Hardware Description Languages: Verilog

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

Quick History of HDLs

- ISP (circa 1977) - research project at CMU
  - Simulation, no synthesis
- Abel (circa 1983) - developed by Data-I/O
  - Targeted at programmable logic devices
  - Not good for much more than state machines
- Verilog (circa 1985) - developed by Gateway (now Cadence)
  - Similar to Pascal (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

- HDL Specification
  - Structural and Functional
    - Specification
  - Simulation
  - Synthesis
    - 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 larger, complex systems
            - Better support for modularization
            - More gritty details
            - "Hello World" much bigger in VHDL

Verilog

- Supports structural and behavioral descriptions
  - Structural
    - Explicit structure of the circuit
  - 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 read
      - Useful if you don't have integrated design tools
    - Behavioral Verilog
      - Describe what a component does, not how it does it
      - Synthesizable into a circuit that has that behavior
module xor_gate (out, a, b);
input     a, b;
output    out;
wire     abar, bbar, t1, t2;
inverter invA (abar, a);
inverter invB (bbar, b);
and_gate and1 (t1, a, bbar);
and_gate and2 (t2, b, abar);
or_gate  or1 (out, t1, t2);
endmodule

module full_adder (A, B, Cin, S, Cout);
input     A, B, Cin;
output    S, Cout;
assign {Cout, S} = A + B + Cin;
endmodule

module adder4 (A, B, Cin, S, Cout);
input  [3:0] A, B;
input  Cin;
output [3:0] S;
output  Cout;
wire    C1, C2, C3;
full_addr fa0 (A[0], B[0], Cin, S[0], C1);
full_addr fa1 (A[1], B[1], C1,  S[1], C2);
full_addr fa2 (A[2], B[2], C2,  S[2], C3);
full_addr fa3 (A[3], B[3], C3,  S[3], Cout);
endmodule

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

Verilog Data Types and Values

- Bits - value on a wire
  - 0, 1
  - X - don't care
  - Z - undriven, tri-state
- Vectors of bits
  - Treated as an unsigned integer value
    - e.g. A < 0
  - Concatenating bits/vectors into a vector
    - e.g. sign extend
      - S[7:0] = {3{A[3]}, A[3:0]};
  - Style: Use a[7:0] = b[7:0] + c;
  - Need: a = b = c;

Verilog Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Numbered Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
<td>3</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
<td>4</td>
</tr>
<tr>
<td>^</td>
<td>Power</td>
<td>5</td>
</tr>
<tr>
<td>&amp;</td>
<td>Logical AND</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Logical OR</td>
</tr>
<tr>
<td>~</td>
<td>Logical NOT</td>
<td>8</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
<td>9</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>10</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
<td>11</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>12</td>
</tr>
</tbody>
</table>

Verilog Numbers

- 34 - ordinary decimal number
- -14 - 2's complement representation
- 11000000.0100.0110 - binary number with 12 bits (not signed)
- 3'B046 - hexadecimal number with 12 bits

Verilog values are unsigned
- if A = 0110 (6) and B = 1010 (-6)
  - C = 10000 not 00000
  - i.e. 3 is zero-padded, not sign-extended
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 an assignment statement.
  - Don’t confuse reg assignments with the combinational assign keyword.
  - E.g., should not be used with always blocks (sequential logic, to be predefined).
  - Confusing isn’t it?

Verilog Module

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

module full_addr (A, B, Cin, S, Cout);
  input A, B, Cin;
  output S, Cout;
  assign (Cout, S) = A + B + Cin;
endmodule

Comparator Example

// Make a 4-bit comparator from 4 1-bit comparators
module Compare4 (A4, B4, Equal, Alarger, Blarger);
  input [3:0] A4, B4;
  output Equal, Alarger, Blarger;
  wire e0, e1, e2, e3, Al0, Al1, Al2, Al3, Bl1, Bl2, Bl3;
  Compare1 cp0(A4[0], B4[0], e0, Al0, Bl0);
  Compare1 cp1(A4[1], B4[1], e1, Al1, Bl1);
  Compare1 cp2(A4[2], B4[2], e2, Al2, Bl2);
  Compare1 cp3(A4[3], B4[3], e3, Al3, Bl3);
  assign Equal = (e0 & e1 & e2 & e3);
  assign Alarger = (Al3 | (Al2 & e3) | (Al1 & e3 & e2) | (Al0 & e3 & e2 & e1));
  assign Blarger = (~Alarger & ~Equal);
endmodule

Simple Behavioral Model - the always block

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

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

- A 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)
- The entire block is executed at once
- The 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 execution of the block

Incomplete Triggers

- Leaving out an input trigger usually results in a sequential circuit
- Example: The output of this “and” gate depends on the input history

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

Verilog if

- Another way

```
// 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
```

Verilog case

- Sequential execution of cases
  - Only first case that matches is executed (no break)
  - Default case can be used

```
// 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)
  case (sel)
    2'b00: Y = A;
    2'b01: Y = B;
    2'b10: Y = C;
    2'b11: Y = D;
  endcase
endmodule
```
Verilog case

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.

// 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'b00000001: Y = 0;
 8'b00000010: Y = 1;
 8'b00000100: Y = 2;
 8'b00001000: Y = 3;
 8'b00010000: Y = 4;
 8'b00100000: Y = 5;
 8'b01000000: Y = 6;
 8'b10000000: Y = 7;
default: Y = 3'bX; // Don't care when input is not 1-hot
endcase
endmodule

Verilog case (cont)

Cases are executed sequentially.

// 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 (1'b1)
  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

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.

// 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 (1'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 casex

Like case, but cases can include 'X'.
X bits not used when evaluating the cases.

// Priority encoder
module encode (A, valid, Y);
input [7:0] A; // 8-bit input vector
output [2:0] Y; // 3-bit encoded output
output valid; // Asserted when an input is not all 0's
reg [2:0] Y; // target of assignment
reg valid;
always @(A) begin
  valid = 1;
  casex (A)
    8'bXXXXXXX1: Y = 0;
    8'bXXXXXX10: Y = 1;
    8'bXXXXX100: Y = 2;
    8'bXXXX1000: Y = 3;
    8'bXXX10000: Y = 4;
    8'bXX100000: Y = 5;
    8'bX1000000: Y = 6;
    8'b10000000: Y = 7;
  default: begin
    valid = 0;
    Y = 3'bX; // Don't care when input is all 0's
  end
endcase
end
endmodule

Verilog for

for is similar to C.
for statement is executed at compile time (like macro expansion).
Result is whatever matters, not how results are calculated.

// 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
integer i; // Temporary variables for program only
always @(A) begin
  test = 8b'00000001;
  Y = 3'bX;
  for (i = 0; i < 8; i = i + 1) begin
    if (A == test) Y = 3'bX;
    test = test << 1;
  end
  end
end
endmodule
Another Behavioral Example

```
module life (neighbors, self, out);
input         self;
input [7:0]   neighbors;
output        out;
reg           out;
integer       count;
integer       i;
always @(neighbors or self) begin
  count = 0;
  for (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
```

Verilog while/repeat/forever

```
while (expression) statement
  Execute statement while expression is true
repeat (expression) statement
  Execute statement a fixed number of times
forever statement
  Execute statement forever
```

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
```