Chapter 4 :: Hardware Description Languages

Digital Design and Computer Architecture

David Money Harris and Sarah L. Harris

Chapter 4 :: Topics

- **Introduction**
- **Combinational Logic**
- **Structural Modeling**
- **Sequential Logic**
- **More Combinational Logic**
- **Finite State Machines**
- **Parameterized Modules**
- **Testbenches**

Introduction

- Hardware description language (HDL): allows designer to specify logic function only. Then a computer-aided design (CAD) tool produces the optimized gates.
- Most commercial designs built using HDLs
- Two leading HDLs:
  - Verilog
    - developed in 1984 by Gateway Design Automation
    - became an IEEE standard (1364) in 1995
  - VHDL
    - Developed in 1981 by the Department of Defense
    - Became an IEEE standard (1076) in 1987

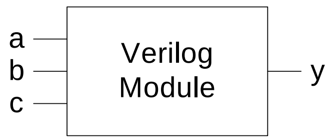
HDL to Gates

- Simulation
  - Input values are applied to the circuit
  - Outputs checked for correctness
  - Millions of dollars saved by debugging in simulation instead of hardware
- Synthesis
  - Transforms HDL code into a *netlist* describing the hardware (i.e., a list of gates and the wires connecting them)

IMPORTANT:

When describing circuits using an HDL, it's critical to think of the **hardware** the code should produce.

Verilog Modules



Two types of Modules:

- Behavioral: describe what a module does
- Structural: describe how a module is built from simpler modules

Behavioral Verilog Example

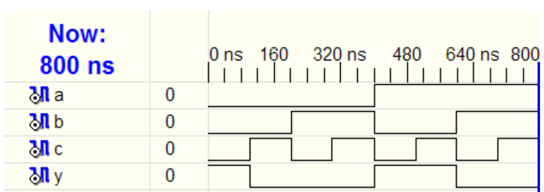
Verilog:

```
module example(input  a, b, c,
               output y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```


Behavioral Verilog Simulation

Verilog:

```
module example(input  a, b, c,
               output y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

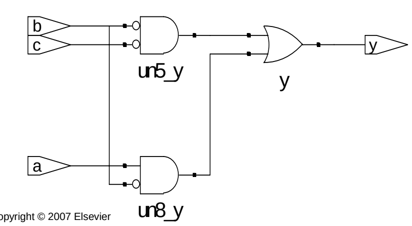


Behavioral Verilog Synthesis

Verilog:

```
module example(input  a, b, c,
               output y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

Synthesis:



Verilog Syntax

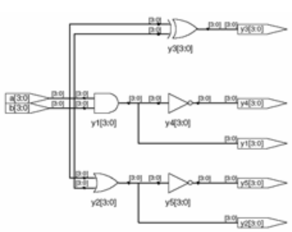
- Verilog is case sensitive. So, `reset` and `Reset` are not the same signal.
- Verilog does not allow you to start signal or module names with numbers. So, for example, `2mux` is an invalid name.
- Verilog ignores whitespace.
- Comments come in single-line and multi-line varieties:
  - // single line comment
  - /* multiline
    comment */

Structural Modeling - Hierarchy

```
module and3(input  a, b, c,
            output y);
  assign y = a & b & c;
endmodule


module inv(input  a,
           output y);
  assign y = ~a;
endmodule


module nand3(input  a, b, c
             output y);
  wire n1;                    // internal signal

  and3 andgate(a, b, c, n1);  // instance of and3
  inv  inverter(n1, y);      // instance of inverter
endmodule
```


Bitwise Operators

```
module gates(input  [3:0]  a, b,
             output [3:0] y1, y2, y3, y4, y5);
   /* Five different two-input logic
      gates acting on 4 bit busses */
   assign y1 = a & b;    // AND
   assign y2 = a | b;    // OR
   assign y3 = a ^ b;    // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```

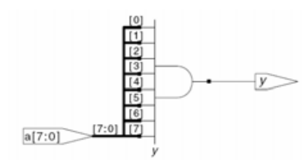


`//` single line comment

`/*…*/` multiline comment

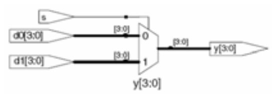
Reduction Operators

```
module and8(input  [7:0] a,
            output       y);
   assign y = &a;
   // &a is much easier to write than
   // assign y = a[7] & a[6] & a[5] & a[4] &
   //            a[3] & a[2] & a[1] & a[0];
endmodule
```



Conditional Assignment

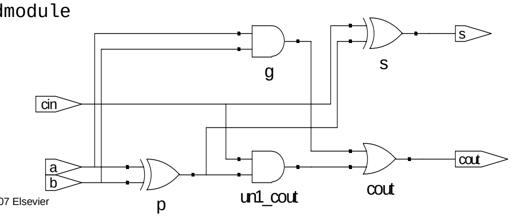
```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);
   assign y = s ? d1 : d0;
endmodule
```



? : is also called a *ternary operator* because it operates on 3 inputs: s, d1, and d0.

Internal Variables

```
module fulladder(input  a, b, cin, output s, cout);
  wire p, g;        // internal nodes

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```



Precedence

Defines the order of operations

Highest

~	NOT
*, /, %	mult, div, mod
+, -	add,sub
<<, >>	shift
<<<, >>>	arithmetic shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, XOR
?:	ternary operator

Lowest

Numbers

Format: N'Bvalue

N = number of bits, B = base

N'B is optional but recommended (default is decimal)

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00…0011
8'b11	8	binary	3	00000011
8'b1010_1011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010
8'hAB	8	hexadecimal	171	10101011
42	Unsized	decimal	42	00…0101010

Bit Manipulations: Example 1

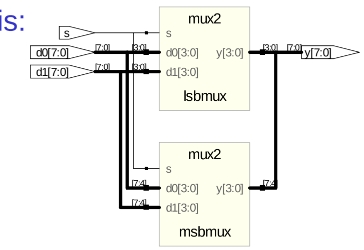
```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};

// if y is a 12-bit signal, the above statement produces:
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0

// underscores (_) are used for formatting only to make
   it easier to read. Verilog ignores them.
```


Bit Manipulations: Example 2

Verilog:

```
module mux2_8(input  [7:0] d0, d1,
              input        s,
              output [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]); // from slide 12
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]); // from slide 12
endmodule
```

Synthesis:

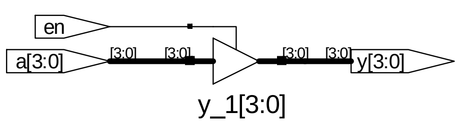


Z: Floating Output

Verilog:

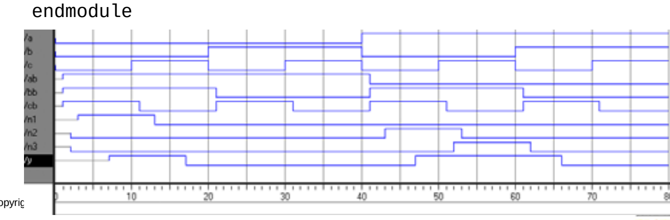
```
module tristate(input  [3:0] a,
                input        en,
                output [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```

Synthesis:



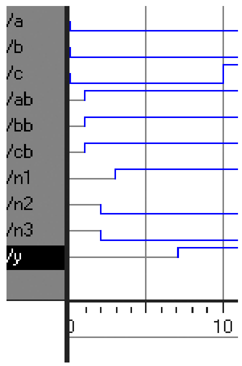
Delays

```
module example(input  a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



Delays

```
module example(input  a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



Sequential Logic

- Verilog uses certain idioms to describe latches, flip-flops and FSMs
- Other coding styles may simulate correctly but produce incorrect hardware

Always Statement

General Structure:

```
always @ (sensitivity list)
  statement;
```

Whenever the event in the `sensitivity list` occurs, the `statement` is executed

D Flip-Flop

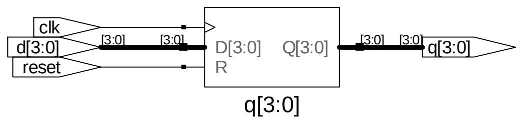
```
module flop(input             clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;              // pronounced “q gets d”

endmodule
```

Any signal on the left side of an `always` statement must be declared `reg`. In this case `q` is declared as `reg`

Beware: A variable declared `reg` is not necessarily a registered output. We will show examples of this later.

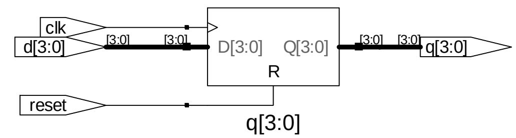
Resettable D Flip-Flop

```
module flopr(input              clk,
             input              reset,
             input       [3:0]  d,
             output reg  [3:0]  q);

  // synchronous reset
  always @ (posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```



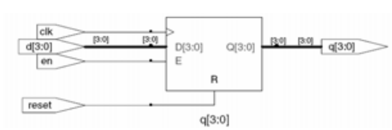
Resettable D Flip-Flop

```
module flopr(input              clk,
             input              reset,
             input       [3:0]  d,
             output reg  [3:0]  q);

  // asynchronous reset
  always @ (posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```



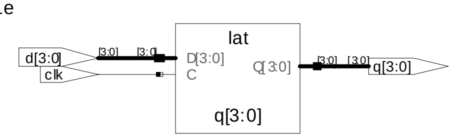
D Flip-Flop with Enable

```
module flopren(input              clk,
               input              reset,
               input              en,
               input       [3:0]  d,
               output reg  [3:0]  q);

  // asynchronous reset and enable
  always @ (posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;

endmodule
```



Latch

```
module latch(input              clk,
             input       [3:0]  d,
             output reg  [3:0]  q);

  always @ (clk, d)
    if (clk) q <= d;

endmodule
```



Warning: We won't use latches in this course, but you might write code that inadvertently implies a latch. So if your synthesized hardware has latches in it, this indicates an error.

Other Behavioral Statements

- Statements that must be inside `always` statements:
  - `if` / `else`
  - `case`, `casez`
- Reminder: Variables assigned in an `always` statement must be declared as `reg` (even if they're not actually registered!)

Combinational Logic using always

```
// combinational logic using an always statement
module gates(input      [3:0] a, b,
             output reg [3:0] y1, y2, y3, y4, y5);
  always @ (*)       // need begin/end because there is
    begin            // more than one statement in always
      y1 = a & b;    // AND
      y2 = a | b;    // OR
      y3 = a ^ b;    // XOR
      y4 = ~(a & b); // NAND
      y5 = ~(a | b); // NOR
    end
endmodule
```

This hardware could be described with assign statements using fewer lines of code, so it's better to use assign statements in this case.

Combinational Logic using case

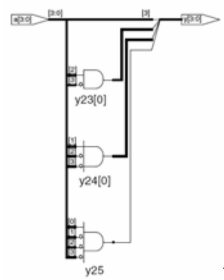
```
module sevenseg(input      [3:0] data,
                output reg [6:0] segments);
  always @(*)
    case (data)
      //              abc_defg
      0: segments = 7'b111_1110;
      1: segments = 7'b011_0000;
      2: segments = 7'b110_1101;
      3: segments = 7'b111_1001;
      4: segments = 7'b011_0011;
      5: segments = 7'b101_1011;
      6: segments = 7'b101_1111;
      7: segments = 7'b111_0000;
      8: segments = 7'b111_1111;
      9: segments = 7'b111_1011;
      default: segments = 7'b000_0000; // required
    endcase
endmodule
```


Combinational Logic using case

- In order for a `case` statement to imply combinational logic, all possible input combinations must be described by the HDL.
- Remember to use a `default` statement when necessary.

Combinational Logic using **casez**

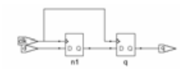
```
module priority_casez(input      [3:0] a,
                      output reg [3:0] y);

  always @(*)
    casez(a)
      4'b1???: y = 4'b1000;  // ? = don't care
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
    endcase

endmodule
```



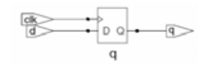
Blocking vs. Nonblocking Assignments

- <= is a "nonblocking assignment"
  - Occurs simultaneously with others
- = is a "blocking assignment"
  - Occurs in the order it appears in the file

```
// Good synchronizer using
// nonblocking assignments
module syncgood(input      clk,
                input      d,
                output reg q);
  reg n1;
  always @(posedge clk)
    begin
      n1 <= d;  // nonblocking
      q  <= n1; // nonblocking
    end
endmodule
```



```
// Bad synchronizer using
// blocking assignments
module syncbad(input      clk,
               input      d,
               output reg q);
  reg n1;
  always @(posedge clk)
    begin
      n1 = d;  // blocking
      q  = n1; // blocking
    end
endmodule
```



Rules for Signal Assignment

- Use `always @ (posedge clk)` and nonblocking assignments to model synchronous sequential logic

```
always @ (posedge clk)
  q <= d; // nonblocking
```

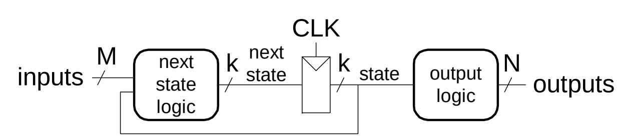
- Use continuous assignments to model simple combinational logic.

```
assign y = a & b;
```

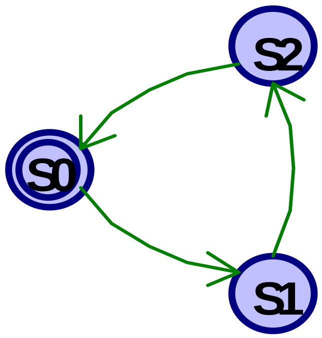
- Use `always @ (*)` and blocking assignments to model more complicated combinational logic where the `always` statement is helpful.
- Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.

Finite State Machines (FSMs)

- Three blocks:
  - next state logic
  - state register
  - output logic



FSM Example: Divide by 3



The double circle indicates the reset state

FSM in Verilog

```
module divideby3FSM (input  clk,
                     input  reset,
                     output y);
   reg  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

// state register
   always @ (posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;
 // next state logic
   always @ (*)
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase
   // output logic
   assign q = (state == S0);
endmodule
```


Parameterized Modules

2:1 mux:

```
module mux2
  #(parameter width = 8)  // name and default value
   (input  [width-1:0] d0, d1,
    input              s,
    output [width-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

Instance with 8-bit bus width (uses default):

```
  mux2 mux1(d0, d1, s, out);
```

Instance with 12-bit bus width:

```
   mux2 #(12) lowmux(d0, d1, s, out);
```


Testbenches

- HDL code written to test another HDL module, the *device under test* (dut), also called the *unit under test* (uut)
- Not synthesizeable
- Types of testbenches:
  - Simple testbench
  - Self-checking testbench
  - Self-checking testbench with testvectors

Example

Write Verilog code to implement the following function in hardware:

$$y = \bar{b}\bar{c} + a\bar{b}$$

Name the module `sillyfunction`

Example

Write Verilog code to implement the following function in hardware:

$$y = \bar{b}\bar{c} + a\bar{b}$$

Name the module `sillyfunction`

Verilog

```
module sillyfunction(input  a, b, c,
                     output y);
  assign y = ~b & ~c | a & ~b;
endmodule
```


Simple Testbench

```
module testbench1();
  reg  a, b, c;
  wire y;
  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // apply inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
  end
endmodule
```


Self-checking Testbench

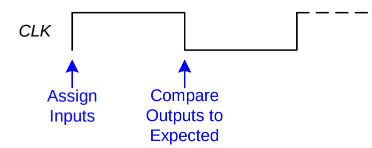
```
module testbench2();
  reg  a, b, c;
  wire y;
  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // apply inputs one at a time
  // checking results
  initial begin
    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1; #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("010 failed.");
    c = 1; #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1; #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("110 failed.");
    c = 1; #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

Testbench with Testvectors

- Write testvectors file
- Testbench:
  1. Generate clock for assigning inputs, reading outputs
  2. Read testvectors file into array
  3. Assign inputs, expected outputs
  4. Compare outputs to expected outputs and report errors

Testbench with Testvectors

- Testbench clock is used to assign inputs (on the rising edge) and compare outputs with expected outputs (on the falling edge).



- The testbench clock may also be used as the clock source for synchronous sequential circuits.

Testvectors File

File: `example.tv` – contains vectors of abc_yexpected

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```


Testbench: 1. Generate Clock

```
module testbench3();
  reg         clk, reset;
  reg         a, b, c, yexpected;
  wire        y;
  reg  [31:0] vectornum, errors;    // bookkeeping variables
  reg  [3:0]  testvectors[10000:0]; // array of testvectors

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always     // no sensitivity list, so it always executes
    begin
      clk = 1; #5; clk = 0; #5;
    end
```

2. Read Testvectors into Array

```
// at start of test, load vectors
// and pulse reset

initial
   begin
     $readmemb("example.tv", testvectors);
     vectornum = 0; errors = 0;
     reset = 1; #27; reset = 0;
   end


// Note: $readmemh reads testvector files written in
// hexadecimal
```


3. Assign Inputs and Expected Outputs

```
// apply test vectors on rising edge of clk
 always @(posedge clk)
   begin
     #1; {a, b, c, yexpected} = testvectors[vectornum];
   end
```


4. Compare Outputs with Expected Outputs

```
// check results on falling edge of clk
   always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin
        $display("Error: inputs = %b", {a, b, c});
        $display("  outputs = %b (%b expected)",y,yexpected);
        errors = errors + 1;
      end

// Note: to print in hexadecimal, use %h. For example,
//       $display("Error: inputs = %h", {a, b, c});
```


4. Compare Outputs with Expected Outputs

```
// increment array index and read next testvector
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
          $display("%d tests completed with %d errors",
                vectornum, errors);
        $finish;
      end
    end
endmodule


// Note: === and !== can compare values that are
// x or z.
```