Anika Haque, 20283

San Francisco Bay University
EE461 Digital Design and HDL

Week#2 Verilog Language Concepts

1. Lab Outlines:

1. Lexical Convention
2. Operator
3. Exercises

2. Lab Procedures

I. Lexical Convention

- // single line comments; /**/ multi line comments
- Case sensitive
- Identifier: (a-z, A-Z, _) + (a-z, A-Z, 0-9, _ , $)

a. Number:

- Integer 4 bytes: 6'b10_0011; 8'hAA; 1 , 8'bx
- Real Number: 12; 0.6; 3.5E6
- signed & unsigned #: 32'hDEAD_BEEF ; -14'h1234

b. Modules

- Example code

```
module      ModuleName(
                        input wire [7:0] a_i,
                        input wire b_i
                        output reg c_o;
            );

      … …
endmodule
```

– Module connection

```
module OneBitFA1TB;
            reg     a_r;
            reg     b_r;
            reg     ci_r;
            wire    sum_w;
            wire    co_w;

            oneBitFA1       uOneBitFA1(

                                        .a_i(a_r),
                                        .b_i(b_r),
                                        .ci_i(ci_r),
                                        .sum_o(sum_w),
                                        .co_o(co_w)
                                );

            initial begin
                    … …
            end
endmodule
```

- Hierarchical Identifiers

```
//-----------------------------------------------------
`include "addbit.v"
module adder_hier (
                        result_o,   // Output of the
                                    // adder
                        carry_o,    // Carry output of
                                    // adder
                        r1_i,       // First input
                        r2_i,       // Second input
                        ci_i        // Carry input
                    );

      // Input Port Declarations
      input    [3:0]   r1_i         ;
      input    [3:0]   r2_i         ;
      input            ci_i         ;

      // Output Port Declarations
      output   [3:0]  result_o      ;
      output          carry_o       ;

      // Port Wires
      wire     [3:0]    r1_i        ;
      wire     [3:0]    r2_i        ;
      wire              ci_i        ;
      wire     [3:0]    result_o    ;
      wire              carry_o     ;

      // Internal variables
      wire              c1_w        ;
      wire              c2_w        ;
      wire              c3_w        ;

      // Code Starts Here
      addbit u0 (r1_i[0],r2_i[0],ci_i,result_o[0],c1_w);
      addbit u1 (r1_i[1],r2_i[1],c1_w,result_o[1],c2_w);
      addbit u2 (r1_i[2],r2_i[2],c2_w,result_o[2],c3_w);
   addbit u3 (r1_i[3],r2_i[3],c3_w,result_o[3],carry_o);

endmodule // End Of Module adder

module tb();

      reg [3:0] r1_r,r2_r;
      reg  ci_r;
      wire [3:0] result_w;
      wire  carry_w;

      // Drive the inputs
      initial begin
            r1_r = 0;
            r2_r = 0;
            ci_r = 0;
            #10 r1_r = 10;
            #10 r2_r = 2;
            #10 ci_r = 1;
            #10 $display("+-------------------------+");
            $finish;
      end

      // Connect the lower module
      adder_hier U (result_w,carry_w,r1_r,r2_r,ci_r);

      // Hier demo here
```

```
      initial begin
            $display("+-----------------------------+");
$display("|r1|r2|ci|u0.sum|u1.sum| u2.sum | u3.sum |");
            $display("+-----------------------------+");
            $monitor("|%h|%h|%h|%h|%h|%h|%h|",r1,r2,ci,
tb.U.u0.sum, tb.U.u1.sum, tb.U.u2.sum, tb.U.u3.sum);
      end

endmodule
```

- Data types:

```
      1. reg: lhs of assignment in always/initial block
      2. wire:  lhs of assignment in "assign block"
```

- String:

\n; \t; \\, \", %%(percent character), \ddd (A character specified in 1-3 octal digits (0 <= d <= 7))

```
module strings();
// Declare a register variable that is 21 bytes
      reg [8*21:0] string ;

      initial begin
            string = "This is sample string";
            $display ("%s \n", string);
      end
endmodule
```

II Operator

Arithmetic Operators: **+, - *, /, %**
rational operator: *a>b, a<b, a>=b, a<=b*
equality operator: *a=b, a!=b, a===b, a!==b*
logic operator: *!, &&, ||*
bitwise operator: *~, (& or ~&), (| or ~|), ^, (~^/^~)*
shift operator: *<<, >>*
concatenation operator *{}*
replication operator *{n{m}}*
conditional operator *cond_expr ? true_expr : false_expr*

- Example Code:

```
module ReductionOperators();
      initial begin
      // Bit Wise AND reduction
         $display (" &  4'b1001 = %b", (&  4'b1001));
         $display (" &  4'bx111 = %b", (&  4'bx111));
         $display (" &  4'bz111 = %b", (&  4'bz111));
         // Bit Wise NAND reduction
         $display (" ~& 4'b1001 = %b", (~& 4'b1001));
         $display (" ~& 4'bx001 = %b", (~& 4'bx001));
         $display (" ~& 4'bz001 = %b", (~& 4'bz001));
         // Bit Wise OR reduction
         $display (" |  4'b1001 = %b", (|  4'b1001));
         $display (" |  4'bx000 = %b", (|  4'bx000));
         $display (" |  4'bz000 = %b", (|  4'bz000));

          #10 $finish;
      end
```

```
endmodule


module ShiftOperators();

   initial begin
      // Left Shift
      $display (" 4'b1001 << 1 = %b", (4'b1001 << 1));
      $display (" 4'b10x1 << 1 = %b", (4'b10x1 << 1));
      $display (" 4'b10z1 << 1 = %b", (4'b10z1 << 1));
      // Right Shift
      $display (" 4'b1001 >> 1 = %b", (4'b1001 >> 1));
      $display (" 4'b10x1 >> 1 = %b", (4'b10x1 >> 1));
      $display (" 4'b10z1 >> 1 = %b", (4'b10z1 >> 1));
      #10 $finish;
    end
endmodule

module ConcatenationOperator();
      initial begin
      // concatenation
            $display (" {4'b1001,4'b10x1}  = %b",{4'b1001,4'b10x1});
             #10 $finish;
      end
endmodule

module ReplicationOperator();

   initial begin
      // replication
      $display (" {4{4'b1001}} = %b", {4{4'b1001}});
      // replication and concatenation
      $display (" {4{4'b1001,1'bz}} = %b", {4{4'b1001,1'bz}});
      #10 $finish;
      end
endmodule
```

III Exercises

- Run module *ReductionOperators, ShiftOperators, ConcatenationOperator* and *ReplicationOperator*

```
module ReductionOperators();
   initial begin
      $display("& 4'b1001 = %b", (&4'b1001)); // Bitwise AND reduction
      $display("~& 4'b1001 = %b", (~&4'b1001)); // Bitwise NAND reduction
      #10 $finish;
   end
endmodule
```

```
module ReductionOperators();
    initial begin
        $display("& 4'b1001 = %b", (&4'b1001)); // Bitwise AND reduction
        $display("~& 4'b1001 = %b", (~&4'b1001)); // Bitwise NAND reduction
        #10 $finish;
    end
endmodule
```

Log Share

```
[2024-12-16 15:27:21 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
& 4'b1001 = 0
~& 4'b1001 = 1
design.sv:5: $finish called at 10 (1s)
Done
```

Anika Haque, 20283

```
module ShiftOperators();
   initial begin
      $display("4'b1001 << 1 = %b", (4'b1001 << 1)); // Left Shift
      $display("4'b1001 >> 1 = %b", (4'b1001 >> 1)); // Right Shift
      #10 $finish;
   end
endmodule
```

```
module ShiftOperators();
    initial begin
        $display("4'b1001 << 1 = %b", (4'b1001 << 1)); // Left Shift
        $display("4'b1001 >> 1 = %b", (4'b1001 >> 1)); // Right Shift
        #10 $finish;
    end
endmodule
```

Log | Share

```
[2024-12-16 15:31:27 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
4'b1001 << 1 = 0010
4'b1001 >> 1 = 0100
design.sv:5: $finish called at 10 (1s)
Done
```

```
module ConcatenationOperator();
   initial begin
      $display("{4'b1001, 4'b0101} = %b", {4'b1001, 4'b0101}); // Concatenation
      #10 $finish;
   end
endmodule

module ReplicationOperator();
   initial begin
      $display("{4{4'b1001}} = %b", {4{4'b1001}}); // Replication
      #10 $finish;
   end
endmodule
```

```
module ConcatenationOperator();
    initial begin
        $display("{4'b1001, 4'b0101} = %b", {4'b1001, 4'b0101}); // Concatenation
        #10 $finish;
    end
endmodule

module ReplicationOperator();
    initial begin
        $display("{4{4'b1001}} = %b", {4{4'b1001}}); // Replication
        #10 $finish;
    end
endmodule
```

Log | Share

```
[2024-12-16 15:33:18 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
{4'b1001, 4'b0101} = 10010101
{4{4'b1001}} = 1001100110011001
design.sv:4: $finish called at 10 (1s)
design.sv:11: $finish called at 10 (1s)
Done
```

Anika Haque, 20283

- Complete the module *addbit* first, and then run module *adder_hier* and its testbench.

MODULES

```
module addbit (
   input a, b, cin,
   output sum, cout
);
   assign sum = a ^ b ^ cin; // XOR for sum
   assign cout = (a & b) | (b & cin) | (a & cin); // Carry logic
endmodule

module adder_hier(
   input [3:0] r1_i, r2_i,  // 4-bit inputs to be added
   input ci_i,              // Carry-in
   output [3:0] result_o,   // 4-bit sum output
   output carry_o           // Final carry-out
);

   // Internal wires for carry propagation
   wire c1_w, c2_w, c3_w;

   // Instantiate the 1-bit adders
   addbit u0 (.a(r1_i[0]), .b(r2_i[0]), .cin(ci_i), .sum(result_o[0]), .cout(c1_w));
   addbit u1 (.a(r1_i[1]), .b(r2_i[1]), .cin(c1_w), .sum(result_o[1]), .cout(c2_w));
   addbit u2 (.a(r1_i[2]), .b(r2_i[2]), .cin(c2_w), .sum(result_o[2]), .cout(c3_w));
   addbit u3 (.a(r1_i[3]), .b(r2_i[3]), .cin(c3_w), .sum(result_o[3]), .cout(carry_o));

endmodule
```

TESTBENCH

```
module tb_adder_hier();

   reg [3:0] r1_r, r2_r;    // 4-bit inputs for the adder
   reg ci_r;                // Carry-in
   wire [3:0] result_w;     // Sum output
   wire carry_w;            // Carry-out

   // Instantiate the adder_hier module
   adder_hier UUT (
      .r1_i(r1_r),
      .r2_i(r2_r),
      .ci_i(ci_r),
      .result_o(result_w),
      .carry_o(carry_w)
   );

   // Stimulus block
   initial begin
      // Initialize inputs
      r1_r = 4'b0000; r2_r = 4'b0000; ci_r = 1'b0;
      #10 r1_r = 4'b1010; r2_r = 4'b0010; ci_r = 1'b1; // Add 10 + 2 + 1
      #10 r1_r = 4'b1111; r2_r = 4'b1111; ci_r = 1'b0; // Add 15 + 15
```

Anika Haque, 20283

```
    #10 r1_r = 4'b0101; r2_r = 4'b1010; ci_r = 1'b0; // Add 5 + 10
    #10 $finish;                                    // End simulation
  end

  // Monitor results
  initial begin
    $monitor("Time: %0d | r1: %b | r2: %b | cin: %b | sum: %b | cout: %b",
             $time, r1_r, r2_r, ci_r, result_w, carry_w);
  end

endmodule
```

testbench.sv

```
module tb_adder_hier();

    reg [3:0] r1_r, r2_r;     // 4-bit inputs for the adder
    reg ci_r;                 // Carry-in
    wire [3:0] result_w;      // Sum output
    wire carry_w;             // Carry-out

    // Instantiate the adder_hier module
    adder_hier UUT (
        .r1_i(r1_r),
        .r2_i(r2_r),
        .ci_i(ci_r),
        .result_o(result_w),
        .carry_o(carry_w)
    );

    // Stimulus block
    initial begin
        // Initialize inputs
        r1_r = 4'b0000; r2_r = 4'b0000; ci_r = 1'b0;
        #10 r1_r = 4'b1010; r2_r = 4'b0010; ci_r = 1'b1; // Add 10 + 2 + 1
        #10 r1_r = 4'b1111; r2_r = 4'b1111; ci_r = 1'b0; // Add 15 + 15
        #10 r1_r = 4'b0101; r2_r = 4'b1010; ci_r = 1'b0; // Add 5 + 10
        #10 $finish;                                     // End simulation
    end

    // Monitor results
    initial begin
        $monitor("Time: %0d | r1: %b | r2: %b | cin: %b | sum: %b | cout: %b",
                 $time, r1_r, r2_r, ci_r, result_w, carry_w);
    end

endmodule
```

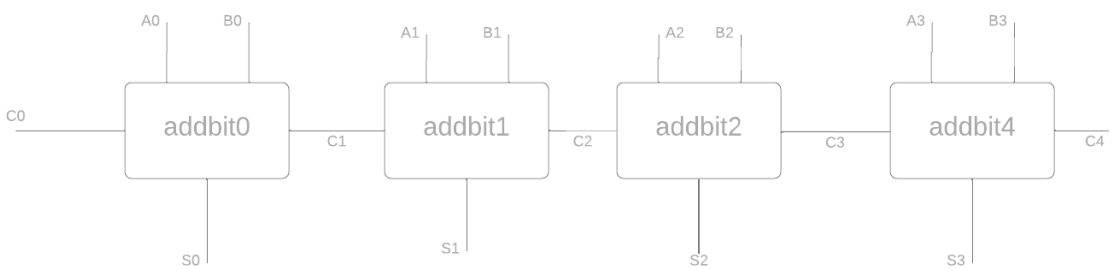
design.sv

```
module addbit (
    input a, b, cin,
    output sum, cout
);
    assign sum = a ^ b ^ cin; // XOR for sum
    assign cout = (a & b) | (b & cin) | (a & cin); // Carry logic
endmodule

module adder_hier(
    input [3:0] r1_i, r2_i,  // 4-bit inputs to be added
    input ci_i,              // Carry-in
    output [3:0] result_o,   // 4-bit sum output
    output carry_o           // Final carry-out
);

    // Internal wires for carry propagation
    wire c1_w, c2_w, c3_w;

    // Instantiate the 1-bit adders
    addbit u0 (.a(r1_i[0]), .b(r2_i[0]), .cin(ci_i), .sum(result_o[0]), .cout(c1_w));
    addbit u1 (.a(r1_i[1]), .b(r2_i[1]), .cin(c1_w), .sum(result_o[1]), .cout(c2_w));
    addbit u2 (.a(r1_i[2]), .b(r2_i[2]), .cin(c2_w), .sum(result_o[2]), .cout(c3_w));
    addbit u3 (.a(r1_i[3]), .b(r2_i[3]), .cin(c3_w), .sum(result_o[3]), .cout(carry_o));

endmodule
```

Log | Share

```
[2024-12-16 15:41:46 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
Time: 0 | r1: 0000 | r2: 0000 | cin: 0 | sum: 0000 | cout: 0
Time: 10 | r1: 1010 | r2: 0010 | cin: 1 | sum: 1101 | cout: 0
Time: 20 | r1: 1111 | r2: 1111 | cin: 0 | sum: 1110 | cout: 1
Time: 30 | r1: 0101 | r2: 1010 | cin: 0 | sum: 1111 | cout: 0
testbench.sv:24: $finish called at 40 (1s)
Done
```

- Based on the module *adder_hier,* draw the circuit functional block diagram



- Complete the testbenches for the modules `DFFSynch`, and `DFFAsynch` in Lab#1

MODULE

```
module DFFSynch(
    input clk, reset, d,     // Inputs: clock, reset, and data
    output reg q             // Output: flip-flop state
);
    always @(posedge clk) begin
        if (reset)           // On reset, clear q to 0
            q <= 0;
        else                 // Otherwise, store d into q
            q <= d;
    end
```

Anika Haque, 20283

```
endmodule

TESTBENCH
module tb_DFFSynch();
    reg clk, reset, d;       // Testbench inputs
    wire q;                  // Testbench output

    // Instantiate the DFFSynch module
    DFFSynch dut (.clk(clk), .reset(reset), .d(d), .q(q));

    // Clock generation: toggles every 5 time units
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    // Stimulus: Apply input combinations to test the flip-flop
    initial begin
        reset = 1; d = 0;     // Reset active
        #10 reset = 0; d = 1; // Release reset, set d to 1
        #10 d = 0;            // Change d
        #10 reset = 1;        // Activate reset again
        #10 reset = 0; d = 1; // Release reset, set d to 1
        #10 $finish;          // End simulation
    end

    // Monitor values to view results in the terminal
    initial begin
        $monitor("Time: %0d | clk: %b | reset: %b | d: %b | q: %b", $time, clk,
reset, d, q);
    end
endmodule
```

testbench.sv

```
module tb_DFFSynch();
    reg clk, reset, d;       // Testbench inputs
    wire q;                  // Testbench output

    // Instantiate the DFFSynch module
    DFFSynch dut (.clk(clk), .reset(reset), .d(d), .q(q));

    // Clock generation: toggles every 5 time units
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    // Stimulus: Apply input combinations to test the flip-flop
    initial begin
        reset = 1; d = 0;     // Reset active
        #10 reset = 0; d = 1; // Release reset, set d to 1
        #10 d = 0;            // Change d
        #10 reset = 1;        // Activate reset again
        #10 reset = 0; d = 1; // Release reset, set d to 1
        #10 $finish;          // End simulation
    end

    // Monitor values to view results in the terminal
    initial begin
        $monitor("Time: %0d | clk: %b | reset: %b | d: %b | q: %b", $time,
clk, reset, d, q);
    end
endmodule
```

design.sv

```
module DFFSynch(
    input clk, reset, d,     // Inputs: clock, reset, and data
    output reg q             // Output: flip-flop state
);
    always @(posedge clk) begin
        if (reset)           // On reset, clear q to 0
            q <= 0;
        else                 // Otherwise, store d into q
            q <= d;
    end
endmodule
```

Log Share

```
[2024-12-16 15:51:13 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
Time: 0 | clk: 0 | reset: 1 | d: 0 | q: x
Time: 5 | clk: 1 | reset: 1 | d: 0 | q: 0
Time: 10 | clk: 0 | reset: 0 | d: 1 | q: 0
Time: 15 | clk: 1 | reset: 0 | d: 1 | q: 1
Time: 20 | clk: 0 | reset: 0 | d: 0 | q: 1
Time: 25 | clk: 1 | reset: 0 | d: 0 | q: 0
Time: 30 | clk: 0 | reset: 1 | d: 0 | q: 0
Time: 35 | clk: 1 | reset: 1 | d: 0 | q: 0
Time: 40 | clk: 0 | reset: 0 | d: 1 | q: 0
Time: 45 | clk: 1 | reset: 0 | d: 1 | q: 1
testbench.sv:21: $finish called at 50 (1s)
Time: 50 | clk: 0 | reset: 0 | d: 1 | q: 1
Done
```

Anika Haque, 20283

MODULE

```
module DFFAsynch(
   input clk, reset, d,    // Inputs: clock, reset, and data
   output reg q            // Output: flip-flop state
);
   always @(posedge clk or posedge reset) begin
      if (reset)          // Asynchronous reset, active immediately
         q <= 0;
      else                // On clock edge, store d into q
         q <= d;
   end
endmodule
```

TESTBENCH

```
module tb_DFFAsynch();
   reg clk, reset, d;      // Testbench inputs
   wire q;                 // Testbench output

   // Instantiate the DFFAsynch module
   DFFAsynch dut (.clk(clk), .reset(reset), .d(d), .q(q));

   // Clock generation: toggles every 5 time units
   initial begin
      clk = 0;
      forever #5 clk = ~clk;
   end

   // Stimulus: Apply input combinations to test the flip-flop
   initial begin
      reset = 1; d = 0;    // Reset active, q should go to 0 immediately
      #3 reset = 0; d = 1; // Release reset (mid-clock), set d to 1
      #7 d = 0;            // Change d
      #10 reset = 1;       // Activate reset again (mid-clock)
      #5 reset = 0; d = 1; // Release reset, set d to 1
      #10 $finish;         // End simulation
   end

   // Monitor values to view results in the terminal
   initial begin
      $monitor("Time: %0d | clk: %b | reset: %b | d: %b | q: %b", $time, clk, reset, d, q);
   end
endmodule
```

Anika Haque, 20283

testbench.sv

```
    wire q;                 // Testbench output

    // Instantiate the DFFAsynch module
    DFFAsynch dut (.clk(clk), .reset(reset), .d(d), .q(q));

    // Clock generation: toggles every 5 time units
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    // Stimulus: Apply input combinations to test the flip-flop
    initial begin
        reset = 1; d = 0;     // Reset active, q should go to 0 immediately
        #3 reset = 0; d = 1; // Release reset (mid-clock), set d to 1
        #7 d = 0;            // Change d
        #10 reset = 1;       // Activate reset again (mid-clock)
        #5 reset = 0; d = 1; // Release reset, set d to 1
        #10 $finish;         // End simulation
    end

    // Monitor values to view results in the terminal
    initial begin
        $monitor("Time: %0d | clk: %b | reset: %b | d: %b | q: %b", $time,
clk, reset, d, q);
    end
endmodule
```

design.sv

```
module DFFAsynch(
    input clk, reset, d,     // Inputs: clock, reset, and data
    output reg q             // Output: flip-flop state
);
    always @(posedge clk or posedge reset) begin
        if (reset)           // Asynchronous reset, active immediately
            q <= 0;
        else                 // On clock edge, store d into q
            q <= d;
    end
endmodule
```

Log Share

```
[2024-12-16 15:54:32 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
Time: 0 | clk: 0 | reset: 1 | d: 0 | q: 0
Time: 3 | clk: 0 | reset: 0 | d: 1 | q: 0
Time: 5 | clk: 1 | reset: 0 | d: 1 | q: 1
Time: 10 | clk: 0 | reset: 0 | d: 0 | q: 1
Time: 15 | clk: 1 | reset: 0 | d: 0 | q: 0
Time: 20 | clk: 0 | reset: 1 | d: 0 | q: 0
Time: 25 | clk: 1 | reset: 0 | d: 1 | q: 1
Time: 30 | clk: 0 | reset: 0 | d: 1 | q: 1
testbench.sv:21: $finish called at 35 (1s)
Time: 35 | clk: 1 | reset: 0 | d: 1 | q: 1
Done
```