EEL 4783: Hardware/Software Co-design with FPGAs

Lecture 8: Short Introduction to Verilog

*

Prof. Mingjie Lin

* Beased on notes of Turfts lecture

Overview

- Recap + Questions?
- What is a HDL? Why do we need it? (simplified view)
  - Guess?
- Verilog (?)
  - History
  - Impact
  - Huge potential for research (surprise ☺)
- VHDL
- Verilog vs. VHDL

HDL (Hardware Description Language)

- HDL is a *language* used to describe a *digital* system, for example, a computer or a component of a computer.
- Most popular HDLs are VHDL and Verilog
  - Exotic ones: bluespec, …
- Verilog programming is similar to *C* programming
- VHDL programming is similar to *PASCAL* (some say like Ada)
  - Is an IEEE standard

Levels of description

- Switch Level
  - Layout of the wires, resistors and transistors on an IC chip
- Gate (structural) Level
  - Logical gates, flip flops and their interconnection
- RTL (dataflow) Level
  - The registers and the transfers of vectors of information between registers
- Behavioral (algorithmic) Level
  - Highest level of abstraction
  - Description of algorithm without implementation details

Tradeoffs between Abstraction Levels

- Behavioral level
  - Easiest to write and debug, not synthesizable
- Register Transfer Level
  - Synthesizable
  - Uses the concept of registers (a set of flipflops) with combinational logic between them
- Structural level
  - Very easy to synthesize
  - A text based schematic entry system

Why Do We Need HDL?

- *NO OTHER CHOICE*
- For large digital systems, gate-level design is unmanageable
- Millions of transistors on a digital chip
- HDL offers the mechanism to describe, test and synthesize large designs

Verilog Language

- Describe a system by a set of *modules* (~functions in C)
- Keywords, e. g., module, are reserved and in all lower case letters
  - Verilog is *case sensitive*
- Operators (some examples)
  - Arithmetic: +, - ! ~ * /
  - Binary operators: &, |, ^, ~, !
  - Shift: << >> Relational: <, <=, >, >=, ==, !=
  - Logical: &&, ||
- Identifiers
  - Equivalent to variable names
  - Identifiers can be up to 1024 characters
- Comments start with a "//" for one line or /* to */ across several lines

Number representation

- 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>*
  - <size>: number of bits (optional)
  - <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
  - <number>: contains digits which are legal for the
  - <base format>
- Examples
  - 'h 8FF // hex number
  - 4'b11 // 4-bit binary number 0011
  - 3'b10x // 3-bit binary # with least significant bit unknown
  - -4'b11 // 4-bit two's complement of 0011, or equivalently 1101

Data types

- Variables of type wires (wire) and registers (reg)
  - NOTE: A variable of type register does not necessarily represent a physical register
- Register variables store the last value that was procedurally assigned
- Wire variables represent physical connections between structural entities such as gates (Does not store anything, only a label on a wire)
- The reg and wire data objects may have the following possible values:
  - 0,1,x,z (0,1,unkown, high impedance of tri-state gate)
  - “reg” variables are initialized to 0 at the start of the simulation
  - “wire” variable not connected to something has the x value.

Data Types: Conceptual Differences

- *Wires/Nets* represent connections between things
  - Do not hold their value
  - Take their value from a driver such as a gate or other module
  - Cannot be assigned in an initial or always block
- *Regs* represent data storage
  - Behave exactly like memory in a computer
  - Hold their value until explicitly assigned in an initial or always block
  - Never connected to something
  - Can be used to model latches, flip-flops, etc., but do not correspond exactly
  - Shared variables with all their attendant problems

Program structure

- A digital system as a set of modules
- Each module has an interface to other module (connectivity)
- GOOD PRACTICE: Place one module per file (not a requirement)
- Modules run concurrently
- Usually there is a top level module which invokes instances of other modules

Module

- Represent bits of hardware ranging from simple gates to complete systems, e. g., a microprocessor
- Specified behaviorally, RTL, or structurally
- The structure of a module is the following:

module <module name> *(*<port list>*);*

<declarations>

<module items>

endmodule

example: NAND gate

Here is an RTL specification of a module NAND

```
// Behavioral model of a NAND gate

module NAND(in1, in2, out);
input in1, in2;
output out;

  // continuous assignment statement
  assign out = ~(in1 & in2);

endmodule
```

Default: All undeclared variables are wires and are one bit wide!

GOOD PRACTICE: Declare all variables

Instance of a module

The general form to invoke an instance of a module is:

<module name> <parameter list> <instance name> (<port list>);

<parameter list> are values of parameters passed to the instance

<instance name> identifies the specific instance of the module

An example parameter passed would be the delay for a gate

We will not use parameter list in this course!

- For our purposes, to invoke an instance of a module

  <module name> <instance name> (<port list>);

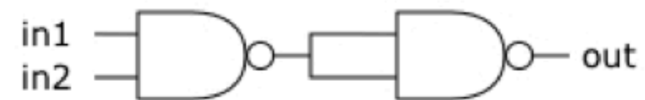
Structural example: AND gate

```
//Structural model of AND gate from two NANDS
module AND(in1, in2, out);
input in1, in2;
output out;

wire w1;

    // two instances of the module NAND
    NAND NAND1(in1, in2, w1);
    NAND NAND2(w1, w1, out);

endmodule
```



- This module has two instances of the NAND module called NAND1 and NAND2 connected together by an internal wire w1.

Continuous vs. procedural assignments

Continuous statement is used to model combinational logic

- Continuous assignments drive wire variables
- Evaluated and updated whenever an input operand changes value

Procedural assignment changes the state of a register

- Used for both combinational and sequential logic
- All procedural statements must be within "always" block

- Example

```
reg A;

always @ (B or C) begin
  A = B & C;
end
```

Events

The execution of a procedural statement is triggered by:

- A value change on a wire
- The occurrence of a named event

```
always @ (B or C) begin  // controlled by any value change in B or C
        X = B & C;
end

always @(posedge Clk) Y <= B&C;   // controlled by positive edge of Clk

always @(negedge Clk) Z <= B&C;   // controlled by negative edge of Clk
```

Model of a D-Flip flop

What is the behavior of a D-flipflop ?

- During every positive clock edge, the input is transferred to the output

RTL model

```
module Dflipflop(D, Clk, Q, Qbar);
input D, Clk;
output Q, Qbar;

reg Qint;

    // always is a procedural construct
    // any assignment may be made only to registers
    always @(posedge Clk) Qint <= D;

assign Q = Qint;
assign Qbar = ~Qint;

endmodule
```

Control constructs

Two control constructs are available:

```
if (A == 4)
      begin
         B = 2;
      end
else if (A == 2)
      begin
         B = 1;
      end
else
      begin
         B = 4;
      end
```

```
case (<expression>)
      <value1>:
          begin
         <statement>;
          end
                <value2>:
          begin
         <statement>;
          end
                default:
<statement>;
 endcase
```

VHDL

- NOT: Very Hard Difficult Language ☺
- VHSIC Hardware Description Language + VHSIC (Very High Speed Integrated Circuits)
- Benefits
  - VHDL is a programming language that allows one to model and develop complex digital systems in a dynamic envirornment.
  - Object Oriented methodology for you C people can be observed -- modules can be used and reused.
  - Allows you to designate in/out ports (bits) and specify behavior or response of the system.

Verilog vs. VHDL

- Verilog and VHDL are comparable languages
- VHDL has a slightly wider scope
- System-level modeling
- Exposes even more discrete-event machinery
- VHDL is better-behaved
- Fewer sources of nondeterminism (e.g., no shared variables)
- VHDL is harder to simulate quickly
- VHDL has fewer built-in facilities for hardware modeling
- VHDL is a much more verbose language
- Most examples don't fit on slides

- *See supplementary document*

Final issues

- Come by my office hours (right after class)
- Any questions or concerns?