EECS150 - Digital Design
Lecture 4 – Verilog Introduction

September 6, 2011

Elad Alon
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www-inst.eecs.berkeley.edu/~cs150

Announcements

• Homework #1 due this Thurs.

• Homework #2 out this Thurs. evening
  - Due next Thurs.
Outline

- Background and History of Hardware Description Languages
- Brief Introduction to Verilog Basics
- Lots of examples
  - structural, data-flow, behavioral
- Verilog in EECS150

Design Entry

- Schematic entry/editing used to be the standard method in industry and universities.
- Used in EECS150 until 2002
  😊 Schematics are intuitive. They match our use of gate-level or block diagrams.
  😐 Somewhat physical. They imply a physical implementation.
  😞 Require a special tool [editor].
  😞 Poorly drawn schematics can be confusing and difficult to follow

- Hardware Description Languages (HDLs) are the new standard
  - except for PC board & some custom IC designs
Hardware Description Languages

- **Basic Idea:**
  - Language constructs describe circuits with two basic forms:
    - **Structural descriptions:** connections of components. Nearly one-to-one correspondence with schematic diagram.
    - **Behavioral descriptions:** use high-level constructs (similar to conventional programming) to describe the circuit function.

- Originally invented for simulation.
  - Now "logic synthesis" tools exist to automatically convert from HDL source to circuits.
  - High-level constructs greatly improves designer productivity.
  - However, this may lead you to falsely believe that hardware design can be reduced to writing programs!* 

"Structural" example:
```
Decoder(output x0,x1,x2,x3; inputs a,b)
{
    wire abar, bbar;
    inv(bbar, b);
    inv(abar, a);
    and(x0, abar, bbar);
    and(x1, abar, b);
    and(x2, a, bbar);
    and(x3, a, b);
}
```

"Behavioral" example:
```
Decoder(output x0,x1,x2,x3; inputs a,b)
{
    case [a b]
      00: [x0 x1 x2 x3] = 0x1;
      01: [x0 x1 x2 x3] = 0x2;
      10: [x0 x1 x2 x3] = 0x4;
      11: [x0 x1 x2 x3] = 0x8;
    endcase;
}
```

Note: this is a fake HDL

A Warning

- "Code" you write in a hardware description language does **not** result in a program!
  - You are describing hardware – i.e., something that will eventually be implemented in real circuits

- There is some similarity between HDL and parallel programming
  - I.e., everything happens "simultaneously"
  - But don't take the analogy too far
Sample Design Methodology

Hierarchically defines structure and/or function of circuit.

HDL Specification

Simulation

Synthesis

Verification: Does the design behave as required with regards to function, timing, and power consumption?

Maps specification to resources of implementation platform (FPGA or custom silicon).

Note: This is not the entire story. Other tools are useful for analyzing HDL specifications. More on this later.

Verilog

- A brief history:
  - Invented as simulation language. Synthesis was an afterthought. Many of the basic techniques for synthesis were developed at Berkeley in the 80's and applied commercially in the 90's.
  - Around the same time as the origin of Verilog, the US Department of Defense developed VHDL (A double acronym! VSIC [Very High-Speed Integrated Circuit] HDL). Because it was in the public domain it began to grow in popularity.
  - Afraid of losing market share, Cadence opened Verilog to the public in 1990.
  - Verilog is the language of choice of Silicon Valley companies, initially because of high-quality tool support and its similarity to C-language syntax.
  - VHDL is still popular within the government, in Europe and Japan, and some Universities.
  - Most major CAD frameworks now support both.
  - Latest Verilog version is "system Verilog".
  - Latest HDL: C++ based. OSCI [Open System C Initiative].
Verilog Introduction

- A **module** definition describes a component in a circuit
- Two ways to describe module contents:
  - Structural Verilog
    - List of sub-components and how they are connected
    - Just like schematics, but using text
    - Tediuous to write, hard to decode
    - You get precise control over circuit details
    - May be necessary to map to special resources of the FPGA
  - Behavioral Verilog
    - Describe what a component does, not how it does it
    - Synthesized into a circuit that has this behavior
    - Result is only as good as the tools
- Build up a hierarchy of modules. Top-level module is your entire design (or the environment to test your design).

Verilog Modules and Instantiation

- Modules define circuit components.
- Instantiation defines hierarchy of the design.

```
module addr_cell (a, b, cin, s, cout);
input a, b, cin;
output s, cout;
endmodule
```

```
module adder (A, B, S);
addr_cell ac1 (              );
endmodule
```

Instance of addr_cell

Note: A module is not a function in the C sense. There is no call and return mechanism. Think of it more like a hierarchical data structure.
### Structural Model – XOR Example

```verilog
module xor gate(out, a, b);
    input a, b;
    output out;
    wire aBar, bBar, t1, t2;
    not invA (aBar, a);
    not invB (bBar, b);
    and and1 (t1, a, bBar);
    and and2 (t2, b, aBar);
    or or1 (out, t1, t2);
endmodule
```

**Notes:**
- The instantiated gates are not “executed”. They are active always.
- `xor gate` already exists as a built-in (so really no need to define it).
- Undeclared variables assumed to be wires. Don’t let this happen to you!

### Structural Example: 2-to1 mux

```verilog
/* 2-input multiplexor in gates */
module mux2 (in0, in1, select, out);
    input in0, in1, select;
    output out;
    wire s0, w0, w1;
    not (s0, select);
    and (w0, s0, in0),
        (w1, select, in1);
    or (out, w0, w1);
endmodule // mux2
```

**C++ style comments**
- Multiple instances can share the same “master” name.
- Built-ins gates can have > 2 inputs. Ex:
  ```verilog
  and (w0, a, b, c, d);
  ```

**Built-ins don’t need instance names**
- Built-ins don’t need instance names.

![Diagram](image)
**Instantiation, Signal Arrays, Named Ports**

```verilog
module mux4 (in0, in1, in2, in3, select, out);
input in0, in1, in2, in3;
input [1:0] select;
output out;
wire w0, w1;
mux2
  m0 (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
  m1 (.select(select[0]), .in0(in2), .in1(in3), .out(w1)),
  m3 (.select(select[1]), .in0(w0), .in1(w1), .out(out));
endmodule // mux4
```

**Signal array.** Declares `select[1]`, `select[0]`

**Named ports.** Highly recommended.

**Simple Behavioral Model**

```verilog
module foo (out, in1, in2);
input         in1, in2;
output        out;

assign out = in1 & in2;
endmodule
```

“continuous assignment”

Connects out to be the “and” of `in1` and `in2`.

**Shorthand for explicit instantiation of “and” gate (in this case).**

The assignment continuously happens, therefore any change on the rhs is reflected in out immediately (except for the small delay associated with the implementation of the &).

No[t like an assignment in C that takes place when the program counter gets to that place in the program.
## Continuous Assignment Examples

```verilog
assign R = X | (Y & ~Z);
assign r = &X;
assign R = (a == 1'b0) ? X : Y;
assign P = 8'hff;
assign P = X * Y;
assign P[7:0] = {4{X[3]}, X[3:0]};
assign {cout, R} = X + Y + cin;
assign Y = A << 2;
assign Y = {A[1], A[0], 1'b0, 1'b0};
```

The use of bit-wise Boolean operators, conditional operator, example constants, arithmetic operators (use with care!), example reduction operator, operator, and equivalent bit shift are highlighted.

## Verilog Operators

<table>
<thead>
<tr>
<th>Verilog Operator</th>
<th>Name</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>bit-select or part-select</td>
<td></td>
</tr>
<tr>
<td>()</td>
<td>parenthesis</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical negation</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>reduction AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>reduction NAND</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>reduction XNOR</td>
<td></td>
</tr>
<tr>
<td>`.</td>
<td>unary (sign) plus</td>
<td></td>
</tr>
<tr>
<td>`'</td>
<td>unary (sign) minus</td>
<td></td>
</tr>
<tr>
<td>()</td>
<td>concatenation</td>
<td></td>
</tr>
<tr>
<td>[]</td>
<td>replication</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>binary plus</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>binary minus</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comparison Operator</th>
<th>Description</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>equality</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>inequality</td>
<td></td>
</tr>
<tr>
<td>===</td>
<td>case equality</td>
<td></td>
</tr>
<tr>
<td>!==</td>
<td>case inequality</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise AND</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bit-wise XNOR</td>
<td></td>
</tr>
<tr>
<td><code>. or .</code></td>
<td>bit-wise OR</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>conditional</td>
<td></td>
</tr>
</tbody>
</table>

The operators are organized into different functional groups such as Logical, Arithmetic, Relational, etc.
**Verilog Numbers**

- **14** ordinary decimal number
- **–14** 2’s complement representation
- **12’b0000_0100_0110** binary number [“_” is ignored]
- **12’h046** hexadecimal number with 12 bits

**Signal Values:**

By default, Values are unsigned

- if \( A = 0110 \) (6) and \( B = 1010 \) (6)
  \( C = 10000 \) not 00000
- i.e., B is zero-padded, not sign-extended

```
wire signed [31:0] x;
Declarations a signed (2’s complement) signal array.
```

---

**Non-continuous Assignments**

A bit strange from a hardware specification point of view. Shows off Verilog roots as a simulation language.

```
module and_or_gate (out, in1, in2, in3);
  input in1, in2, in3;
  output out;
  reg out;

  always @(in1 or in2 or in3) begin
    out = (in1 & in2) | in3;
  end
endmodule
```

Isn’t this just:

```
assign out = (in1 & in2) | in3;
```

Why bother?
Always Blocks

Always blocks give us some constructs that are impossible or awkward in continuous assignments.

**case statement example:**

```vhdl
module mux4 (in0, in1, in2, in3, select, out);
  input in0, in1, in2, in3;
  input [1:0] select;
  output out;
  reg out;

  always @ (in0 in1 in2 in3 select)
    case (select)
      2'b00: out = in0;
      2'b01: out = in1;
      2'b10: out = in2;
      2'b11: out = in3;
    endcase
endmodule // mux4
```

**Nested if-else example:**

```vhdl
module mux4 (in0, in1, in2, in3, select, out);
  input in0, in1, in2, in3;
  input [1:0] select;
  output out;
  reg out;

  always @ (in0 in1 in2 in3 select)
    if (select == 2'b00) out = in0;
    else if (select == 2'b01) out = in1;
    else if (select == 2'b10) out = in2;
    else out = in3;
endmodule // mux4
```

*Could we just do this with nested “if”s?*

Yes and no…

**Nested if structure leads to “priority logic” structure, with different delays for different inputs (in3 to out delay > than in0 to out delay). Case version treats all inputs the same.**
State Elements

Always blocks are the only way to specify the "behavior" of state elements. Synthesis tools will turn state element behaviors into state element instances.

D-flip-flop with synchronous set and reset example:

```verilog
module dff(q, d, clk, set, rst);
    input d, clk, set, rst;
    output q;
    reg q;

    always @(posedge clk)
        if (rst)
            q <= 1'b0;
        else if (set)
            q <= 1'b1;
        else
            q <= d;
endmodule
```

How would you add a CE (clock enable) input?

Flip-Flop Example (v1)

How would you add a CE (clock enable) input?

```verilog
module dff(q, d, clk, set, rst);
    input d, clk, set, rst;
    output q;
    reg q;

    always @(posedge clk)
        if (rst)
            q <= 1'b0;
        else if (set)
            q <= 1'b1;
        else
            q <= d;
endmodule
```
Flip-Flop Example (v2)

How would you add a CE (clock enable) input?

```verilog
module dff(q, d, clk, set, rst);
    input d, clk, set, rst;
    output q;
    reg q;

    always @(posedge clk)
        if (rst)
            q <= 1'b0;
        else if (set)
            q <= 1'b1;
        else
            q <= d;

endmodule
```

Finite State Machines

State Transition Diagram

Implementation Circuit Diagram

Holds a symbol to keep track of which bubble the FSM is in.

CL functions to determine output value and next state based on input and current state.

out = f(in, current state)
next state = f(in, current state)
module FSM1(clk, rst, in, out);
input clk, rst;
input in;
output out;

// Defined state encoding:
parameter IDLE = 2'b00;
parameter S0 = 2'b01;
parameter S1 = 2'b10;
reg [1:0] state, next_state;

// always block for state register
always @(posedge clk)
    if (rst) state <= IDLE;
    else state <= next_state;

// always block for combinational logic portion
always @(state or in)
    case (state)
        IDLE : begin
            out = 1'b0;
            if (in == 1'b1) next_state = S0;
            else next_state = IDLE;
        end
        S0 : begin
            out = 1'b0;
            if (in == 1'b1) next_state = S1;
            else next_state = S0;
        end
        S1 : begin
            out = 1'b1;
            if (in == 1'b1) next_state = S1;
            else next_state = IDLE;
        [default: begin
            next_state = IDLE;
            data_out = 1'b0;
        end
        endcase
endmodule

FSMs (cont.)

// always block for combinational logic portion
always @(state or in)
    case (state)
        // For each state def output and next
        IDLE : begin
            out = 1'b0;
            if (in == 1'b1) next_state = S0;
            else next_state = IDLE;
        end
        S0 : begin
            out = 1'b0;
            if (in == 1'b1) next_state = S1;
            else next_state = S0;
        end
        S1 : begin
            out = 1'b1;
            if (in == 1'b1) next_state = S1;
            else next_state = IDLE;
        [default: begin
            next_state = IDLE;
            data_out = 1'b0;
        end
        endcase
endmodule
Verilog in EECS150

- We will primarily use **behavioral modeling** along with **instantiation** to 1) build hierarchy and, 2) map to FPGA resources not supported by synthesis.
- Favor continuous assign and avoid always blocks unless:
  - no other alternative: ex: state elements, case
  - helps readability and clarity of code: ex: large nested if else
- Use named ports.
- Verilog is a big language. This is only an introduction.
  - Our text book is a good source. Read and use chapter 4.
  - Be careful of what you read on the web. Many bad examples out there.
  - We will be introducing more useful constructs throughout the semester. Stay tuned!

Final thoughts on Verilog Examples

Verilog looks like C, but it describes hardware
Multiple physical elements with parallel activities and temporal relationships.

A large part of digital design is knowing how to write Verilog that gets you the desired circuit. **First understand the circuit you want, then figure out how to code it in Verilog.** If you do one of these activities without the other, you will struggle. These two activities will merge at some point for you.

Be suspicious of the synthesis tools! Check the output of the tools to make sure you get what you want.