Lecture #2: Verilog HDL

Kunle Olukotun
Stanford EE183
January 10, 2003

Why Verilog?

• Why use an HDL?
  – Describe complex designs (millions of gates)
  – Input to synthesis tools (synthesizable subset)
  – Design exploration with simulation

• Why not use a general purpose language
  – Support for structure and instantiation (objects?)
  – Support for describing bit-level behavior
  – Support for timing
  – Support for concurrency

• Verilog vs. VHDL
  – Verilog is relatively simple and close to C
  – VHDL is complex and close to Ada
  – Verilog has 60% of the world digital design market (larger share in US)

• Verilog modeling range
  – From gates to processor level
  – We’ll focus on RTL (register transfer level)
EE183 Design Process

- Understand problem and generate block diagram of solution (datapath control decomposition)
- Code block diagram in verilog
- 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

Event Driven Simulation

- Verilog is really a language for modeling event-driven systems
  - Event: change in state
    - Simulation starts at $t = 0$
    - Processing events generates new events
    - When all events at time $t$ have been processed simulation time advances to $t + 1$
    - Simulation stops when there are no more events in the queue
Modeling Structure: Modules

- The module is the basic building block in Verilog
  - Modules can be interconnected to describe the structure of your digital system
  - Modules start with keyword `module` and end with keyword `endmodule`

```
Module AND <port list>
  .
  .
endmodule
```

```
Module CPU <port list>
  .
  .
endmodule
```

Modeling Structure: Ports

- Module Ports
  - Similar to pins on a chip
  - Provide a way to communicate with outside world
  - Ports can be `input`, `output` or `inout`

```
Module AND (i0, i1, o);
  input i0, i1;
  output o;
  .
  .
endmodule
```
Modeling Structure

- Module instances
  - Verilog models consist of a hierarchy of module instances
  - In C++ speak: modules are classes and instances are objects

Logic Values

- 0: zero, logic low, false, ground
- 1: one, logic high, power
- X: unknown
- Z: high impedance, unconnected, tri-state
Data Types

• Nets
  – Nets are physical connections between devices
  – Nets always reflect the logic value of the driving device
  – Many types of nets, but all we care about is wire

• Registers
  – Implicit storage – unless variable of this type is modified it retains previously assigned value
  – Does not necessarily imply a hardware register
  – Register type is denoted by reg
  – int is also used

Variable Declaration

• Declaring a net
  wire [<range>] <net_name> [<net_name>*];
  Range is specified as [MSb:LSb]. Default is one bit wide

• Declaring a register
  reg [<range>] <reg_name> [<reg_name>*];

• Declaring memory
  reg [<range>] <memory_name> [<start_addr>: <end_addr>];

• Examples
  reg r; // 1-bit reg variable
  wire w1, w2; // 2 1-bit wire variable
  reg [7:0] vreg; // 8-bit register
  reg [7:0] memory [0:1023]; a 1 KB memory
Ports and Data Types

- Correct data types for ports

Module synchronizer (in, out, clk);
parameter SIZE = 1;

input [SIZE-1:0] in
input clk;
output [SIZE-1:0] out;
wire [SIZE-1:0] x;

dff #(SIZE) dff_1(.d(in[SIZE-1:0]), .clk(clk), .q(x[SIZE-1:0]));
dff #(SIZE) dff_2(.d(x[SIZE-1:0]), .clk(clk), .q(out[SIZE-1:0]));
endmodule

Example Module

Buses are created as vectors. For n bit bus use convention: [n-1:0]

All Input and Output ports must be declared as such. Can also be “inout” for tri-state but rarely used

All internal variables must be explicitly declared. “wire” is one type of net used to connect things

Instantiation: “dff” is name of module
“#(SIZE)” overwrites parameters
“port_in_called_module(signal_in_this_model)”
Modeling Behavior

• Behavioral Modeling
  Describes functionality of a module

• Module Behavior
  Collection of concurrent processes
  1. Continuous assignments
  2. Initial blocks
  3. Always blocks

Verilog Operators

Arithmetic: +, =, *,/, %
Binary bitwise: ~, &, |, ^, ~^,
Unary reduction: &, ~&, |, ~|, ^, ~^
Logical: !, &&, ||, ==, ===, !=, !==
    == returns x if any of the input bits is x
    or z
    === compares xs and zs
Relational: <, >, <=, >=
Logical shift: >>, <<
Conditional: ?:
Concatenation: {}
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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Bitwise negation</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td>~&amp;</td>
<td>Bitwise NAND</td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td><del>^ or ^</del></td>
<td>Equivalence</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>AND reduction</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>XOR reduction</td>
</tr>
<tr>
<td>~&amp;</td>
<td>NAND reduction</td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td>~^</td>
<td>XNOR reduction</td>
</tr>
</tbody>
</table>

- I have never used these, if you find a realistic application, let me know… 😊

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!

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>==</td>
<td>Logical equality</td>
</tr>
<tr>
<td>!=</td>
<td>Logical inequality</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Logical negation</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Continuous Assignment

Continually drive wire variables
Used to model combinational logic or make connections between wires

```verbatim
Module half_adder(x, y, s, c)
  input x, y;
  output s, c;
  assign s = x ^ y;
  assign c = x & y;
endmodule
```

```verbatim
Module adder_4(a, b, ci, s, co)
  input [3:0] a, b;
  input ci;
  output [3:0] s;
  output co;
  assign {co, s} = a + b + ci;
endmodule
```

- Anytime right hand side (RHS) changes, left hand side (LHS) is updated
- LHS must be a “net”
Initial and Always

• Multiple statements per block
  Procedural assignments
  Timing control
  control

• Initial blocks execute once
  • at \( t = 0 \)

• Always blocks execute continuously
  • at \( t = 0 \) and repeatedly thereafter

```
initial
begin
  .
  .
  .
end

always
begin
  .
  .
  .
end
```

Procedural assignments

• Blocking assignment =
  Regular assignment inside procedural block
  Assignment takes place immediately
  LHS must be a register

```
always
begin
  A = B;
  B = A;
end
```

• Nonblocking assignment <=
  Compute RHS
  Assignment takes place at end of block
  LHS must be a register

```
always
begin
  A <= B;
  B <= A;
end
```

`swap A and B`
Using Procedural Assignments

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

- Example:
  ```verilog
  reg out;
  always @(in1 or in2)
  begin
    out = in1 & in2;
  end
  ```

If-Else Conditional

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

  ```verilog
  always @(control or in1 or in2)
  begin
    if (control == 1'b1) begin
      out = in1;
    end
    else begin
      out = in2;
    end
  end
  ```

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

Loop Statements

• Repeat

```verbatim
i = 0;
repeat (10)
begin
  i = i + 1;
  $display("i = %d", i);
end
```

• While

```verbatim
i = 0;
while (i < 10)
begin
  i = i + 1;
  $display("i = %d", i);
end
```

• For

```verbatim
for (i = 0; i < 10; i = i + 1)
begin
  i = i + 1;
  $display("i = %d", i);
end
```
Verilog Coding Rules

- Coding rules eliminate strange simulation behavior
  - When modeling sequential logic, use nonblocking assignments
  - When modeling combinational logic with `always` block, use blocking assignments. Make sure all RHS variables in block appear in `@` expression
  - If you mix sequential and combinational logic within the same `always` block use nonblocking assignments
  - Don’t mix blocking and nonblocking assignments in the same `always` block

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

Monday Jan 13

• Lab project #1
• The Game of Life