Today (1)

- Lab #2 Solution
- Simulation vs Hardware
- Debugging
  - Goals
  - Tips
  - Algorithm
- Administrative Info

Lab #2 Solution (1)

```verilog
module Accumulator(In, Out, Enable, Clock, Reset);
  input [7:0] In;
  output [7:0] Out;
  input Enable;
  input Clock, Reset;
  reg [7:0] Out;
  always @ (posedge Clock) begin
    if (Reset) Out <= 8'h00;
    else if (Enable) Out <= Out + In;
  end
endmodule
```
Lab #2 Solution (2)

- Accumulator
  - Simple, easy to build
  - What is the actual circuit?
    - Get used to answering this in your head
    - RTL View from Lab #1 (Section 4.3 Synthesis)

- Peak Detector
  - Hard to build
  - Minute control of hardware
  - Tools couldn’t optimize though!!

Lab #2 Solution (3)

Simulation vs. Hardware (1)

- Debugging in Simulation
  - Slow Running Time
  - Fast Debugging
    - Waveforms
    - Text messages
  - Full Visibility
    - Can examine any signal
  - Easy to Fix
    - A few minutes to compile and resimulate

Simulation vs. Hardware (2)

- Debugging in Hardware
  - Fast Running Time
    - Full speed in fact
  - Slow Debugging
    - Synthesis can take hours
  - Little or No Visibility
    - Very hard to probe signals
  - Maybe Impossible to Fix (ASICs)
Simulation vs. Hardware (3)

- Simulation
  - Functional Testing & Verification
    - Test everything at least minimally
    - Fully Verify what you can
  - This will save you many sleepless nights
- Hardware
  - Debugging
    - Treat this as a last resort
    - It is painful

Debugging (1)

- Debugging Algorithm
  - Hypothesis: What’s broken?
  - Control: Give it controlled test inputs
  - Expected Output: What SHOULD it do?
  - Observe: Did it work right?
  - If it broke: THAT’S GREAT!
    - If we can’t break anything like this then the project must be working...

Debugging (2)

- First, check for Verilog syntax errors
  - Run Synthesis on the module
  - View Synthesis report to see errors
- Don’t debug randomly
  - Just changing things at random often makes things look fixed
  - It won’t really help
  - Debug systematically
  - Your first design may be the best

Debugging (3)

- High Level Debugging
- Localize the problem
  - N64? SDRAM? Video?
- Test Patterns
  - Lets you easily isolate the broken component
  - If you know exactly what’s going in you can check what’s coming out
Debugging (4)

- Simulate the broken component(s)
  - Writing test benches takes less time than sitting around wondering why it's broken
  - Everyone hates writing testbenches
    - (Even the TA’s)
    - You will hate hardware debugging more
    - Get used to it

Debugging (5)

- Your best debugging tool is logic
  - If 3 out of 4 components work, what's broken?
  - Question all your assumptions!
    - Just because you think it's true doesn't mean it is
    - 90% of debugging time is wasted debugging the wrong problem otherwise
    - Given solutions and modules may not work the way you expect!

Debugging (6)

- Before you change anything
  - Understand exactly what the problem is
  - Find an efficient solution
  - Evaluate alternative solutions

- After the change
  - Fixes may make things worse sometimes
    - May uncover a second bug
    - May be an incorrect fix
  - Repeat the debugging process

Debugging (7)

- Ask around
  - Someone else may have had the same bug
  - They'll probably at least know about where the problem is
  - Different bugs may produce the same results

- TAs
  - The TAs know common problems
  - We're here to help, not solve it for you
Administrative Info

- Midterm I
  - Thursday 02/16, 2-3:30pm, Room 125 Cory
  - Review Session TBA, 125 Cory
- Partners
  - You MUST have one for this week
  - Try someone other than your best friend
  - Restrictions
    - You can change partners until the project starts
    - You must be in the same lab
- Project in 2 weeks

Part1: Bottom Up Testing (1)

What if EqualOut = 1'b0 and GreaterOut = 1'b0?

Part1: Bottom Up Testing (2)

- Exhaustive Testing
  - Ideal Testing Method
    - Circuit is 100% tested!
    - Requires us to test a LOT!
    - Can we do it here? (2^4 possible inputs)
  - Method
    - Make a truth table
    - Have the testbench generate all inputs
    - Make sure outputs match truth table

Part1: Bottom Up Testing (3)
Part1: Bottom Up Testing (4)

- Exhaustive Testing?
  - $2^8 = 256$ Possible Inputs
- Method
  - Use a for loop to generate all inputs
    - Loops allowed only in testbenches
    - They will not synthesize
  - Compare against a "$\geq\$"
    - Print a message if they differ

Part1: Bottom Up Testing (5)

Exhaustive Testing?
- $2^4 = 16$ Possible Inputs
- $2^4 = 16$ Possible States
- $16 \times 16 = 256$ combinations
- We could do it in this case
- Can’t exhaustively test FSMs
  - Too many state/input combinations
  - Must rely on directed testing

Part1: Bottom Up Testing (6)

Part1: Bottom Up Testing (7)
Part 1: Bottom Up Testing (8)
- Read Test Vectors from a File
- Designing Test Vectors
  - Make sure to cover most cases
    - We want 95%+ coverage
  - Designing test vectors is a “black art”
- $ Processes
  - Not synthesizeable
  - More information in IEEE Verilog Reference

Part 2: Test Hardware (1)
- A

Part 2: Test Hardware (2)
- Test Procedure
  - Hit Reset (SW1)
  - Hit Go (SW2)
  - Record an error
    - DD1-8 show \{A, B\}
    - SW10[1] selects the sum on DD4-8
  - Hit Go
  - Repeat until the tester stops

Part 2: Test Hardware (3)
- The Broken Adder
  - 16bit Adder
    - \(2^{32} \approx 4\) Billion Test Vectors
    - Can’t simulate this much
    - 2:40 to test this at 27MHz
  - Fail Modes
    - 0: No Errors
    - 2: Will claim \(1 + 1 = 3\)
    - 1-3: Can have anywhere from 0 to 4 errors
Part3: FSM Testing (1)

- Exhaustive Testing Again!
  - Check every arc
  - Check every output
- You don’t need to correct this one...
  - We’re not giving you the source code
- Boring (and Easy)
  - You will have FSM bugs
  - Get used to debugging them