Hardware Description Languages: Verilog

- Verilog
  - Structural Models
  - (Combinational) Behavioral Models
  - Syntax
  - Examples

Quick History of HDLs

- ISP (circa 1977) - research project at CMU
  - Simulation, but no synthesis
- Abel (circa 1983) - developed by Data-I/O
  - Targeted to programmable logic devices
  - Not good for much more than state machines
- Verilog (circa 1985) - developed by Gateway (now Cadence)
  - Similar to Pascal and C
  - Delays is only interaction with simulator
  - Fairly efficient and easy to write
  - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
  - Similar to Ada (emphasis on re-use and maintainability)
  - Simulation semantics visible
  - Very general but verbose
  - IEEE standard

Design Methodology

- Structure and Function (Behavior) of a Design
- HDL Specification
  - Simulation
  - Synthesis
  - Verification: Design Behave as Required?
  - Functional: I/O Behavior
  - Register-Level (Architectural)
  - Logic-Level (Gates)
  - Transistor-Level (Electrical)
  - Timing: Waveform Behavior
  - Generation: Map Specification to Implementation

Verilog/VHDL

- The "standard" languages
- Very similar
  - Many tools provide front-ends to both
  - Verilog is "simpler"
  - Less syntax, fewer constructs
  - VHDL supports large, complex systems
  - Better support for modularization
  - More grungy details
  - "Hello world" is much bigger in VHDL

Verilog

- Supports structural and behavioral descriptions
- Structural
  - Explicit structure of the circuit
  - How a module is composed as an interconnection of more primitive modules/components
  - E.g., each logic gate instantiated and connected to others
- Behavioral
  - Program describes input/output behavior of circuit
  - Many structural implementations could have same behavior
  - E.g., different implementations of one Boolean function

Verilog Introduction

- the module describes a component in the circuit
- Two ways to describe:
  - Structural Verilog
    - List of components and how they are connected
    - Just like schematics, but using text
    - Hard to write, hard to decode
    - Useful if you don't have integrated design tools
  - Behavioral Verilog
    - Describe what a component does, not how it does it
    - Synthesized into a circuit that has this behavior
### Structural Model

- **Composition of primitive gates to form more complex module**
- **Note use of wire declaration**

```verbatim
module xor_gate (out, a, b);
    input a, b;
    output out;
    wire abar, bbar, tl, t2;
    inverter invA (abar, a);
    inverter invB (bbar, b);
    and_gate and1 (tl, a, bbar);
    and_gate and2 (t2, b, abar);
    or_gate or1 (out, tl, t2);
endmodule
```

### Simple Behavioral Model

- **Combinational logic**
  - **Describe output as a function of inputs**
  - **Note use of assign keyword: continuous assignment**

```verbatim
module and_gate (out, in1, in2);
    input in1, in2;
    output out;
    assign out = in1 & in2;
endmodule
```

### Verilog Numbers

- **14** - ordinary decimal number
- **-14** - 2's complement representation
- **12'b0000_0100_0110** - binary number with 12 bits (_ is ignored)
- **12'h046** - hexadecimal number with 12 bits

### Verilog Data Types and Values

- **Bits - value on a wire**
  - 0, 1
  - X - don't care/don't know
  - Z - undriven, tri-state

- **Vectors of bits**
  - Treated as an unsigned integer value if A < 0?
  - Concatenating bits/operands into a vector
    - e.g., sign extend
  - Style: Use a[7:0] = b[7:0] + c;
    - Not: a = b + c; // need to look at declaration

### Verilog Operators

<table>
<thead>
<tr>
<th>Verilog Operator</th>
<th>Name</th>
<th>Priority Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>add</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>&amp;</td>
<td>and</td>
<td>Boolean</td>
</tr>
<tr>
<td>^</td>
<td>xnor</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift_left</td>
<td>Shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift_right</td>
<td>Shift</td>
</tr>
<tr>
<td>.</td>
<td>dot</td>
<td>Selector</td>
</tr>
<tr>
<td>==</td>
<td>compare</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less_or_equal</td>
<td>Relational</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater_or_equal</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;</td>
<td>less_than</td>
<td>Relational</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater_than</td>
<td>Relational</td>
</tr>
<tr>
<td>!=</td>
<td>not_equal</td>
<td>Relational</td>
</tr>
<tr>
<td>[</td>
<td>index</td>
<td>Range</td>
</tr>
<tr>
<td>(</td>
<td>start</td>
<td>Range</td>
</tr>
<tr>
<td>)</td>
<td>end</td>
<td>Range</td>
</tr>
<tr>
<td>.</td>
<td>property</td>
<td>Attribute</td>
</tr>
<tr>
<td>.</td>
<td>method</td>
<td>Method</td>
</tr>
<tr>
<td>.</td>
<td>function</td>
<td>Method</td>
</tr>
<tr>
<td>.</td>
<td>stream</td>
<td>Stream</td>
</tr>
<tr>
<td>.</td>
<td>object</td>
<td>Object</td>
</tr>
<tr>
<td>.</td>
<td>class</td>
<td>Class</td>
</tr>
</tbody>
</table>
Verilog "Variables"

- wire
  - Variable used simply to connect components together
- reg
  - Variable that saves a value as part of a behavioral description
  - Usually corresponds to a wire in the circuit
  - Is NOT necessarily a register in the circuit
- The Rule:
  - Declare a variable as a reg if it is the target of a concurrent (non-blocking) assignment statement
  - Don’t confuse reg assignments with the combinational continuous assign statement
  - Reg should only be used with always blocks (sequential logic, to be presented ...)
  - Confusing isn’t it?

Verilog Continuous Assignment

- Assignment is continuously evaluated
- assign corresponds to a connection or a simple component with the described function
- Target is NEVER a reg variable

use of Boolean operators
(- for bit-wise, | for logical negation)
assign A = X | (Y & ~Z);
assign B[3:0] = 4’b01XX;
assign C[15:0] = 16’h00FF;

use of arithmetic operator
multiple assignment (concatenation)
delay of performing computation, only used by simulator, not synthesis

Verilog Module

- Corresponds to a circuit component
- "Parameter list" is the list of external connections, aka "ports"
- Ports are declared "input", "output" or "inout"
- input ports used on tri-state buses
- Port declarations imply that the variables are wires

Comparator Example

module Compare1 (A, B, Equal, Alarger, Blarger);
input A, B;
output Equal, Alarger, Blarger;
assign Equal = (A & ~B) | (~A & ~B);
assign Alarger = (A & ~B) & (~A & ~B);
assign Blarger = (~A & ~B) & (A & ~B);
endmodule

module Compare2 (A4, B4, Equal, Alarger, Blarger);
input [3:0] A4, B4;
output Equal, Alarger, Blarger;
wire e0, e1, e2, e3, A10, A11, A12, A13, B10, B11, B12, B13;
Compare1 cp0(A4[0], B4[0], e0, A10, B10);
Compare1 cp1(A4[1], B4[1], e1, A11, B11);
Compare1 cp2(A4[2], B4[2], e2, A12, B12);
Compare1 cp3(A4[3], B4[3], e3, A13, B13);
assign Equal = (e0 & e1 & e2 & e3);
assign Alarger = (A13 | A12 & e3) | (A11 & e3 & e2) | (A10 & e3 & e2 & e1);
assign Blarger = (~Alarger & ~Equal);
endmodule

Announcements

- Lecture room change EFFECTIVE 1 FEB 07:
  - Beware of what you ask for!
  - 159 Mulford Hall (near West Gate/Oxford Street)

Card Key Access to 125 Cory

- You can’t get it until 7 February
- For access:
  - EECS Keys/Cardkeys, Copy Cards Assistant: Loretta Lutcher
  - 253 Cory, 642-1572, loret@eecs
  - Loretta issues keys and electronic cardkeys for Cory Hall. Handles cardkey problems. She can add 125 Cory to your Cal Card.
Simple Behavioral Model: the always block

- **always block**
  - Always waiting for a change to a trigger signal
  - Then executes the body

```verilog
module and_gate (out, in1, in2);
  input in1, in2;
  output reg out;
  always @ (in1 or in2)
  begin
    out = in1 & in2;
  end
endmodule
```

```
CS 150 - Spring 2007 - Lecture #4: Verilog - 19
```

always Block

- **Procedure that describes the function of a circuit**
  - Can contain many statements including if, for, while, case
  - Statements in the always block are executed sequentially
  - (Continuous assignments <= are executed in parallel)
  - Entire block is executed at once
  - **Final result** describes the function of the circuit for current set of inputs
  - intermediate assignments don’t matter, only the final result

```
begin/end used to group statements
```

Complete” Assignments

- If an always block executes, and a variable is **not** assigned
  - Variable keeps its old value (think implicit state)
  - NOT combinational logic ==> latch is inserted (implied memory)

This is usually not what you want: dangerous for the novice!

- Any variable assigned in an always block should be assigned for any (and every!) execution of the block

```
module and_gate (out, in1, in2);
  input in1, in2;
  output out;
  reg out;
  always @ (in1 or in2)
  begin
    out = in1 & in2;
  end
endmodule
```

Incomplete Triggers

- Leaving out an input trigger usually results in a sequential circuit

**Example:** Output of this "and" gate depends on the input history

```verilog
module and_gate (out, in1, in2);
  input in1, in2;
  output reg out;
  always @ (in1)
  begin
    out = in1 & in2;
  end
endmodule
```

```
CS 150 - Spring 2007 - Lecture #4: Verilog - 21
```

Verilog if

- Same as C if statement

```verilog
// Simple 4:1 mux
module mux4 (sel, A, B, C, D, Y);
  input [1:0] sel; // 2-bit control signal
  input A, B, C, D;
  output Y;
  reg Y; // target of assignment
  always @(sel or A or B or C or D)
    if (sel == 2’b00) Y = A;
    else if (sel == 2’b01) Y = B;
    else if (sel == 2’b10) Y = C;
    else if (sel == 2’b11) Y = D;
endmodule
```

```
CS 150 - Spring 2007 - Lecture #4: Verilog - 22
```

```
CS 150 - Spring 2007 - Lecture #4: Verilog - 23
```

Verilog if

- Another way

```verilog
// Simple 4:1 mux
module mux4 (sel, A, B, C, D, Y);
  input [1:0] sel; // 2-bit control signal
  input A, B, C, D;
  output Y;
  reg Y; // target of assignment
  always @(sel or A or B or C or D)
    if (sel[0] == 0)
      if (sel[1] == 0) Y = A;
      else Y = B;
    else
      if (sel[1] == 0) Y = C;
      else Y = D;
endmodule
```

```
CS 150 - Spring 2007 - Lecture #4: Verilog - 24
```
Verilog case

- Sequential execution of cases
  - Only first case that matches is executed (implicit break)
  - Default case can be used

```verilog
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel; // 2-bit control signal
input A, B, C, D;
output Y;      // target of assignment
reg Y;
always @ (sel or A or B or C or D)
case (sel)
  2'b00: Y = A;
  2'b01: Y = B;
  2'b10: Y = C;
  2'b11: Y = D;
endcase
endmodule
```

- Without the default case, this example would create a latch for Y
- Assigning X to a variable means synthesis is free to assign any value

```verilog
// Simple binary encoder (input is 1-hot)
module encode (A, Y);
input [7:0] A; // 8-bit input vector
output [2:0] Y; // 3-bit encoded output
reg [2:0] Y; // target of assignment
always @(A)
case (A)
  8'b00000000: Y = 0;
  8'b00000001: Y = 1;
  8'b00000010: Y = 2;
  8'b00000100: Y = 3;
  8'b00001000: Y = 4;
  8'b00010000: Y = 5;
  8'b00100000: Y = 6;
  8'b01000000: Y = 7;
default: Y = 3'bX; // Don’t care when input is not 1-hot
endcase
endmodule
```

Parallel Case

- A priority encoder is more expensive than a simple encoder
  - If we know the input is 1-hot, we can tell the synthesis tools
    - “parallel-case” pragma says the order of cases does not matter

```verilog
// simple encoder
module encode (A, Y);
input [7:0] A; // 8-bit input vector
output [2:0] Y; // 3-bit encoded output
reg [2:0] Y; // target of assignment
always @(A)
case (A)
  8'h1'b1: // synthesis parallel-case
    A[0]: Y = 0;
    A[1]: Y = 1;
    A[2]: Y = 2;
    A[3]: Y = 3;
    A[4]: Y = 4;
    A[5]: Y = 5;
    A[6]: Y = 6;
    A[7]: Y = 7;
  default: Y = 3'bX; // Don’t care when input is all 0’s
endcase
endmodule
```

Verilog case (cont)

- Cases are executed sequentially
  - Following implements a priority encoder

```verilog
// Priority encoder
module encode (A, Y);
input [7:0] A; // 8-bit input vector
output [2:0] Y; // 3-bit encoded output
reg [2:0] Y; // target of assignment
always @(A)
case (A)
  8'h1'b0: Y = B;
  8'h1'b1: Y = C;
  default: Y = 3'bX; // Don’t care when input is all 0’s
endcase
endmodule
```

Verilog casex

- Like case, but cases can include ‘X’
  - X bits not used when evaluating the cases
  - In other words, you don’t care about those bits!
case Example

module encode (A, valid, Y);
input A;
input [7:0] X;
output valid;
output [2:0] Y;
reg [7:0] E;
reg valid;

always @(A) begin
    valid = 1;
    case (A)
        0: E = 0;
        1: E = 1;
        2: E = 2;
        3: E = 3;
        4: E = 4;
        5: E = 5;
        6: E = 6;
        7: E = 7;
    endcase
    valid = 0;
end
endmodule

Another Behavioral Example

Computing Conway's Game of Life rule

Cell with no neighbors or 4 neighbors dies; with 2-3 neighbors lives.

module life (neighbors, self, out);
    input self;
    input [7:0] neighbors;
    output out;
    reg [2:0] Y;
    reg valid;
    always @ (neighbors or self)
        begin
            if (i = 0; i <= 8; i = i+1) count = count + neighbors[i];
            out = 0;
            out = out | (count == 3);
            out = out | (self == 1) & (count == 2);
        end
endmodule

full-case and parallel-case

// synthesis parallel_case
- Tells compiler that ordering of cases is not important
- That is, cases do not overlap
  - e.g., state machine - can’t be in multiple states
  - Gives cheaper implementation

// synthesis full_case
- Tells compiler that cases left out can be treated as don’t cares
- Avoids incomplete specification and resulting latches