Lecture 2: Verilog

David Black-Schaffer
davidbbs@stanford.edu
EE183 Spring 2003

Overview

- EE183 Design Process
  - Understand the problem and do a hierarchical decomposition
- Verilog is an event-driven modeling language
  - Ideal for large circuits operating in parallel
- Verilog reference material
  - Continuous = you derive the logic (assign)
  - Procedural = the tools derive the logic (case, if)
- All storage elements must be explicitly instantiated
  - You must use the 183lib.v flip-flops for all storage

Logistics

- Any questions?
- Started the tutorial yet?
  - Due next Wednesday by midnight
    - You should have lab & computer access now
  - If a board doesn’t work check the “Tao of 183” page
  - Make sure the clock is set to 50MHz (divider of 2)
  - Run the XTEST program
  - Lab 1 pre-lab due next Friday
  - All submissions are PDF format email/URL
    - See “Tao of EE183” on web
  - Pick up CDs to install at home
    - Do NOT install the MultiLINX software
    - You must return the CDs (we have only 10 copies)

Differences from EE121

- Larger, Faster Systems
- No Schematic Entry
- Verilog hardware description language
  - Easier to learn than VHDL (more C-like)
  - Other Stanford classes use it
  - More popular in the Valley
- Let the tools do a lot of the dirty work
- But, ALWAYS know what logic you are synthesizing. This is key!
EE183 Design Process

1. Understand the problem and choose a hierarchical decomposition
2. Design the FSMs (bubble diagrams) and Data Path (block diagram)
3. Code the FSMs in verilog
4. Simulate the FSMs
5. Connect the FSMs to the data path in a top-level verilog file
6. Run the static timing tool to check timing
7. Test it out, then repeat the above until it works

What is Verilog?

- Really an event-driven modeling language
- Ideally suited for modeling large circuits where everything operates in parallel.
- Two sides to Verilog:
  - Structural (gates, logic, storage)
  - Behavioral (for loops, initial statements, delays)
- Only the structural elements will synthesize
  - What hardware would a for loop map into?
  - Good for simulations only

The Verilog module

```verilog
module synchronizer (in, out, clk);
  input in;
  input clk;
  output out;
  wire x;
dff dff_1 (.d(in), .clk(clk), .q(x));
dff dff_2 (.d(x), .clk(clk), .q(out));
endmodule
```

Ref: Lexical Conventions

- The lexical conventions are close to the programming language C++.
- Comments are designated by // to the end of a line or by /* to */ across several lines.
- Keywords, e.g., module, are reserved and in all lower case letters.
- The language is case sensitive, meaning upper and lower case letters are different.
- Spaces are important in that they delimit tokens in the language.
Ref: Number Specification

- Numbers are specified in the traditional form of a series of digits with or without a sign but also in the following form:
  - `<size>`<base format>`<number>`
  - where `<size>` contains decimal digits that specify the size of the constant in the number of bits. The `<size>` is optional. The `<base format>` is the single character followed by one of the following characters b, d, o and h, which stand for binary, decimal, octal and hex, respectively. The `<number>` part contains digits which are legal for the `<base format>`.
  - Some examples:
    - `4'b0011` // 4-bit binary number 0011
    - `5'd3` // 5-bit decimal number
    - `32'hdeadbeef` // 32 bit hexadecimal number

- What is 12?

Ref: Bitwise Operators

- Bitwise operators operate on the bits of the operand or operands.
- For example, the result of A & B is the AND of each corresponding bit of A with B. Operating on an unknown (x) bit results in the expected value. For example, the AND of an x with a FALSE is an FALSE. The OR of an x with a TRUE is a TRUE.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Bitwise negation</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td>~&amp;</td>
<td>Bitwise NAND</td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td>=~ or ^=</td>
<td>Equivalence (Bitwise NOT XOR)</td>
</tr>
</tbody>
</table>

Ref: Logical Operators

- Logical operators operate on logical operands and return a logical value, i.e., TRUE(1) or FALSE(0).
- Used typically in if and while statements.
- Do not confuse logical operators with the bitwise Boolean operators. For example, `|` is a logical NOT and `~` is a bitwise NOT. The first negates, e.g., `5 == 6` is TRUE. The second complements the bits, e.g., `~{0,0,1,1}` is 0100.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Logical negation</td>
</tr>
<tr>
<td>&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ref: Unary Reduction Operators

- Unary reduction operators produce a single bit result from applying the operator to all of the bits of the operand. For example, `&A` will AND all the bits of A.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>AND reduction</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>XOR reduction</td>
</tr>
<tr>
<td>~&amp;</td>
<td>NAND reduction</td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td>=~ or ^=</td>
<td>XNOR reduction</td>
</tr>
</tbody>
</table>

- You may use these for the zero test in lab 3, but otherwise they aren’t all that useful.
Ref: Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>Compares two operands and returns a logical value, i.e., TRUE(1) or FALSE(0)</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
<td>Compares two operands and returns a logical value, i.e., TRUE(1) or FALSE(0)</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>Compares two operands and returns a logical value, i.e., TRUE(1) or FALSE(0)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
<td>Compares two operands and returns a logical value, i.e., TRUE(1) or FALSE(0)</td>
</tr>
<tr>
<td>!=</td>
<td>Logical inequality</td>
<td>Compares two operands and returns a logical value, i.e., TRUE(1) or FALSE(0)</td>
</tr>
</tbody>
</table>

What do these synthesize into?

- If any bit is unknown, the relation is ambiguous and the result is unknown – should never happen!

Ref: Miscellaneous Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ , ]</td>
<td>Concatenation of nets</td>
</tr>
<tr>
<td></td>
<td>Joins bits together with 2 or more comma-separated expressions, e.g., ([A[0], B[1:7]]) concatenates the zeroth bit of (A) to bits 1 to 7 of (B).</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Shift left (Multiplication by power of 2)</td>
</tr>
<tr>
<td></td>
<td>Vacated bit positions are filled with zeros, e.g., (A = A &lt;&lt; 2); shifts (A) two bits to left with zero fill.</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Shift right (Division by power of 2)</td>
</tr>
<tr>
<td></td>
<td>Vacated bit positions are filled with zeros.</td>
</tr>
<tr>
<td></td>
<td>What about 2s complement (negative) numbers?</td>
</tr>
<tr>
<td>?:</td>
<td>Conditional (Creates a MUX)</td>
</tr>
<tr>
<td></td>
<td>Assigns one of two values depending on the conditional expression. E.g., (A = C &gt; D ? B + 3 : B - 2); means if (C) greater than (D), the value of (A) is (B + 3) otherwise (B - 2).</td>
</tr>
</tbody>
</table>

How to use this stuff

- Continuous Assignment
  - Uses the assign statement
  - Wire things up explicitly (MUXes, ANDs, etc.)
  - You figure out the logic on your own

- Procedural Assignment
  - Uses if/else and case statements
  - Wire things up implicitly (write behavioral code)
  - The tools determine the logic from your code

Continuous Assignment

- assign out = in1 & in2;
  - Amazingly enough creates an “and” gate!
  - Anytime right hand side (RHS) changes, left hand side (LHS) is updated
  - LHS must be a wire

- Good for explicitly creating MUXes and adders
  - assign alu_in = zeros ? 12'b0 : alu_input_a;
  - assign next_count = previous_count + 8'b1;
### Procedural Assignments

- We will only use them to define combinational logic (as a result, blocking = and non-blocking <= assignment are the same)

```verilog
reg out;
always @(in1 or in2)
begin
  out = in1 & in2;
end
```

- LHS must be of type reg
- Does NOT mean this is a DFF

- All input signals must be in sensitivity list, i.e., anything that can cause the output to change

- Begin and End define a block in Verilog

### If-Else Conditional Procedural Assignment

- Just a combinational logic mux
- Every if must have matching else or state element will be inferred—why?

```verilog
always @(control or in)
begin
  if (control == 1'b1)
  out = in;
end
```

- What is the value of `out` if `control` is not `1'b1`?

**This is a big EE183 no-no. We will be looking for this.**

### Case Statement Procedural Assignment

```verilog
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
  case ({s1, s0})
    2'b00: out = i0;
    2'b01: out = i1;
    2'b10: out = i2;
    2'b11: out = i3;
    default: out = 1'bx;
  endcase
end
endmodule
```

- Note how all nets that are inputs to the always block are specified in the sensitivity list
- Make sure all 2^n cases are covered or include a "default" statement or else state elements will be inferred
- X is don’t care
- After initial synchronous reset there should never be any X’s in your design

### So how do I get D-FlipFlops?

- Use 183lib.v to instantiate them
-dff, dffd, dffr
- These are the *only* state elements (except for CoreGen RAMs) allowed in your design
- Let me say that again: These are the *only* state elements (except for CoreGen RAMs) allowed in your design
## Dffre guts

```vhdl
// Dffre: D flip-flop with active high enable and reset
// Parametrized width; default of 1
module dffre (d, en, r, clk, q);
    parameter WIDTH = 1;
    input en;
    input r;
    input clk;
    input [WIDTH-1:0] d;
    output [WIDTH-1:0] q;
    reg [WIDTH-1:0] q;
    always @ (posedge clk)
        if (r)
            q <= {WIDTH{1'b0}};
        else if (en)
            q <= d;
        else
            q <= q;
    endmodule
endmodule
```

WIDTH is a parameter which can be used elsewhere in the module. It can be changed when the module is instantiated in another module.

Only change LHS on "posedge clk". Note the use of the <= assignment which allows for storage.

Replicator Operator. “Make WIDTH zeros”

## Use Case Statement for FSM

- Instantiate state elements as `dff`
- Put next state logic in `always @( )` block
  - Input is `curstate (.q of dff)` and other inputs
  - Output is `nextstate` which goes to `.d of dff`
- Use combined `case` and `if` statements

Synthesis tools auto-magically minimizes all combinational logic.
- Three cheers for synthesis!
- See the tutorial for an in-depth example

## 8-bit Counter

```vhdl
module counter_8 (clk, reset, en, cntr_q);
    input clk;
    input reset;
    input en;
    output [7:0] cntr_q;
    reg [7:0] cntr_d;
    wire [7:0] cntr_q;
    // Counter next state logic
    always @(cntr_q)
        begin
            cntr_d = cntr_q + 8'b1;
        end
    // Counter state elements
    dffre #(8) cntr_reg (.clk(clk), .r(reset), .en(en), .d(cntr_d), .q(cntr_q));
endmodule
```
CoreGen

- Core Generator
  - Useful info appears in “language assistant”—Read it!
- Only use this for memories for now
  - Do you need anything else?
    - FIR filters for Lab 4
    - Multipliers for Lab 2
    - Other things?
- Caveat: Block Memory does not simulate correctly with initial values.
  - Try using a distributed memory or talk to Joel

Important

- Get ID Card Access & Lab Account
  - Write your ID#, email, and phone # on the list
  - See me after class for a computer account
- Start on the Tutorial ASAP
  - Downloads on the handouts page
  - Due next Wednesday at midnight
  - 6-8 hours to complete
  - Covers all the basics for EE183

Lecture 2 Key Points

- All the verilog you’ll need for your designs is in this lecture
- You must explicitly instantiate all storage elements using the 183 library
- Use the tools to do your minimization work. Concentrate on the overall design.

- Logistics
  - How is the tutorial going?