O declarations>
    <variable and event declarations>
    begin // if more than one statement needed
      <statement(s)>
    end   // if begin used!
endtask
```

Tasks (cont'd)

- Task declaration and invocation
  - Task invocation syntax

```
<task_name>;
<task_name> (<arguments>);
```

  - `input` and `inout` arguments are passed into the task
  - `output` and `inout` arguments are passed back to the invoking statement when task is completed

Tasks (cont'd)

- I/O declaration in modules vs. tasks
  - Both used keywords: `input, output, inout`
  - In modules, represent ports
    - connect to external signals
  - In tasks, represent arguments
    - pass values to and from the task

Task Examples
Use of input and output arguments

```
module operation;
parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;

initial
   $monitor( …);

initial
begin
   …
end

always @(A or B)
begin
   bitwise_oper(AB_AND, AB_OR,
        AB_XOR, A, B);
end
```

```
task bitwise_oper;
output [15:0]  ab_and, ab_or,
               ab_xor;
input [15:0] a, b;
begin
        #delay ab_and = a & b;
        ab_or = a | b;
        ab_xor = a ^ b;
end
endtask


endmodule
```

Task Examples
Use of module local variables

```
// clk2 runs at twice the frequency of clk
    and is synchronous
// with clk.
module top;
reg [15:0] cd_xor, ef_xor; //variables in
    module top
reg [15:0] c, d, e, f; //variables in
    module top
-
task automatic bitwise_xor;
output [15:0] ab_xor; //output from the
    task
input [15:0] a, b; //inputs to the task
begin
#delay ab_and = a & b;
ab_or = a | b;
ab_xor = a ^ b;
end
endtask
```

```
// These two always blocks will
   call the bitwise_xor task
// concurrently at each positive
   edge of clk. However, since
// the task is re-entrant, these
   concurrent calls will work
   correctly.
always @(posedge clk)
bitwise_xor(ef_xor, e, f);
-
always @(posedge clk2) // twice the
   frequency as the previous block
bitwise_xor(cd_xor, c, d);
-
-
endmodule
```

Tasks and Functions

Differences between Tasks and Functions

Differences between...

- Functions
  - Can enable (call) just another function (not task)
  - Execute in 0 simulation time
  - No timing control statements allowed
  - At lease one input
  - Return only a single value
- Tasks
  - Can enable other tasks and functions
  - May execute in non-zero simulation time
  - May contain any timing control statements
  - May have arbitrary `input`, `output`, or `inout`
  - Do not return any value

Differences between… (cont'd)

- Both
  - are defined in a `module`
  - are local to the `module`
  - can have local variables (registers, but not nets) and events
  - contain only behavioral statements
  - do not contain `initial` or `always` statements
  - are called from `initial` or `always` statements or other tasks or functions

Differences between… (cont'd)

- Tasks can be used for common Verilog code
- Function are used when the common code
  - is purely combinational
  - executes in 0 simulation time
  - provides exactly one output
- Functions are typically used for conversions and commonly used calculations