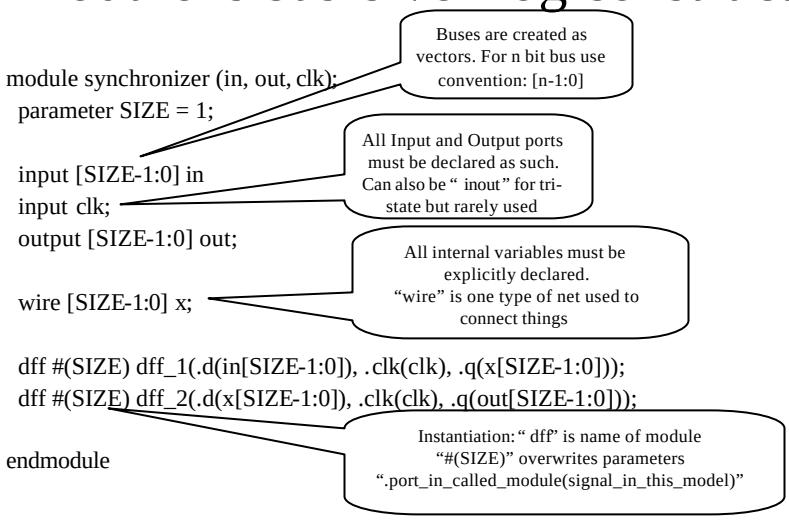
Lecture #2: Verilog HDL

Paul Hartke
Phartke@stanford.edu
Stanford EE183
April 8, 2002

EE183 Design Process

- Understand problem and generate block diagram of solution
- Code block diagram in verilog HDL
- Synthesize verilog
- Create verification script to test design
- Run static timing tool to make sure timing is met
- Design is mapped, placed, routed, and *.bit file is created download to FPGA

Module is basic verilog construct



Lexical Conventions

- The lexical conventions are close to the programming language C++.
- Comments are designated by // to the end of a line or by /* to */ across several lines.
- Keywords, e. g., module, are reserved and in all lower case letters.
- The language is case sensitive, meaning upper and lower case letters are different.
- Spaces are important in that they delimit tokens in the language.

Number specification

- 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>
    - where <size> contains decimal digits that specify the size of the constant in the number of bits. The <size> is optional. The <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. The <number> part contains digits which are legal for the <base format>. Some examples:
  - 4'b0011 // 4-bit binary number 0011
  - 5'd3 // 5-bit decimal number
  - 32'hdeadbeef // 32 bit hexadecimal number

Bitwise/Logical Operators

- Bitwise operators operate on the bits of the operand or operands.
    - For example, the result of A & B is the AND of each corresponding bit of A with B. Operating on an unknown (x) bit results in the expected value. For example, the AND of an x with a FALSE is an FALSE. The OR of an x with a TRUE is a TRUE.
  - Operator Name
  - ~ Bitwise negation
  - & Bitwise AND
  - | Bitwise OR
  - ^ Bitwise XOR
  - ~& Bitwise NAND
  - ~| Bitwise NOR
  - ~^ or ^~ Equivalence (Bitwise NOT XOR)

Miscellaneous Operators

- { , } Concatenation of nets
  - Joins bits together with 2 or more comma-separated expressions, e, g. {A[0], B[1:7]} concatenates the zeroth bit of A to bits 1 to 7 of B.
- << Shift left (Multiplication by power of 2)
  - Vacated bit positions are filled with zeros, e. g., A = A << 2; shifts A two bits to left with zero fill.
- >> Shift right (Division by power of 2)
  - Vacated bit positions are filled with zeros.
- ?: Conditional (Creates a MUX)
  - Assigns one of two values depending on the conditional expression. E.g., A = C > D ? B+3 : B-2; means if C greater than D, the value of A is B+3 otherwise B-2.

Unary Reduction Operators

- Unary reduction operators produce a single bit result from applying the operator to all of the bits of the operand. For example, &A will AND all the bits of A.
  - Operator Name
  - & AND reduction
  - | OR reduction
  - ^ XOR reduction
  - ~& NAND reduction
  - ~| NOR reduction
  - ~^ XNOR reduction
- I have never used these, if you find a realistic application, let me know… ☺

Continuous Assignment

- assign out = in1 & in2;
  - Amazingly enough creates an “and” gate!
  - Anytime right hand side (RHS) changes, left hand side (LHS) is updated
  - LHS must be a “wire”

Procedural Assignments

- We will only use them to define combinational logic
  - as a result, blocking (=) and nonblocking assignment (<=) are the same
- Example:

```
reg out;
always @(in1 or in2)
begin
  out = in1 & in2;
end
```

LHS must be of type reg
Does ~~***NOT***~~ ~~mean this is a DFF~~

All input signals must be in sensitivity list

Begin and End define a block in Verilog

If-Else Conditional Procedural Assignment

- Just a combinational logic mux
- Every if must have matching else or state element will be inferred—why?

```
always @(control or in)
begin
  if (control == 1'b1)
    out = in;
 end
```

- Watch nestings—make life easy, always use begin…end

Logical Operators

- Logical operators operate on logical operands and return a logical value, i. e., TRUE(1) or FALSE(0).
  - Used typically in if and while statements.
    - Do not confuse logical operators with the bitwise Boolean operators. For example , ! is a logical NOT and ~ is a bitwise NOT. The first negates, e. g., !(5 == 6) is TRUE. The second complements the bits, e. g., ~{1,0,1,1} is 0100.
  - Operator Name
  - ! Logical negation
  - && Logical AND
  - || Logical OR

Relational Operators

- Relational operators compare two operands and return a logical value, i. e., TRUE(1) or FALSE(0)—what do these synthesize into?
  - If any bit is unknown, the relation is ambiguous and the result is unknown – should never happen!

Operator	Name
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Logical equality
!=	Logical inequality

Case Statement Procedural Assignment

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
  output out;
  input i0, i1, i2, i3;
  input s1, s0;
  reg out;
  always @(s1 or s0 or i0 or i1 or i2 or i3)
    begin
      case ({s1, s0})
        2'b00: out = i0;
        2'b01: out = i1;
        2'b10: out = i2;
        2'b11: out = i3;
        default: out = 1'bx;
      endcase
    end
endmodule
```

Note how all nets that are inputs to the always block are specified in the sensitivity list

Make sure all 2^n cases are covered or include a "default:" statement or else state elements will be inferred

X is don't care
After initial synchronous reset there should ***never*** be any X's in your design

So how do I get D-FlipFlops?

- Use 183lib.v to instantiate them
  - dff, dffr, dffre
- These are the ***only*** state elements (except for CoreGen RAMs) allowed in your design

Dffre guts

```
// dffre: D flip-flop with active high enable and reset
// Parametrized width; default of 1
module dffre (d, en, r, clk, q);
  parameter WIDTH = 1;
  input en;
  input r;
  input clk;
  input [WIDTH-1:0] d;
  output [WIDTH-1:0] q;
  reg [WIDTH-1:0] q;
  always @ (posedge clk)
  if ( r )
    q <= {WIDTH{1'b0}};
  else if (en)
    q <= d;
  else q <= q;
endmodule
```

Only change LHS on " posedge clk"
Note that if statement is missing an else

Replicator Operator.
How cute!! ☺

No Behavioral Code

- No “initial” statements
  - Often used to reset/initialize design
- No system tasks
  - “$” commands (ie, “$display()”)
- For both, use Xilinx simulator and scripts

Use Case Statement for FSM

- Instantiate state elements as dffX
- Put next state logic in always @() block
  - Input is curstate (.q of dffX) and other inputs
  - Output is nextstate which goes to .d of dffX
  - Use combined case and if statements
    - “If” good for synchronous resets and enables
- Synthesis tools auto-magically minimizes all combinational logic.
  - Three cheers for synthesis!! ☺

8-bit Counter

```
module counter_8 (clk, reset, en, cntr_q);
  input clk;
  input reset;
  input en;
  output [7:0] cntr_q;

  reg [7:0] cntr_d;
  wire [7:0] cntr_q;

  // Counter next state logic
  always @(cntr_q)
  begin
      cntr_d = cntr_q + 8'b1;
  end

  // Counter state elements
  dffre #(8) cntr_reg (.clk(clk), .r(reset), .en(en), .d(cntr_d), .q(cntr_q));

Endmodule
```

CoreGen

- Tools → Design Entry → Core Generator
  - Useful info appears in "language assistant"—Read it!
- Only use this for memories for now
  - Do you need anything else??
    - I really cannot think of anything now
- Caveat: Block Memory does not simulate correctly with initial values.
  - Must create gate netlist by completing synthesis and implementation.
  - Simulate by loading time_sim.edn into Simulator