ECE 551
System on Chip Design

Introduction to Verilog

Garrett S. Rose
Fall 2018
Outline

- HDL versus schematic entry
- Basic structure of a Verilog module
- Common syntax of Verilog
- Continuous versus Procedural Assignments
- The Verilog always block
- Sensitivity lists
- Nets versus variables
## Schematic versus HDL

<table>
<thead>
<tr>
<th>Schematic</th>
<th>HDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Good for multiple data flow</td>
<td>✓ Flexible and parameterizable</td>
</tr>
<tr>
<td>✓ Provides overview picture</td>
<td>✓ Excellent for optimization</td>
</tr>
<tr>
<td>✓ Relates directly to hardware</td>
<td>✓ Excellent for top-down synthesis</td>
</tr>
<tr>
<td>✓ No need for programming skills</td>
<td>✓ Direct mapping to algorithms</td>
</tr>
<tr>
<td>✓ High information density</td>
<td>✓ Excellent for datapaths</td>
</tr>
<tr>
<td>✓ Easy back annotations</td>
<td>✓ Easy to handle electronically</td>
</tr>
<tr>
<td>✓ Useful for mixed analog/digital</td>
<td>✓ Serial representation</td>
</tr>
<tr>
<td>✓ Not good for algorithms</td>
<td>✓ May not show overall picture</td>
</tr>
<tr>
<td>✓ Not good for datapaths</td>
<td>✓ Need good programming skills</td>
</tr>
<tr>
<td>✓ Poor interface for optimization</td>
<td>✓ Divorced from physical hardware</td>
</tr>
<tr>
<td>✓ Difficult to reuse</td>
<td></td>
</tr>
<tr>
<td>✓ Difficult to parameterize</td>
<td></td>
</tr>
</tbody>
</table>
Verilog Basics

• Similar to C language but for describing hardware
• Description can be at different levels:
  – Behavioral
  – Register-transfer
  – Structural or gate level
• Not just a specification language – also associated with simulation environment
• Considered easier and “lighter weight” compared to VHDL
• Very popular in industry
Structure of a Module

- Verilog design contains interconnected **modules**
- A module has collections of low-level gates, statements and instances of other modules (similar to VHDL entity/architecture)
- Example:

```verbatim
// Function: 2-to-1 multiplexer
module mux2to1 (out, outbar, a, b, sel);

  output out, outbar;
  input a, b, sel;

  assign out = sel ? a : b;
  assign outbar = ~out;

endmodule
```
Structure of a Module

- Verilog design contains interconnected **modules**
- A module has collections of low-level gates, statements and instances of other modules (similar to VHDL entity/architecture)
- Example:

  ```
  // Function: 2-to-1 multiplexer
  module mux2to1 (out, outbar, a, b, sel);

  output out, outbar;
  input a, b, sel;

  assign out = sel ? a : b;
  assign outbar = ~out;

  endmodule
  ```

  - Use // to comment; can also use /* ... */
  - Declare and name **module**; list the ports; terminate with ';
  - Specify port type as **input**, **output** or **inout** if bidirectional
  - Express module behavior
  - Each statement executes concurrently
  - Order does not matter!
Continuous Assignment

- Keyword **assign** specifies continuous assignment to describe combinational logic
- Right-hand continuously evaluated, immediate response to inputs
- Left-hand is a *net* driven with evaluated value
- Left side must be scalar, a net or concatenation or nets or vector nets
- All continuous assignments execute concurrently
- Operators are low-level:
  - Conditional assignment: `(condition) ? vTrue : vFalse`
  - Boolean: `~`, `&`, `|`
  - Arithmetic: `+`, `-`, `*`
- Operators can be nested:

  ```
  assign out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0);
  ```
Verilog Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>concatenation</td>
</tr>
<tr>
<td>+ - * /</td>
<td>arithmetic</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
</tr>
<tr>
<td>&gt; &gt;= &lt; &lt;=</td>
<td>relational</td>
</tr>
<tr>
<td>!</td>
<td>logical NOT</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>logical equality</td>
</tr>
<tr>
<td>!=</td>
<td>logical inequality</td>
</tr>
<tr>
<td>?:</td>
<td>conditional</td>
</tr>
<tr>
<td>~</td>
<td>bit-wise NOT</td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bit-wise XOR</td>
</tr>
<tr>
<td>^^ ~~~</td>
<td>bit-wise XNOR</td>
</tr>
<tr>
<td>~&amp;</td>
<td>reduction AND</td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td><del>&amp;</del></td>
<td>reduction NAND</td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>reduction XOR</td>
</tr>
<tr>
<td>^^ ~~~</td>
<td>reduction XNOR</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
</tr>
</tbody>
</table>
Gate Level Description

- Built-in logic primitives:
  - and, nand, or, nor, xor, xnor, not, buf

- Can use arbitrary number of inputs

- Tri-state buffers: bufif1 and bufif0

- Declare nets (signals, wires) using wire keyword

- An instantiation of an AND gate:
  - and a1 (out1, a, sel);

```verilog
// Function: 2-to-1 multiplexer
module mux_gate (out, outbar, a, b, sel);

output out, outbar;
input a, b, sel;
wire out1, out2, selb;

not i1 (selb, sel);
and a1 (out1, a, sel);
and a2 (out2, b, selb);
or o1 (out, out1, out2);
not i2 (outbar, out);

endmodule;
```
Procedural Assignment & “always”

- Keywords always and initial are used to define procedural assignments, similar to procedures in software
- Good for behavioral descriptions of hardware
- Support for C-like constructs such as if, for, while, case, etc.

```verilog
module mux_2_to_1 (out, outbar, a, b, sel);
output out, outbar;
input a, b, sel;
reg out, outbar;

always @ (a or b or sel)
begin
  if (sel) out = a;
  else out = b;
  outbar = ~out;
end
endmodule
```

- Input and output declarations as before
- Assignment in an always block must be declared as variable type such as reg
- always runs whenever a signal in the sensitivity list changes values
- Statements inside the always block are executed sequentially
- Sandwicched between begin & end
Verilog “register” – NOT What it Appears

- Registers normally represent storage elements in digital logic – they need clock signals to update their output value.
- In Verilog, `reg` artifacts are not same as physical registers, used only to declare a variable for holding the value.
- Values of variables (declared as `reg`) can be changed anytime in simulation, and can be used for nets of combinational circuit.
Mixing Procedural & Continuous

- Procedural and continuous assignments can co-exist
- In procedural assignments, variables declared as `reg` changed only when procedural block is invoked via sensitivity list
- In continuous assignments, right-hand expression constantly evaluated and the left-side net updated instantaneously

```verilog
modulemux_2_to_1(out, outbar, a, b, sel);
output out, outbar;
input a, b, sel;
reg out;
always @(a or b or sel)
begin
  if (sel) out = a;
  else out = b;
end
assign outbar = ~out;
endmodule
```
The “case” Statement

- The `case` statement can often replace `if-else` constructs inside an `always` block, and provides better abstraction.

- An example:

```verilog
always @ (a or b or sel) begin
  case (sel)
    1'b0: out = b;
    1'b1: out = a;
  endcase
end
```

Notation for numbers:

<table>
<thead>
<tr>
<th>&lt;size&gt;'&lt;base&gt;&lt;number&gt;</th>
<th>2-bit binary, value=2</th>
<th>Unsigned binary 32-bit, value=2</th>
<th>Unsigned decimal, value=31</th>
<th>8-bit hex, value=175</th>
<th>16-bit negative decimal, value=-47</th>
</tr>
</thead>
<tbody>
<tr>
<td>2'b10</td>
<td>2-bit binary, value=2</td>
<td>Unsigned binary 32-bit, value=2</td>
<td>Unsigned decimal, value=31</td>
<td>8'hAf</td>
<td>16-bit negative decimal, value=-47</td>
</tr>
</tbody>
</table>
Buses – n-bit Signals

- Verilog is powerful in specifying buses
- Example: 8-bit wide 2-to-1 MUX

```verilog
module mux_2_to_1 (out, outbar, a, b, sel);
    input[7:0] a, b,
    input sel;
    output[7:0] out, outbar;
    reg[7:0] out;

    always @ (a or b or sel)
    begin
        if (sel) out = a;
        else out = b;
    end

    assign outbar = ~out;
endmodule
```

Concatenate signals using the `{ }` operator
```
assign {b[7:0], b[15,8]} = {a[15:8], a[7:0]};
// effects a byte swap
```
module reg4(d0, d1, d2, d3, en, clk, q0, q1, q2, q3);
    input d0, d1, d2, d3, en, clk;
    output q0, q1, q2, q3;
    wire d0, d1, d2, d3, en, clk;
    wire q0, q1, q2, q3;
    wire n_0;
    LATCH stored_d0_reg(.CLK (n_0), .D (d0), .Q (q0));
    LATCH stored_d1_reg(.CLK (n_0), .D (d1), .Q (q1));
    LATCH stored_d3_reg(.CLK (n_0), .D (d3), .Q (q3));
    LATCH stored_d2_reg(.CLK (n_0), .D (d2), .Q (q2));
    AND2X1 g21(.A (en), .B (clk), .Y (n_0));
endmodule
Language Overload?

- Most of you are experts with VHDL
  - Now you've been exposed to Verilog
- VHDL and Verilog compete in that they exist for same purposes
- Verilog, like VHDL, starts as a *modeling* language
- A subset of Verilog is synthesizable, used to generate circuits
- Take time to explore modeling capabilities as well
  - Remember: verification is critical!
  - Modeling and high-level specification aid verification
- For projects: Either Verilog or VHDL OK... be ready for anything in real world!
Language Overload?

- **SKILL** – scripts to manage custom design
  - Python can also be useful for similar purposes
- **VHDL & Verilog** – hardware description languages for high-level design specification, modeling and *design entry*
- **Tcl** (Tool Command Language) – made to be simple (aren't they all); used a lot for tool management, constraint setup, etc.
- **SystemC & SystemVerilog** – similar to HDL but even higher levels of abstraction
- **BlueSpec** (used with **SystemVerilog**), **Haskell** – Verification languages
- **CHISEL** – high-level code generators... is it useful?
Summary

- Verilog basics for digital system design
- Use either VHDL or Verilog for class projects, should expose yourself to many languages
- All of these languages are tools, each with different purposes
- We will explore some other languages as we go

- Challenge: Wasn't this a hardware design class?

YES! … but programming skills are critical to simplying the design of very/ultra large scale integrated systems