Overview of Verilog

Some properties:
- Syntax is similar to C (terse)
- Modeling at various levels: switch level, gate level, RTL, behavioral level
- More of a chip level design language than system level (like VHDL)

<table>
<thead>
<tr>
<th>Verilog</th>
<th>VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestFixture</td>
<td>TestBench</td>
</tr>
<tr>
<td>case sensitive</td>
<td>case insensitive</td>
</tr>
<tr>
<td>always</td>
<td>process</td>
</tr>
<tr>
<td>module</td>
<td>entity and architecture combined</td>
</tr>
</tbody>
</table>

General format:
```
module module_name (module_terminal_list_separated_by_commas);
    port and net declarations (IO, wires and regs for internal nodes)
    IO types: input, output, inout
    wire = combinational logic, reg = sequential logic (retains value)
    functional description
endmodule
```

Comments are like C and C++:
```
// this is a one line comment to the end of line
/* this is a multiple line comment */
```

2-to-1 MUX example:
```
module MUX2 (A,B,S,Z);
    input A,B,S;
    output Z;
    always
        begin
            if (S == 0) Z = A;
            else Z = B;
        end
endmodule
```

Hierarchy example of 4-to-1 MUX (calling the 2-to-1 MUX above):
```
module MUX4 (A,B,C,D,S0,S1,Z);
    input A,B,C,D,S0,S1;
    output Z;
    wire Z1,Z2;
    MUX2 M1(A,B,S0,Z1);
    MUX2 M2(C,D,S0,Z2);
    MUX2 M3(Z1,Z2,S1,Z);
endmodule
```

*note no component declaration, just instantiation – must include MUX2 module in Verilog source before MUX4 module*

Vector notation supported (an example of four 2-to-1 MUXs):
module MUX2ARR(A,B,S,Z);
input [3:0]   A,B; // note whitespace after array declaration
input S;
output [3:0]   Z;
always
    begin
      if (S == 0) Z = A;
      else Z = B;
    end
endmodule

Like VHDL, whitespaces include space, tab, and newline

Numbers:

<table>
<thead>
<tr>
<th>Verilog:</th>
<th>VHDL:</th>
<th>note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>4'b1010</td>
<td>“1010” or B“1010”</td>
<td>a 4-bit binary value</td>
</tr>
<tr>
<td>12’ha5c</td>
<td>X“0a5c”</td>
<td>a 12-bit hexadecimal value</td>
</tr>
<tr>
<td>6’o71</td>
<td>O“71”</td>
<td>a 6-bit octal value</td>
</tr>
<tr>
<td>8’d255</td>
<td>255</td>
<td>an 8-bit decimal value</td>
</tr>
</tbody>
</table>

Logic values: 0, 1, x, z (note lowercase x and z, for undefined logic and tri-state values, respectively)

Identifiers: and sequence of letters (a-z, A-Z), digits (0-9), $ (dollar sign) and (underscore). The first character must be a letter or and underscore.

Clock edges:
    @ (posedge CLK) or @ (negedge CLK) for rising or falling clock edges

Example of a simple rising edge triggered flip-flop:
    always @ (posedge CLK)
        begin
        Q = D;
        end

Example of a falling edge triggered flip-flop with sync preset and clock enable:
    always @ (negedge CLK)
        begin
        if (PR == 1) Q = 1;
        else if (CE = 1) Q = D;
        end
Operators (in increasing order of precedence):

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>==</td>
<td>logical equality</td>
</tr>
<tr>
<td>!=</td>
<td>logical inequality</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
</tr>
</tbody>
</table>

Unary operators:

- !: logical negation
- ~: bitwise negation
- &: reduction AND
- &&: reduction NAND
- |: reduction OR
- ^: reduction XOR
- ^: reduction XNOR

Verilog has 6 gate types that can be called hierarchically: and, or, nand, nor, xor, xnor

gate GATE_INST_NAME (Z,I1,I2,…IN); // the output comes first followed by inputs

if-else constructs are like C except that instead of open bracket `{` and close bracket `}`, use keywords begin and end, respectively, for multiple assignments associated with a given condition (begin and end are not needed for single assignments)

module MUX4 (A,B,C,D,S0,S1,Z);
input A,B,C,D,S0,S1;
output Z1,Z2;
wire Z1,Z2;
always begin
    if ((S1 == 1'b0) && (S0 == 1'b0)) Z = A;
    else if ((S1 == 1'b0) && (S0 == 1'b1)) Z = B;
    else if ((S1 == 1'b1) && (S0 == 1'b0)) Z = C;
    else Z = D;
end
endmodule
two process modeling style can be used as illustrated for the following FSM example

```verilog
module FSM (CLK,X,Z);
input CLK,X;
output Z;
reg [1:0] CS;
parameter SA = 2'b00  // define state A with binary value 00
parameter SB = 2'b01  // define state B with binary value 01
parameter SC = 2'b10  // define state C with binary value 10

always @(posedge CLK)
  begin
    if (CS == SA)
      begin
        if (X == 0) CS = SC;
        else CS = SB;
      end
    else if (CS == SB)
      begin
        if (X == 0) CS = SA;
        else CS = SC;
      end
    else
      begin
        if (X == 0) CS = SB;
        else CS = SA;
      end
  end

always @(CS)
  begin
    case (CS)
      SA: begin
        Z = 1'b0;
      end
      SB: begin
        Z = 1'b1;
      end
      SC: begin
        Z = 1'b1;
      end
    endcase
  end
endmodule
```