Cloud FPGA

bit.ly/cloudfpga
Lecture: Module Hierarchy

Prof. Jakub Szefer
Dept. of Electrical Engineering, Yale University

EENG 428 / ENAS 968
Cloud FPGA
This lecture is mostly based on contents of Chapter 5, from “The Verilog Hardware Description Language” book [1], 5th edition. Example figures and (modified) code are from the textbook unless otherwise specified.

**Topics covered:**

- Module hierarchy
- Port specification
- Parameters
- Generate statements
Module Hierarchy
The structural model of a digital system design uses Verilog module definitions to describe the hierarchy of modules that make up a design.

- A “top” module is the highest-level module.
  - Often contain I/O connections to the external world, i.e., connections to FPGA chip’s physical pins.
- An arbitrary number of sub-modules, sub-sub-modules, etc. can be instantiated.
  - Each module has unique instance name in the hierarchy so it can be referenced.
- Input, output, and inout connections between the modules.

Verilog code needs to define each interface and have proper connections.

Even if signals just pass through a module, interface needs to be defined.
Module Port Specifications

• Ports of a module specify the interface to the external world
• Ports are only means of getting data in and out of a module
  • For simulation, recall that hierarchical names can be used
  • This is no possible for code that is to be synthesized into hardware

• Examples of port specification do’s and don’ts:

```verilog
module wrongPortDefinitions
  (input reg inputA,
   inout reg inoutB);

// ...
endmodule
```

Inputs and inouts can only be wires

```verilog
module port examples
  (input wire inputA,
   inout wire inoutB,
   output wire outputC,
   output reg outputD);

wire someWire;
assign someWire = inputA;

reg someReg;
always @(posedge clock)
someReg <= inputA;

wire moreWiresA;
xor (moreWiresA, inputA, inoutB);

xnor (outputC, inputA, inoutB);

wire moreWiresB;
wire outputControl;
assign inoutB = outputControl ? 1'b1 : 1'b0;
assign moreWiresB = inoutB;

endmodule
```

Inputs can be used in assign statements

Inputs can be used in procedural assignments

Inputs can be used directly as inputs to other sub-modules

Outputs can be assigned directly by sub-modules

Inouts can always be read, but can be set to value or high-impedance Z
Module Port Specifications – inouts

- Inputs and outputs are most common type of ports
- The **inout** ports can be used for tri-state logic: output 0, 1, or Z
  - Multiple outputs connected to same wire
  - Only one input drives the wire, others are high-impedance
    - Else create a short-circuit if two inputs drive the wire
- Examples:
  ```
  bufif1 (outputSignal, inputSignal, controlSignal)
  assign outputSignal = controlSignal ? inputSignal : 1'bz;
  ```
- Should be avoided for FPGA design
  - Danger if logic not designed correctly and two or more signals drive the same wire, or if no signal drives the wire and it’s floating
  - FPGA itself needs to support tri-state logic to realize the design
  - Tools need to recognize the tri-state logic, i.e., support `bufif1`, etc.
Module Port Specifications – FPGA Pins

- The “top” module’s pins are always assigned to some pins on the FPGA chip
- This specification is not part of Verilog, but depends on tools
  - Intel (previously Altera) uses Quartus’ *.qsf configuration file
  - Xilinx uses Vivado’s *.xdc configuration file

- Xilinx example:

```vhs
// top.v
module top
(input switch,
 output led
);

assign led = switch;
endmodule

// top.xdc
set_property IOSTANDARD LVCMOS25 [get_ports switch]
set_property IOSTANDARD LVCMOS33 [get_ports led]
set_property PACKAGE_PIN F22 [get_ports switch]
set_property PACKAGE_PIN T22 [get_ports led]
```

- Set voltage and other I/O standards for the pin
- Match names to what is defined in the Verilog top module
- Location of the pin on the FPGA package
Module Port Connections

• Each module defines the inputs and outputs with unique names and in a specific order

• When instantiating the module, need to specify which signals are connected to which port
  • Specify the connections based on the original order of declaration
  • Or by using named connections

• Verilog 2001 introduced the named connections and they should always be used

```
module binaryToESeg (  
  output eSeg,  
  input A, B, C, D  
);

nand #1  
  g1 (p1, C, ~D),  
  g2 (p2, A, B),  
  g3 (p3, ~B, ~D),  
  g4 (p4, A, C),  
  g5 (eSeg, p1, p2, p3, p4);
endmodule

binaryToESeg disp m1 (eSeg, w3, w2, w1, w0);
binaryToESeg disp m1 (  
  .eSeg(eSeg),  
  .A(w3),  
  .B(w2),  
  .C(w1),  
  .D(w0)  
);
```

Order does not matter anymore, also some signals can be left unconnected if desired, e.g. `.D()`
Parameters

- Parameters allow for specifying values that will be used to control some aspect of the module at compile time, e.g., width of signals.

```
module xor8
  ( output [1:8] xout, 
    input [1:8] xin1, xin2 
  );

  xor (xout[8], xin1[8], xin2[8]), 
    (xout[7], xin1[7], xin2[7]), 
    (xout[6], xin1[6], xin2[6]), 
    (xout[5], xin1[5], xin2[5]), 
    (xout[4], xin1[4], xin2[4]), 
    (xout[3], xin1[3], xin2[3]), 
    (xout[2], xin1[2], xin2[2]), 
    (xout[1], xin1[1], xin2[1]);
endmodule
```

```
module xorx
  #( parameter width = 4, 
    delay = 10
  )
  ( output [1:width] xout, 
    input [1:width] xin1, xin2 
  );

  assign #(delay) xout = xin1 ^ xin2;
endmodule
```

```
module overriddenParameters
  ( output [3:0] a1, a2 
  );

  reg[3:0] b1, c1, b2, c2;
  xorx #(4, 0) a(a1, b1, c1), 
    b(a2, b2, c2);
endmodule
```

```
module xorsAreUs
  ( output [3:0] a1, a2 
  );

  reg[3:0] b1, c1, b2, c2;
  xorx a(a1, b1, c1), 
    b(a2, b2, c2);
endmodule
```

```
module annotate;
  defparam xorsAreUs.b.delay = 5;
endmodule
```

Don't confused #() with delay specification

Range expressions [msb:lsb] specify the width of the signal

This module supports any (integer) width for inputs and outputs and for delays of XOR gate

Overwrite default values for the parameters, now it's 4 bit wide XOR

Can also used names of parameters, e.g., .width()
Array of Instances

- Verilog supports specification of array of instances
  - Each instance in array uses corresponding bits from inputs and outputs
  - Expect to connect from lsb to msb
  - Behavior not clear if the sizes do not match
  - Evaluated at compile time

- Example of array of instances with scalar values clock and clear

  ```verilog
  module regExpanding
  ( output [7:0] Q,
    input [7:0] D,
    input clock, clear
  );
  
  dS r[7:0] (Q, D, clear, clock);

  endmodule
  ```

Connections can become confusing when using arrays of instances, do not recommend to use this feature
Generate Blocks

- Verilog has very support for automating generation of repeated sequences of code
- A more powerful version of arrays of instances
- Generate code blocks at compile time

- Use a for-loop to generate N instances of some code
  - Can use if-then-else and case statements in the generate loop

```verilog
module xorGen
  #( parameter width = 4,
    delay = 10
  )
  ( output [1:width] xout,
    input [1:width] xin1, xin2
  );

  generate
    genvar i;
    for(i = 1; i <= width; i=i+1)
      begin:
        assign #delay xout[i] = xin1[i] ^ xin2[i];
      end
  endgenerate
endmodule
```

Loop control only uses genvars, and any values known at compile-time

Generates...

Generate Example of Adder Module

Based on genvar for-loop generate different code for different genvar value

```verilog
module adderWithConditionCodes
  #(parameter width = 1)
  (output reg [width-1:0] sum,
   output reg cOut, neg, overflow,
   input [width-1:0] a, b,
   input cin)
);

  reg [width -1:0] c;

  generate
    genvar i;
    for (i = 0; i <= width-1; i=i+1) begin: stage
      case(i)
        0: begin
          always @(posedge begin
            sum[i] = a[i] ^ b[i] ^ cin;
            c[i] = a[i]&b[i] | b[i]&c[i] | a[i] & c[i];
          end
        end
        width-1: begin
          always @(posedge begin
            sum[i] = a[i] ^ b[i] ^ c[i-1];
            cOut = a[i]&b[i] | b[i]&c[i-1] | a[i] & c[i-1];
            neg = sum[i];
            overflow = cOut ^ c[i-1];
          end
        end
        default: begin
          always @(posedge begin
            sum[i] = a[i] ^ b[i] ^ c[i-1];
            c[i] = a[i]&b[i] | b[i]&c[i-1] | a[i] & c[i-1];
          end
        end
      endcase
    end
  endgenerate
endmodule

module testbench;

  localparam width = 4;

  wire [width-1:0] sum;
  wire cOut, neg, overflow;
  reg [width-1:0] a, b;

  adderWithConditionCodes
  @(width(width)) addit (sum, cOut, neg, overflow, a, b, 1'b0);

  initial begin
    $monitor(a,b,sum,a+b);
    a = 0;
    b = 0;
    #10;
    while (!overflow)
      begin
        a = a + 1;
        #10;
        b = 0;
        #10;
      end
    end
  end
endmodule
```
References