Lecture 1: Verilog HDL Introduction

What is Verilog HDL?

- Verilog Hardware Description Language (HDL)?
  - A high-level computer language can model, represent and simulate digital design
    - Hardware concurrency
    - Parallel Activity Flow
    - Semantics for Signal Value and Time
  - Design examples using Verilog HDL
    - Intel Pentium, AMD K5, K6, Atheon, ARM7, etc
    - Thousands of ASIC designs using Verilog HDL

What is VHDL?

- VHDL represents another high level language for digital system design.
- In this course we study Verilog HDL
  - reason:
    - used more often in electronic and computer industry
    - programming style is very similar to C programming language

Basic Design Methodology
Modeling Digital Systems

- Verilog HDL is for writing models of a system
- Reasons for modeling
  - requirements specification
  - documentation
  - testing using simulation
  - formal verification
  - synthesis
- Goal
  - most reliable design process, with minimum cost and time
  - avoid design errors!

Domains and Levels of Modeling

- Functional
  - Algorithm (behavioral)
  - Register-Transfer Language
  - Boolean Equation
  - Differential Equation
- Structural
- Geometric

“Y-chart” due to Gajski & Kahn
Domains and Levels of Modeling

Structural

Functional

Polygons

Sticks

Standard Cells

Floor Plan

Geometric

“Y-chart” due to Gajski & Kahn

Verilog HDL Models

- HDL model specifies the relationship between input signals and output signals
- HDL uses special constructs to describe hardware concurrency, parallel activity flow, time delays and waveforms

Verilog code for a AND gate

```verilog
module and_gate(y, x1, x2);
input x1, x2;
output y;
and(y, x1, x2);
endmodule
```

Verilog Examples

```verilog
module Add_half (sum, c_out, a, b);
output sum, c_out;
input a, b;
wire c_out_bar;
xor G1 (sum, a, b);
nand G2 (c_out_bar, a, b);
not G3 (c_out, c_out_bar);
endmodule
```

* the instance name of Verilog primitives is optional.

Verilog Example: A D Flip Flop

```verilog
module Flip_Flop (q, data_in, clk, rst);
input data_in, clk, rst;
output q;
reg q;
always @(posedge clk)
begin
  if (rst == 1) q = 0;
  else q = data_in;
end
endmodule
```
Verilog Example: behavioral model

module adder_4_RTL (a, b, c_in, sum, c_out);
output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;
assign {c_out, sum} = a + b + c_in;
endmodule

Verilog Example: Structural Models

- Structural models
  - Are built from gate primitives and/or other modules
  - They describe the circuit using logic gates — much as you would see in an implementation of a circuit.
    - You could describe your assignment 1 circuit this way
- Identify
  - Gate instances, wire names, delay from a or b to f:

```verilog
module mux (f, a, b, sel);
output f;
input a, b, sel;
and #5 g1 (f1, a, nsel),
g2 (f2, b, sel);
or    #5 g3 (f, f1, f2);
not g4 (nsel, sel);
endmodule
```

Verilog Example: Gate-Level Models

- Need to model the gate’s:
  - Function
  - Delay
- Function
  - Generally, HDLs have built-in gate-level primitives
    - Verilog has NAND, NOR, AND, OR, XOR, XNOR, BUF, NOT, and some others
  - The gates operate on input values producing an output value
    - typical Verilog gate instantiation is:

```verilog
optional "many"
and #delay instance-name (out, in1, in2, in3, ...);
```

How to build and test a module

Construct a “test bench” for your design
- Develop your hierarchical system within a module that has input and output ports (called “design” here)
- Develop a separate module to generate tests for the module (“test”)
- Connect these together within another module (“testbench”)
module testbench ();
    wire l, m, n;
design d (l, m, n);
test t (l, m);
    initial begin
    //drive the outputs with signals

module design (a, b, c);
    input a, b;
    output c;
    ...

module test (q, r);
    output q, r;
    initial begin
    //monitor and display

module testbench ();
wire l, m, n;
design d (l, m, n);
test t (l, m);
    initial begin
    //monitor and display

module testAdd(a, b, sum, cOut);
    input sum, cOut;
    output a, b;
    reg a, b;
    initial begin
    $monitor ($time,,
        "a=%b, b=%b, sum=%b, cOut=%b",
        a, b, sum, cOut);
        a = 0; b = 0;
        #10 b = 1;#10 a = 1;
        #10 b = 0;
        #10 $finish;
    end
endmodule

module halfAdd (sum, cOut, a, b);
    output sum, cOut;
    input a, b;
xor #2 (sum, a, b);
    and #2 (cOut, a, b);
endmodule

module testAdd(a, b, sum, cOut);
    input sum, cOut;
    output a, b;
    reg a, b;
    initial begin
    $monitor ($time,,
        "a=%b, b=%b, sum=%b, cOut=%b",
        a, b, sum, cOut);
        a = 0; b = 0;
        #10 b = 1;#10 a = 1;
        #10 b = 0;
        #10 $finish;
    end
endmodule

Another view of this
• 3 chunks of verilog, one for each of:

TESTBENCH is the final piece of hardware which connect DESIGN with TEST so the inputs generated go to the design you want to test...

Another piece of hardware, called TEST, to generate interesting inputs

Your hardware called DESIGN

Verilog Examples
Module testAdd generated inputs for module halfAdd and displayed changes. Module halfAdd was the design

module testAdd(a, b, sum, cOut);
    input sum, cOut;
    output a, b;
    reg a, b;
    initial begin
    $monitor ($time,,
        "a=%b, b=%b, sum=%b, cOut=%b",
        a, b, sum, cOut);
        a = 0; b = 0;
        #10 b = 1;#10 a = 1;
        #10 b = 0;
        #10 $finish;
    end
endmodule

module halfAdd (sum, cOut, a, b);
    output sum, cOut;
    input a, b;
xor #2 (sum, a, b);
    and #2 (cOut, a, b);
endmodule

The test module
• It’s the test generator
• $monitor
  – prints its string when executed.
  – after that, the string is printed when one of the listed values changes.
  – only one monitor can be active at any time
  – prints at end of current simulation time

module testAdd(a, b, sum, cOut);
    input sum, cOut;
    output a, b;
    reg a, b;
    initial begin
    $monitor ($time,,
        "a=%b, b=%b, sum=%b, cOut=%b",
        a, b, sum, cOut);
        a = 0; b = 0;
        #10 b = 1;
        #10 a = 1;
        #10 b = 0;
        #10 $finish;
    end
endmodule
The test module (continued)

- Function of this tester
  - at time zero, print values and set a=b=0
  - after 10 time units, set b=1
  - after another 10, set a=1
  - after another 10 set b=0
  - then another 10 and finish

```verilog
module testAdd(a, b, sum, cOut);
input sum, cOut;
output a, b;
reg a, b;
initial begin
    $monitor ($time,, "a=%b, b=%b, sum=%b, cOut=%b", a, b, sum, cOut);
    a = 0; b = 0;
    #10 b = 1;#10 a = 1;
    #10 b = 0;
    #10 $finish;
end
endmodule
```

Structural vs Behavioral Models

- Structural model
  - Just specifies primitive gates and wires
  - i.e., the structure of a logical netlist
  - You basically know how to do this now.

- Behavioral model
  - More like a procedure in a programming language
  - Still specify a module in Verilog with inputs and outputs...
  - ...but inside the module you write code to tell what you want to have happen, NOT what gates to connect to make it happen
  - i.e., you specify the behavior you want, not the structure to do it

- Why use behavioral models
  - For testbench modules to test structural designs
  - For high-level specs to drive logic synthesis tools

```verilog
module ripple_adder (A, B, carry_in, carry_out, Out);
input A, B, carry_in;
output Out, carry_out;
assign {carry_out, Out} = A + B + carry_in;
endmodule
```
Summary

- Model hardware at different levels of abstraction
- Mix different levels of abstraction in description and simulation
- Able to model hardware concurrency
- Support Hierarchical decomposition
- Availability of ASIC Foundry Support