EECS150 - Digital Design
Lecture 3 - Verilog Introduction

Jan 29, 2013
John Wawrzynek

Outline

• Background and History of Hardware Description
• 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).
- Unless hierarchy is carefully designed, schematics can be confusing and difficult to follow on large designs.

Hardware Description Languages (HDLs) are the new standard
- except for PC board design, where schematics are still used.

Hardware Description Languages

• Basic Idea:
  - Language constructs describe circuits with two basic forms:
  - Structural descriptions: connections of components. Nearly one-to-one correspondence to 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, a, b);
    and(x1, a, b);
    and(x2, a, b);
    and(x3, a, b);
}

Warning: this is a fake HDL!

*Describing hardware with a language is similar, however, to writing a parallel program.
Sample Design Methodology

Hierarchically defines structure and/or function of circuit.

HDL Specification

Simulation

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

Synthesis

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
    - Tedious 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
```

Instance of `addr_cell`

```
module adder (A, B, S);
  addr_cell acl1;  // ...connections...
endmodule
```

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
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
```

---

**Structural Example: 2-to1 mux**

- **a)** 2-input mux symbol
- **b)** 2-input mux gate-level circuit diagram

```c
/* 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
```
### Instantiation, Signal Array, 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 &).

**Not like an assignment in C that takes place when the program counter gets to that place in the program.**
Example - Ripple Adder

module FullAdder(a, b, ci, r, co);
    input a, b, ci;
    output r, co;
    assign r = a ^ b ^ ci;
    assign co = a&ci | a&b | b&cin;
endmodule

module Adder(A, B, R);
    input [3:0] A;
    input [3:0] B;
    output [4:0] R;
    wire c1, c2, c3;
    FullAdder
        add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1),   .r(R[0]) ),
        add1(.a(A[1]), .b(B[1]), .ci(c1),   .co(c2),   .r(R[1]) ),
        add2(.a(A[2]), .b(B[2]), .ci(c2),   .co(c3),   .r(R[2]) ),
        add3(.a(A[3]), .b(B[3]), .ci(c3),   .co(R[4]), .r(R[3]) );
endmodule

Continuous Assignment Examples

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};
### 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>Logical</td>
</tr>
<tr>
<td>&amp;</td>
<td>negation</td>
<td>Bit-wise</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reduction AND</td>
</tr>
<tr>
<td>-&amp;</td>
<td>reduction NAND</td>
<td>Reduction</td>
</tr>
<tr>
<td>^</td>
<td>reduction NOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>^- or ^'</td>
<td>reduction XOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>+</td>
<td>unary (sign) plus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>unary (sign) minus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>(())</td>
<td>concatenation</td>
<td>Concatenation</td>
</tr>
<tr>
<td>()</td>
<td>replication</td>
<td>Replication</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>+</td>
<td>binary plus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>binary minus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>Shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>Shift</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>Relational</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td>==</td>
<td>logical equality</td>
<td>Equality</td>
</tr>
<tr>
<td>!=</td>
<td>logical inequality</td>
<td>Equality</td>
</tr>
<tr>
<td>===</td>
<td>case equality</td>
<td>Equality</td>
</tr>
<tr>
<td>!==</td>
<td>case inequality</td>
<td>Equality</td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise AND</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>^</td>
<td>bit-wise XOR</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>^- or ^=</td>
<td>bit-wise XNOR</td>
<td>Bit-wise</td>
</tr>
</tbody>
</table>

### Verilog Numbers

**Constants:**

- **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

```verilog
e.g., C[4:0] = A[3:0] + B[3:0];
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;**

Declares 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.

“always” block example:

```verilog
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.

```verilog
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
```

Couldn’t we just do this with nested “if”s?

Well yes and no!
Always Blocks

Nested if-else example:

```verilog
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
```

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.

Review - Ripple Adder Example

```verilog
module FullAdder(a, b, ci, r, co);
    input a, b, ci;
    output r, co;
    
    assign r = a ^ b ^ ci;
    assign co = a&ci + a&b + b&cin;
endmodule
```

```verilog
module Adder(A, B, R);
    input [3:0] A;
    input [3:0] B;
    output [4:0] R;
    
    wire c1, c2, c3;
    FullAdder
        add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1), .r(R[0]) ),
        add1(.a(A[1]), .b(B[1]), .ci(c1), .co(c2), .r(R[1]) ),
        add2(.a(A[2]), .b(B[2]), .ci(c2), .co(c3), .r(R[2]) ),
        add3(.a(A[3]), .b(B[3]), .ci(c3), .co(R[4]), .r(R[3]) );
endmodule
```
Example - Ripple Adder Generator

Parameters give us a way to generalize our designs. A module becomes a “generator” for different variations. Enables design/module reuse. Can simplify testing.

```verilog
module Adder(A, B, R);

parameter N = 4;
input [N-1:0] A;
input [N-1:0] B;
output [N:0] R;
wire [N:0] C;

genvar i;

generate
for (i=0; i<N; i=i+1) begin:bit
    FullAdder add(.a(A[i], .b(B[i]), .ci(C[i]), .co(C[i+1]), .r(R[i]));
end
endgenerate

assign C[0] = 1'b0;
assign R[N] = C[N];

endmodule
```

Parameters give us a way to generalize our designs. A module becomes a “generator” for different variations. Enables design/module reuse. Can simplify testing.

- Declare a parameter with default value.
- Note: this is not a port. Acts like a “synthesis-time” constant.
- Replace all occurrences of “4” with “N”.
- variable exists only in the specification - not in the final circuit.

```verilog
// Gray-code to binary-code converter
module gray2bin1 (bin, gray);

parameter SIZE = 8;
output [SIZE-1:0] bin;
input [SIZE-1:0] gray;

genvar i;

generate
for (i=0; i<SIZE; i=i+1)
begin:bit
    assign bin[i] = ^gray[SIZE-1:i];
end
endgenerate

assign C[0] = 1'b0;
assign R[N] = C[N];

endmodule
```

Variable exists only in the specification - not in the final circuit.

Keywords that denote synthesis-time operations

For-loop creates instances (with unique names)

More on Generate Loop

Permits variable declarations, modules, user defined primitives, gate primitives, continuous assignments, initial blocks and always blocks to be instantiated multiple times using a for-loop.

```verilog
// Gray-code to binary-code converter
module gray2bin1 (bin, gray);

parameter SIZE = 8;
output [SIZE-1:0] bin;
input [SIZE-1:0] gray;

genvar i;

generate
for (i=0; i<SIZE; i=i+1)
begin:bit
    assign bin[i] = ^gray[SIZE-1:i];
end
endgenerate

assign C[0] = 1'b0;
assign R[N] = C[N];

endmodule
```

generate if-else-if based on an expression that is deterministic at the time the design is synthesized.
generate case: selecting case expression must be deterministic at the time the design is synthesized.
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.