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 or synthesizes 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

SystemVerilog

• These lecture notes cover SystemVerilog
  – An improved version of Verilog
  – Standardized in 2005
  – Widely used in industry
  – Backward compatible
    • Verilog examples from the text will work
  – Eliminates weird syntax artifacts of Verilog
    • Use logic type in place of wire and reg
    • Better ways to differentiate between combinational and sequential logic
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 logic a, b, c, output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
endmodule

Behavioral Verilog Simulation

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

Now: 800 ns
<table>
<thead>
<tr>
<th></th>
<th>0 ns</th>
<th>160 ns</th>
<th>320 ns</th>
<th>480 ns</th>
<th>640 ns</th>
<th>800 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Behavioral Verilog Synthesis

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

Synthesis:

Verilog Syntax

- Case sensitive
  - Example: reset and Reset are not the same signal.
- No names that start with numbers
  - Example: 2mux is an invalid name.
- Whitespace ignored
- Comments:
  - // single line comment
  - /* multiline comment */
- logic represents a Boolean signal in SystemVerilog
  - 0, 1, x, z
  - Essentially replaces wire and reg data types from Verilog
  - Exception: use tri or trireg on nets with multiple drivers

Structural Modeling - Hierarchy

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

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

module nand3(input logic a, b, c
              output logic n1;
    logic 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 logic [3:0] a, b,
             output logic [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
/*...*/
Reduction Operators

module and8(input logic [7:0] a, output logic y);
    assign y = &a;
    // &a is much easier to write than
endmodule

Conditional Assignment

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

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

Internal Variables

module fulladder(input logic a, b, cin, output logic s, cout);
    logic p, g;    // internal nodes
    assign p = a ^ b;
    assign g = a & b;
    assign s = p ^ cin;
    assign cout = g | (p & cin);
endmodule

Precedence

\begin{tabular}{|c|c|}
\hline
\textbf{Highest} & \textbf{Lowest} \\
\hline
\texttt{~, !} & not \texttt{~, %} \\
\texttt{*}, \texttt{/}, \texttt{\%} & mult, div, mod \texttt{+, -, \%} \\
\texttt{<}, \texttt{>, >>} & add, sub \texttt{<<, >>} \\
\texttt{<, <, >, >>} & shift \texttt{<, <, >, >>} \\
\texttt{==, !=} & arithmetic shift \texttt{==, !=} \\
\texttt{&}, \texttt{~&} & equal, not equal \texttt{&}, \texttt{~&} \\
\texttt{^}, \texttt{~^} & AND, NAND \texttt{^}, \texttt{~^} \\
\texttt{|}, \texttt{~|} & XOR, XNOR \texttt{|}, \texttt{~|} \\
\texttt{?} : & ternary operator \\
\hline
\end{tabular}
Numbers

Format: N'B<value

- N = number of bits, B = base
- N'B is optional but recommended (default is decimal)

<table>
<thead>
<tr>
<th>Number</th>
<th># Bits</th>
<th>Base</th>
<th>Decimal Equivalent</th>
<th>Stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>3'b101</td>
<td>3</td>
<td>binary</td>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>'b11</td>
<td>unsized</td>
<td>binary</td>
<td>3</td>
<td>00...0011</td>
</tr>
<tr>
<td>8'b111</td>
<td>8</td>
<td>binary</td>
<td>3</td>
<td>00000001</td>
</tr>
<tr>
<td>8'b111_1011</td>
<td>8 binary</td>
<td>171</td>
<td>10101011</td>
<td></td>
</tr>
<tr>
<td>3'd6</td>
<td>3</td>
<td>decimal</td>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>6'o42</td>
<td>6</td>
<td>octal</td>
<td>34</td>
<td>100010</td>
</tr>
<tr>
<td>8'hAB</td>
<td>8</td>
<td>hexadecimal</td>
<td>171</td>
<td>10101011</td>
</tr>
<tr>
<td>42</td>
<td>Unsized</td>
<td>decimal</td>
<td>42</td>
<td>00...0101010</td>
</tr>
</tbody>
</table>

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 logic [7:0] d0, d1,
               input logic s,
               output logic [7:0] y);

mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule

Synthesis:

Z: Floating Output

Verilog:

module tristate(input logic [3:0] a,
                input logic en,
                output tristate [3:0] y);

assign y = en ? a : 4'bz;
endmodule

Synthesis:
Tristate Busses

- logic signals must have exactly one driver
- Tristate busses may have multiple drivers
  - Use tri or trireg
- Typically one driver is active at a time
  - If multiple drivers give conflicting values, bus gets x
  - If no driver is active:
    - tri floats (z)
    - trireg keeps its prior value

Delays

module example(input logic a, b, c,
    output logic y);
    logic 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.

**Specialized Forms:**
always_ff
always_latch
always_comb

Check for common mistakes in the always statement.

D Flip-Flop

```verilog
module flop(input logic clk,
            input logic [3:0] d,
            output logic [3:0] q);
    always_ff @(posedge clk)
        q <= d;  // pronounced "q gets d"
endmodule
```

Also could have written
```verilog
always @(posedge clk)
    q <= d;
```

but always_ff checks for typos that don’t imply a flip-flop.

Resettable D Flip-Flop

```verilog
module flopr(input logic clk,
             input logic reset, input logic [3:0] d,
             output logic [3:0] q);
    // synchronous reset
    always_ff @(posedge clk)
        if (reset) q <= 4'b0;
        else      q <= d;
endmodule
```

Resettable D Flip-Flop

```verilog
module flopr(input logic clk,
             input logic reset, input logic [3:0] d,
             output logic [3:0] q);
    // asynchronous reset
    always_ff @(posedge clk, posedge reset)
        if (reset) q <= 4'b0;
        else      q <= d;
endmodule
```
D Flip-Flop with Enable

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

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

Latch

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

   always_latch
     if (clk) q <= d;
endmodule

Other Behavioral Statements

- Statements that must be inside always statements:
  - if/else
  - case, casez

Combinational Logic using always

   // combinational logic using an always statement
module gates(input logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);

   always_comb // 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

   Also could have written
   always @(a or b) ... 
   always @(*) ...

   but always_comb checks for typos that don’t imply comb logic.
   This hardware could be described with assign statements using fewer lines of code, so it is
   better to use assign statements in this scenario.
Combinational Logic using **case**

module sevenseg(input logic [3:0] data, output logic [6:0] segments);
always_comb
  case (data)
    //
    0: segments = 7'b111_1110;
    1: segments = 7'b011_0000;
    2: segments = 7'b110_1101;
    3: segments = 7'b011_0001;
    4: segments = 7'b111_0011;
    5: segments = 7'b101_1111;
    6: segments = 7'b101_1111;
    7: segments = 7'b111_1000;
    8: segments = 7'b111_1110;
    9: segments = 7'b111_1111;
    default: segments = 7'b000_0000; // required
  endcase
endmodule

Combinational Logic using **casez**

module priority_casez(input logic [3:0] a, output logic [3:0] y);
always_comb
  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

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

// Bad synchronizer using blocking assignments
module syncbad(input logic clk, input logic d, output logic q);
logic n1;
always_ff @(posedge clk)
  begin
    n1 = d; // blocking
  end
endmodule
```
**Rules for Signal Assignment**

- Use `always_ff @(posedge clk)` and nonblocking assignments (`<=`) to model synchronous sequential logic.
  ```verilog
  always_ff @(posedge clk)
  q <= d; // nonblocking
  ```
- Use continuous assignments (`assign ...`) to model simple combinational logic.
  ```verilog
  assign y = a & b;
  ```
- Use `always_comb` 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.

**Life Before SystemVerilog**

- In the original Verilog standard, `logic` didn’t exist
- `wire` and `reg` were used inside
  - `wire` used for signals driven in assign statements
  - `reg` used for signals driven in always blocks
    - Whether or not they are registers (!)
- SystemVerilog avoids this confusing issue
  - But old code with `wire` and `reg` still works

**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

```verilog
module divideby3FSM (input logic clk,
                      input logic reset, output logic q);
logic [1:0] state, nextstate;
parameter S0 = 2'b00;
parameter S1 = 2'b01;
parameter S2 = 2'b10;
// state register
always_ff @(posedge clk, posedge reset)
if (reset) state <= S0;
else       state <= nextstate;
// next state logicalways_comb
case (state)
    S0:      nextstate = S1;
    S1:      nextstate = S2;
    S2:      nextstate = S0;
    default: nextstate = S0;
endcase
// output logic
assign q = (state == S0);
endmodule
```

Parameterized Modules

```verilog
module 2:1 mux#
        #(parameter width = 8)  // name and default value
        (input  logic [width-1:0] d0, d1,
         input  logic           s,
         output logic [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 (utt)
- Not synthesizable
- Types of testbenches:
  - Simple testbench
  - Self-checking testbench
  - Self-checking testbench with testvectors

Example

Write Verilog code to implement the following function in hardware:

\[ y = \overline{b}c + a\overline{b} \]

Name the module sillyfunction

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

module testbench1();
logic a, b, c;
logic 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; 
       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();
logic a, b, c;logic y;
   // instantiate device under test
   sillyfunction dut(a, b, c, y);
   // 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 testvector file: inputs and expected outputs
- 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**

```verilog
module testbench3();
logic        clk, reset;
logic        a, b, c, yexpected;
logic        y;
logic [31:0] vectornum, errors;    // bookkeeping variables
logic [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
```

**Testbench: 2. Read Testvectors into Array**

```verilog
// 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
```

---

**Testbench: 3. Assign Inputs and Expected Outputs**

```verilog
// 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
  end

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

// 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
endmodule

// Note: === and !== can compare values that are
// l, 0, x, or z.