CSE140L: Components and Design
Techniques for Digital Systems Lab

Verilog HDL

Tajana Simunic Rosing

Source: Eric Crabill, Xilinx
Overview

• Lab #1 due

• What we’ve covered previously:
  – Transistor level design – delay analysis
  – Delay – when is it good to have delay?

• What we’ll be covering next:
  – Verilog HDL
Hardware description languages

- Used to describe & model the operation of digital circuits.

- Specify simulation procedure for the circuit and check its response.
  - Simulation requires a logic simulator.

- Synthesis: transformation of the HDL description into a physical implementation (transistors, gates)
  - When a human does this, it is called logic design.
  - When a machine does this, it is called synthesis.
HDLs

- Abel (circa 1983) - developed by Data-I/O
  - targeted to programmable logic devices
  - not good for much more than state machines
- ISP (circa 1977) - research project at CMU
  - simulation, but no synthesis
- Verilog (circa 1985) - developed by Gateway (absorbed by Cadence)
  - similar to C
  - delays are the only interaction with the simulator
  - fairly efficient and easy to write
  - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
  - VHSIC Hardware Description Language
    (VHSIC is Very High Speed Integrated Circuit).
  - simulation semantics visible; very general but verbose
  - IEEE standard
Verilog Usage

- Verilog may be used to model circuits and behaviors at various levels of abstraction:
  - Transistor. LOW
  - Gate.
  - Logic.
  - Behavioral.
  - Algorithmic. HIGH

- Transistor and gate level modeling is not appropriate for design with FPGA devices.
Verilog

• Supports structural and behavioral descriptions

  • Structural
    – explicit structure of the circuit
    – e.g., each logic gate instantiated and connected to others

  • Behavioral
    – program describes input/output behavior of circuit
    – many structural implementations could have same behavior
    – e.g., different implementation of one Boolean function
module xor_gate (out, a, b);
    input a, b;
    output out;
    wire abar, bbar, t1, t2;

    inverter invA (abar, a);
    inverter invB (bbar, b);
    and_gate and1 (t1, a, bbar);
    and_gate and2 (t2, b, abar);
    or_gate or1 (out, t1, t2);
endmodule
Behavioral model

module xor_gate (out_or, out_and, a, b);
    input         a, b;
    output        out_or, out_and;
    reg           out_or, out_and;

    always @(a or b) begin
        out_or = a ^ b;
    end

    assign out_and = a & b;
endmodule
Data Values

- For our logic design purposes, we’ll consider Verilog to have four different bit values:
  - 0, logic zero.
  - 1, logic one.
  - z, high impedance.
  - x, unknown.
Data Types and Values

• There are two main data types in Verilog.
  – Wires.
  – Regs.
• These data types may be single bit or multi-bit.
  – The general syntax is: `{bit width}'{base}{value}
    • 4’d14 // 4-bit value, specified in decimal
    • 4’he // 4-bit value, specified in hex
    • 4’b1110 // 4-bit value, specified in binary
    • 4’b10xz // 4-bit value, with x and z, in binary
Data Types

• Wires:
  – “continuously assigned” values and do not store information.
  – may have multiple drivers assigning values.
  – When multiple drivers exist, the simulator resolves them into a single value for the wire.
  – Every time a driver changes its output value, the resulting wire value is re-evaluated.

• This behaves much like an electrical wire...
Data Types

• Regs
  – “procedurally assigned” values that store information until the next value assignment is made.
  – can be used to model combinational or sequential logic.
  – The name “reg” does not mean it is a register!
  – A reg may be assigned by multiple processes.
  – Other reg varieties include integer, real, and time.
• Consider a top level module declaration:

```plaintext
module testbench;
    // Top level modules do not have ports.
endmodule
```

• Consider a module declaration with ports:

```plaintext
module two_input_xor (in1, in2, out);
    input in1, in2;
    output out;
    // We’ll add more later...
endmodule
```
## Modules and Ports

- Ports may be of type \{input, output, inout\} and can also be multiple bits wide.

```verilog
module some_random_design (fred, bob, joe, sam, tom, ky);
    input        fred; // 1-bit input port
    input  [7:0] bob;  // 8-bit input port
    output       joe;  // 1-bit output port
    output [1:0] sam;  // 2-bit output port
    inout        tom;  // 1-bit bidirectional port
    inout [3:0]  ky;   // 4-bit bidirectional port

    // Some design description would be here...
endmodule
```
Port and Data Types

- **Input port:**
  - driven from *outside* the module by a wire or a reg,
  - *inside* the module it can only drive a wire
- **Output port**
  - driven from *inside* the module by a wire or a reg,
  - *outside* the module it can only drive a wire.
- **Inout port**
  - May be driven by a wire, and drive a wire.
module testbench;
wire sig3; // wire driven by submodule
reg sig1, sig2; // regs assigned by testbench

two_input_xor my_xor (.in1(sig1), .in2(sig2), .out(sig3));

endmodule

module two_input_xor (in1, in2, out);
input in1, in2;
output out;
// We’ll add more later...
endmodule
Operators

- Used in both procedural and continuous assignments.
- Listed in the order of evaluation precedence:
  - `{ }` is used for concatenation.
    Say you have two 1-bit data objects, sam and bob.
    `{sam, bob}` is a 2-bit value from concatenation
  - `{{{ }}}` is used for replication.
    Say you have a 1-bit data object, ted.
    `{4{ted}}` is a 4-bit value, ted replicated four times.
  - Unary operators:
    - `!` Performs logical negation (test for non-zero).
    - `~` Performs bit-wise negation (complements).
    - `&` Performs unary reduction of logical AND.
    - `|` Performs unary reduction of logical OR.
    - `^` Performs unary reduction of logical XOR.
Operators cont.

- **Dyadic arithmetic operators (signed and can generate carry out):**
  - * Multiplication.
  - / Division.
  - % Modulus.
  - + Addition.
  - - Subtraction.

- **Dyadic logical shift operators:**
  - << Shift left.
  - >> Shift right.

- **Dyadic relational operators:**
  - > Greater than.
  - < Less than.
  - >= Greater than or equal.
  - <= Less than or equal.
Operators cont.

- **Dyadic comparison operators:**
  
  `==` Equality operator (compares to z, x are invalid).
  
  `!=` Not equal.

- **Dyadic binary bit-wise operators:**
  
  `&` Bit-wise logical AND.
  
  `|` Bit-wise logical OR.
  
  `^` Bit-wise logical XOR.
  
  `~^` Bit-wise logical XNOR.

- **Dyadic binary logical operators:**
  
  `&&` Binary logical AND.
  
  `||` Binary logical OR.

- **Ternary operator for conditional selection:**
  
  `sel ? value_if_sel_is_one : value_if_sel_is_zero`  
  
  `oe ? driven_value : 1’bz`
Continuous Assignment

- Continuous assignments are made with the assign statement:
  - `assign LHS = RHS;`
    - The left hand side, LHS, must be a wire.
    - The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants.
- The value of the RHS is continuously driven onto the wire of the LHS.
- Values x and z are allowed and processed.
- All assign statements operate concurrently.
Continuous Assignment

module two_input_xor (in1, in2, out);
input in1, in2; // use these as a wire
output out; // use this as a wire

assign out = in1 ^ in2;

endmodule

module two_input_xor (in1, in2, out);
input in1, in2;
output out;
wire product1, product2;

assign product1 = in1 & !in2; // could have done all in
assign product2 = !in1 & in2; // assignment of out with
assign out = product1 | product2; // bigger expression

endmodule
Continuous Assignment

module two_input_xor (in1, in2, out);
    input in1, in2;
    output out;
    assign out = (in1 != in2);
endmodule

module two_input_xor (in1, in2, out);
    input in1, in2;
    output out;
    assign out = in1 ? (!in2) : (in2);
endmodule
Procedural Assignment

• Models combinational and sequential logic
• Operates concurrently and is preceded by event control.
• In block statements start with “begin” and end with “end”.
  – Single assignments can omit begin and end.
• A sensitivity list specifies events which cause the execution to begin:
  – always @(a or b) // any changes in a or b
  – always @(posedge a) // a transitions from 0 to 1
  – always @(negedge a) // a transitions from 1 to 0
  – always @(a or b or negedge c or posedge d)
Procedural Assignment

initial
begin
  // These procedural assignments are executed
  // one time at the beginning of the simulation.
end

always @(sensitivity list)
begin
  // These procedural assignments are executed
  // whenever the events in the sensitivity list
  // occur.
end
Procedural Assignment

• Inside a block, two types of assignments exist:
  – LHS = RHS;       // blocking assignment
  – LHS <=RHS;       // non-blocking assignment
  – Do not mix them in a given block.

• Assignment rules:
  – The left hand side, LHS, must be a reg.
  – The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants.
Procedural Assignment

• Do I use blocking or non-blocking assignments?
  – Blocking assignments are especially useful when describing combinational logic.
    • You can “build up” complex logic expressions.
    • Blocking assignments make your description evaluate in the order you described it.
  – Non-blocking assignments are useful when describing sequential logic.
    • At a clock or reset event, all memory elements change state at the same time, best modeled by non-blocking assignments.

• For conditional assignments use if-else and various types of case statements.

• You also can make use of additional timing control:
  – wait, #delay, repeat, while, etc…
• Combinational logic using operators:

```verilog
module two_input_xor (in1, in2, out);
    input  in1, in2; // use these as wires
    output out; // use this as a wire
    reg    out;

    always @(in1 or in2) // Note that all input terms
        begin // are in sensitivity list!
            out = in1 ^ in2; // Or equivalent expression...
        end

    // I could have simply used:
    // always @(in1 or in2) out = in1 ^ in2;

endmodule
```
Procedural Assignment

- Combinational logic using if-else:

```verilog
module two_input_xor (in1, in2, out);
input  in1, in2; // use these as wires
output out;    // use this as a wire
reg    out;

always @(in1 or in2) // Note that all input terms
begin                 // are in sensitivity list!
    if (in1 == in2) out = 1'b0;
    else out = 1'b1;
end

endmodule
```
Procedural Assignment

• Combinational logic using case:

```verbatim
two_input_xor (in1, in2, out);
input in1, in2; // use these as wires
output out; // use this as a wire
reg out;

always @(in1 or in2) // Note that all input terms
begin // are in sensitivity list!
    case {{in2, in1}} // Concatenated 2-bit selector
        2'b01: out = 1'b1;
        2'b10: out = 1'b1;
        default: out = 1'b0;
    endcase
end
endmodule
```
Delay Control

• You can add delay to continuous assignments.
  – assign #delay LHS = RHS;
    • The #delay indicates a time delay in simulation time units; for example, #5 is a delay of five.
    • This can be used to model physical delays of combinational logic.

• The simulation time unit can be changed by the Verilog "`timescale" directive.
Delay Control

- Control the timing of assignments in procedural blocks by:
  - Level triggered timing control.
    - wait (!reset);
    - wait (!reset) $a = b$;
  - Simple delays.
    - #10;
    - #10 $a = b$;
  - Edge triggered timing control.
    - @(a or b);
    - @(a or b) $c = d$;
    - @(posedge clk);
    - @(negedge clk) $a = b$;
Comparator example

module Compare1 (Equal, Alarger, Blarger, A, B);
    input    A, B;
    output   Equal, Alarger, Blarger;

    assign #5 Equal = (A & B) | (~A & ~B);
    assign #3 Alarger = (A & ~B);
    assign #3 Blarger = (~A & B);
endmodule
Delay Control

- Generation of clock and resets in testbench:

```verilog
reg rst, clk;
initial // this happens once at time zero
begin
    rst = 1'b1; // starts off as asserted at time zero
    #100; // wait for 100 time units
    rst = 1'b0; // deassert the rst signal
end
always // this repeats forever
begin
    clk = 1'b1; // starts off as high at time zero
    #25; // wait for half period
    clk = 1'b0; // clock goes low
    #25; // wait for half period
end
```
System Tasks

- The $ sign denotes Verilog system tasks, there are a large number of these, most useful being:
  - $display("The value of a is %b", a);
    - Used in procedural blocks for text output.
    - The %b is the value format (binary, in this case…)
  - $finish;
    - Used to finish the simulation.
    - Use when your stimulus and response testing is done.
  - $stop;
    - Similar to $finish, but doesn’t exit simulation.
Driving a simulation through a “testbench”

module testbench (x, y);
    output x, y;
    reg [1:0] cnt;

    initial begin
        cnt = 0;
        repeat (4) begin
            #10 cnt = cnt + 1;
            $display (@ time=%d, x=%b, y=%b, cnt=%b, $time, x, y, cnt);
            #10 $finish;
        end
    end

    assign x = cnt[1];
    assign y = cnt[0];
endmodule
More complex behavioral model

module life (n0, n1, n2, n3, n4, n5, n6, n7, self, out);
    input n0, n1, n2, n3, n4, n5, n6, n7, self;
    output out;
    reg out;
    reg [7:0] neighbors;
    reg [3:0] count;
    reg [3:0] i;

    assign neighbors = {n7, n6, n5, n4, n3, n2, n1, n0};

    always @(neighbors or self) begin
        count = 0;
        for (i = 0; i < 8; i = i+1) count = count + neighbors[i];
        out = (count == 3);
        out = out | ((self == 1) & (count == 2));
    end
endmodule
HDLs vs. PLs

- **Program structure**
  - instantiation of multiple components of the same type
  - specify interconnections between modules via schematic
  - hierarchy of modules

- **Assignment**
  - continuous assignment (logic always computes)
  - propagation delay (computation takes time)
  - timing of signals is important (when does computation have its effect)

- **Data structures**
  - size explicitly spelled out - no dynamic structures
  - no pointers

- **Parallelism**
  - hardware is naturally parallel (must support multiple threads)
  - assignments can occur in parallel (not just sequentially)