Introduction to Verilog

Mitch Trope
mtrope@ittc.ku.edu
EECS 240
Spring 2004
Overview

- What is Verilog?
- Verilog History
- Max+Plus II
  - Schematic entry
  - Verilog entry
- System Design Using Verilog: Sum of Products
  - Define the System
  - Define the Relationships
  - Use Verilog Primitives
  - Behavioral Design
- System Design Using Verilog: Multiplexer
- Other Verilog Syntax
- Submitting Verilog for Homework
What is Verilog?

- Hardware description language
- Used to develop simulations and implementations of hardware systems
- Syntax is similar to Pascal or C
- It’s not VHDL....
Verilog History

- Originally developed for simulating gate-level design in 1984
- Synopsis developed a synthesis tool that used Verilog as an input for register-level design
- Used for timing certification starting in 1989
- Open Verilog Initiative created in 1990 to respond to VHDL
- IEEE standard in 1995
Verilog Competitor: VHDL

- Very High Speed Integrated Circuit Hardware Description Language
- Another hardware description language
- Originally developed by DARPA to provide standard descriptions of military hardware
- Syntax is like C
What do we use Verilog for?

- Circuit design
- Circuit simulation
- Circuit synthesis
- Timing analysis
- Power analysis
- Just about anything in the hardware design path...
How is Verilog used today?

- Design ASICs
- Design FPGA implementations
  - Xilinx ISE and Altera Max+Plus II software suites support both VHDL and Verilog
- Verify system characteristics
- Teaching digital design....
Max+Plus II

- Altera’s system design software
- Installed on EECS 240 lab machines, department Windows machines
- Tutorial in the book in Appendix B
- You will need to learn schematic entry and Verilog entry methods
Schematic Entry

- Specify gates or components to be used in block diagram format
- Draw out physical connections between components
- Directly route off-chip input pins to the circuit
- **Floorplan** - Specify logic resources used by each component
Verilog Entry

- Describe system in terms of Verilog statements
- Compiler translates Verilog into sequences of gates
- Several “black box” layers available
- Describe a hardware system like a software system
- Floorplan editor still available, but extra abstraction layer increases difficulty of use
What Else is There?

- Simulation
  - See how your design will perform with a given set of inputs
  - Evaluate the correctness of your design
- Timing Analysis
  - Adjust gate delays to mimic actual hardware, and see if your system is still correct
- Synthesis
  - Drop your design in to hardware and watch it go...
- All this is covered in Appendix B...
Introduction to Verilog Systems Design

- Concepts
  - Node analysis
  - Gate-level design
  - Component Design
    - Comes-in-to
    - Goes-out-of

- Your book is an excellent syntax guide
- Some syntax will be presented
Start With What We Know…

Diagram:

- A connected to OUT1
- B connected to OUT1
- C connected to OUT2
- D connected to OUT2
- OUT1 connected to OUT3
Change the Format...

NODE1

A
B

OUT1

C
D

NODE2

OUT2

OUT3

NODE3
What does this buy us?

- In the first figure, we used discrete gates.
- In the second figure, we abstracted the signal path to data flow.
- They really are the same thing.
- Now, we can express connections in terms of node relationships.
Let’s Define the Relationships...

NODE1(OUT1, A, B)

NODE2(OUT2, C, D)

NODE3(OUT3, OUT1, OUT2)
Let’s Define the Relationships...

- Each node has an output parameter and input parameters listed in that order.
- NODE1(OUT1, A, B) has an output of OUT1 and inputs of A and B
  - Remember, this was the top AND gate in our original design.
- NODE2(OUT2, C, D) has an output of OUT2 and inputs of C and D
  - This was the bottom AND gate in our original design.
- NODE3(OUT3, OUT1, OUT2) has an output of OUT3 and inputs of OUT1 and OUT2
  - This was the OR gate in our original design.
Using Verilog Primitives

- Verilog primitives include AND, OR, NAND, NOR, XOR, XNOR
- Any system can be built using these primitives
- The primitives have the format: PRIM(OUTPUT, INPUT, INPUT)
- Let’s go back to our diagrams, and see if we can describe our system with Verilog...
Look Back at our System
and (OUT1, A, B);
and (OUT2, C, D);
or (OUT3, OUT1, OUT2);
Make it a Module

- We need to add some syntax in order to compile this

```verilog
module gate_logic (OUT3, A, B, C, D)
    output OUT3;
    input A, B, C, D;

    and(OUT1, A, B);
    and(OUT2, C, D);
    or(OUT3, OUT1, OUT2);

endmodule
```
What did we add?

- The `module` and `endmodule` keywords allow us to form building blocks of code that can be instantiated elsewhere.
- The parameter list after the `module` keyword specifies the outputs and inputs.
- Note: module definitions cannot be nested.
- A module must have its outputs and inputs declared in the body of the module.
- Use the `output` and `input` keywords for this.
Verilog can describe the behavior of signals as well as the structure.

Different symbols are used to define behavioral Verilog.

Structural Verilog can always be synthesized, but writing it is much more involved.

Behavioral Verilog is more concise and easier to write, but less deterministic when you synthesize it.
Behavioral Symbols

- **AND**: & (ampersand)
- **OR**: | (pipe)
- **NOT**: ~ (tilde)
- **XOR**: ^ (carat)
Describe it in Behavioral Verilog

\[ \text{OUT3} = (A \& B) \mid (C \& D); \]
What Else Do We Need?

- Behavioral logic cannot be constantly evaluated by a simulator.
- You must include an `always` block that references a sensitivity list. 
- Sensitivity List – a group of signals watched by the simulator for changes – when one changes, the expression is re-evaluated.
- Syntax:
  ```
  always @(SIG1 or SIG2 or SIG3 or ....) 
  begin
  <expressions with SIG1, SIG2 and SIG3 and....> 
  end
  ```
Behavioral Verilog

- This code will re-evaluate the expression when any of the input signals change

```verilog
module gate_logic (OUT3, A, B, C, D)
    output OUT3;
    input A, B, C, D;

    always @(A or B or C or D)
    begin
        OUT3 = (A & B) | (C & D);
    end
endmodule
```
2-Input Multiplexer Example
Multiplexer in Verilog

- **Structural**
  
  ```
  NOT(NOTS, S);
  AND(OUT1, A, NOTS);
  AND(OUT2, B, S);
  OR(OUT3, OUT1, OUT2);
  ```

- **Behavioral**
  
  ```
  OUT3 = (A & ~S) | (B & S);
  ```
Full Verilog Multiplexor Implementations

```verilog
dmodule(OUT3,A,B,S)
output  OUT3;
input   A,B,S;

NOT(NOTS,S);
AND(OUT1,A,NOTS);
AND(OUT2,B,S);
OR(OUT3,OUT1,OUT2);
endmodule
```

```verilog
dmodule(OUT3,A,B,S)
output  OUT3;
input   A,B,S;

always @(A,B,S)
begin
  OUT3 = (A & ~S) | (B & S);
end
endmodule
```
Other Bits of Syntax

- if-else conditional
  ```
  if <condition>
    <statement1>;
  else
    <statement2>;
  ```

- case statement
  ```
  case <vector>
    <case_1>: <statement1>;
    <case_2>: <statement2>;
    default: <statement3>;
  endcase
  ```
Comments in Verilog

- Comments in any code are very important – often more important than the code itself.
- Comment Syntax:
  ```
  // this is a single line comment
  */
  Hello,
  I am a multi-line comment.
  */
  ```
Submitting Verilog for Homework - Guidelines

- Source code **MUST** be typed. No handwritten code will be accepted.
- All source code must be accompanied by simulation output.
- All source code must include your name, KUID, assigned problems and the assignment date at the top of the file (use comments to put this in)
- All source code must be commented. Describe the inputs and outputs, and what your component does.
- All submissions must be **YOUR OWN WORK**.
Submitting Verilog for Homework - Tips

- Don’t wait until the last minute to learn the software
  - Learning the tool chain is often the hardest part of learning a new language
- When you have an assignment, start on it as soon as possible
- Use your textbook as a language reference
- Ask for help if you need it!
Questions?

mtrope@ittc.ku.edu