Chapter 4 :: Hardware Description Languages

Digital Design and Computer Architecture

David Money Harris and Sarah L. Harris
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
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

• 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 */
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
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.
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

<table>
<thead>
<tr>
<th>Highest</th>
<th>Lowest</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>?:</td>
</tr>
<tr>
<td>*, /, %</td>
<td>=, ==, !=, &amp;</td>
</tr>
<tr>
<td>+, -</td>
<td>^, ~^</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;&lt;, &gt;&gt;&gt;</td>
<td>&gt;&gt;</td>
</tr>
<tr>
<td>arithmetic shift</td>
<td>comparison</td>
</tr>
<tr>
<td>comparison</td>
<td>equal, not equal</td>
</tr>
<tr>
<td>AND, NAND</td>
<td>XOR, XNOR</td>
</tr>
<tr>
<td>OR, XOR</td>
<td></td>
</tr>
</tbody>
</table>

~ NOT

*, /, % mult, div, mod

+,- add, sub

<<, >> shift

<<<, >>> arithmetic shift

<, <=, >, >= comparison

==, != equal, not equal

&, ~& AND, NAND

^, ~^ XOR, XNOR

|, ~| OR, XOR

?: ternary operator
# Numbers

Format: N'Bvalue

- 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'b1010_1011</td>
<td>8</td>
<td>binary</td>
<td>171</td>
<td>10101011</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>
assign y = {a[2:1], {3{b[0]}}}, a[0], 6'b100_010;

// if y is a 12-bit signal, the above statement produces:
// underscores (_) are used for formatting only to make it easier to read. Verilog ignores them.
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0
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]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
Verilog:

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

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

Synthesis:
Tri-State Buffers

• ‘Z’ value is the tri-stated value

• This example implements tri-state drivers driving BusOut

module tstate (EnA, EnB, BusA, BusB, BusOut);
    input EnA, EnB;
    input [7:0] BusA, BusB;
    output [7:0] BusOut;

    assign BusOut = EnA ? BusA : 8’bZ;
    assign BusOut = EnB ? BusB : 8’bZ;
endmodule
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
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

Delay annotation is *ignored* by synthesis!
• Only useful for simulation/modeling
• But may cause simulation to work when synthesis doesn’t
  – Beware!!
Inertial and Transport Delays

- **Inertial Delay**
  - \#3 \( X = A \);
  - Wait 3 time units, then assign value of \( A \) to \( X \)
  - The usual way delay is used in simulation
    - models logic delay reasonably

- **Transport Delay**
  - \( X <= \#3 A \);
    - Current value of \( A \) is assigned to \( X \), after 3 time units
  - Better model for transmission lines and high-speed logic
Parameterized Modules

2:1 mux:

```verilog
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):

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

Instance with 12-bit bus width:

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

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);

Naming parameters – order doesn’t matter:

mux2 mux1(.s(s), .y(out), .d0(d0), .d1(d1));

Instance with 12-bit bus width:

mux2 #(width=12) lowmux
  (.s(s), .y(out), .d0(d0), .d1(d1));
Always Statement

General Structure:

\[
\text{always @ (sensitivity list)}
\text{statement;}
\]

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

This is dangerous

For combinational logic use the following!

\[
\text{always @ (*)}
\text{statement;}
\]
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 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`

```verilog
module sevenseg(input [3:0] data,
                output reg [6:0] segments);
    always @(*)
        case (data)
            // abc_defg
            0: segments = 7'b111_1110;  // required
            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;
        endcase
endmodule
```
In order for an always block statement to implement combinational logic, all possible input combinations must be described by the HDL.

Remember to use a default statement when necessary in case statements.

This is why assign statements are always preferable to combinational always blocks (when possible)
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
for loops

Remember – always block is executed at compile time!

module what  
    input [8:0] data, 
    output reg [3:0] count 
); 
integer    i; 
always @(*) begin 
    count = 0; 
    for (i=0; i<9; i=i+1) begin 
        count = count + data[i]; 
    end 
end 
endmodule
while loops

module what(
in [15:0] in,
output reg [4:0] out);
integer i;
always @(*) begin: count
  out = 0;
i = 15;
  while (i >= 0 && ~in[i]) begin
    out = out + 1;
i = i - 1;
  end
end
endmodule
disable – exit a named block

module what(
    input [15:0] in,
    output reg [4:0] out
);

integer i;
always @(*) begin: count
    out = 0;
    for (i = 15; i >= 0; i = i - 1) begin
        if (~in[i]) disable count;
        out = out + 1;
    end
end
endmodule