Hardware Description Languages:
Verilog
- Verilog
  - Structural Models
  - (Combinationl) Behavioral Models
  - Syntax
  - Examples

Quick History of HDLs
- ISP (circa 1977) - research project at CMU
  - Simulation, but no synthesis
- Abel (circa 1983) - developed by Dato-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

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
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

Structural Model

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

```verilog
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
```

Simple Behavioral Model

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

```verilog
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/don't know
  - Z - undriven, tri-state
- Vectors of bits
  - Treated as an unsigned integer value
  - E.g., A[3:0] = 0b1010
  - Concatenating bits/vectors into a vector
    - E.g., sign extend
    - S[7:0] = 4{A[3:0], A[3:0]:3}
    - Style: Use `A[7:0] = b[7:0] + c`;
      Not: `a = b + c` // need to look at declaration
- Verilog values are unsigned
  - If `A = 0100` (6) and `B = 1010` (-6)
    - `C = 10000 not 00000`
    - i.e., `B` is zero-padded, not sign-extended

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

```verilog
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
```

Verilog Operators
Verilog "Variables"
- wire
- Variable used simply to connect components together
- reg
- Variable that stores 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?

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
```

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

```
assign A = X | (Y & ~Z);  // use of Boolean operators (- for bit-wise, ! for logical negation)
assign B[3:0] = 4'b10XX;  // bits can take on four values (0, 1, X, Z)
assign C[15:0] = 12'h00ff; // variables can be n-bits wide (MSB, LSB)
```

Comparator Example
- Make a 4-bit comparator from 4 x 1-bit comparators

```
module Compare(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;
    Compare cp0(A4[0], B4[0], e0, A10, B10);
    Compare cp1(A4[1], B4[1], e1, A11, B11);
    Compare cp2(A4[2], B4[2], e2, A12, B12);
    Compare cp3(A4[3], B4[3], e3, A13, B13);
    assign Equal = (~e0 & e1 & e2 & e3);
    assign Alarger = (A10 & e1 & e2 & e3);  // (Alarger & Equal);
    assign Blarger = (~Alarger & Equal);
endmodule
```

Announcements
- Card Key Access to 125 Cory
- EECS Keys/Cardkeys, Copy Cards Assistant
- Lorettta Lutcher
- 253 Cory, 642-1527, lorett@eecs
- Issues, keys and electronic cardkeys for Cory and Soda Halls Handles cardkey problems
- From the Reader on Quiz #1:
  - Common mistakes:
    - Inverted 1’s and the 0’s (SIM vs. I’m retinnes)
    - No overlapping circles, or not the largest circle possible for K-maps: for simplest SOP result (minimum cover, what constitutes adjacency)
    - Many mixed up the order on the K-map (some people started with 1, a list were confused) (e.g., Grey code order: 0 1 3 2 4 5 7 6 12 13 15 14 18 9 11 10)
module and_gate (out, in1, in2);
input in1, in2;
output out;
reg out;

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

Simple Behavioral Model: the always block
- always block
  - Always waiting for a change to a trigger signal
  - Then executes the body

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

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 if
- Same as C if statement

Verilog if
- Another way

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

// 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

```verilog
module mux (sel, A, B, C, D, Y);
input [1:0] sel;
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 (cont)

- Cases are executed sequentially
- Following implements a priority-encoder

```verilog
module encode (A, Y);
// simple encoder
output [2:0] Y;
input  [7:0] A;

always @(A)
begin
  reg [2:0] Y;
  Y = A[7];
  Y = Y | A[6];
  Y = Y | A[5];
  Y = Y | A[4];
  Y = Y | A[3];
  Y = Y | A[2];
  Y = Y | A[1];
  Y = Y | A[0];

  case (A[7])
    8'b00000010: Y = 8;
    8'b00000100: Y = 7;
    8'b00001000: Y = 6;
    8'b00010000: Y = 5;
    8'b00100000: Y = 4;
    8'b01000000: Y = 3;
    8'b10000000: Y = 2;
    8'b11000000: Y = 1;
    default: Y = 0';
  endcase
end
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

```verilog
module encode (A, Y);
// simple encoder
output [2:0] Y;
input  [7:0] A;

always @(A)
begin
  reg [2:0] Y;
  Y = A[7];
  Y = Y | A[6];
  Y = Y | A[5];
  Y = Y | A[4];
  Y = Y | A[3];
  Y = Y | A[2];
  Y = Y | A[1];
  Y = Y | A[0];

  case (A[7])
    8'b00000010: Y = 8;
    8'b00000100: Y = 7;
    8'b00001000: Y = 6;
    8'b00010000: Y = 5;
    8'b00100000: Y = 4;
    8'b01000000: Y = 3;
    8'b10000000: Y = 2;
    8'b11000000: Y = 1;
    default: Y = 0';
  endcase
end
endmodule
```

casex Example

```verilog
module casex (valid, Y);
input [7:0] A;
output [2:0] Y;
output valid;
// Don't care when input is not all 0's
reg [2:0] Y;
// target of assignment

always @(A)
begin
  case (A)
    8'b00000000: Y = 0;
    8'b00000100: Y = 1;
    8'b00001000: Y = 2;
    8'b00010000: Y = 3;
    8'b00100000: Y = 4;
    8'b01000000: Y = 5;
    8'b10000000: Y = 6;
    8'b11000000: Y = 7;
    default: Y = 0'; // Don't care when input is not all 0's
  endcase
end
endmodule
```
Verilog for

- for is similar to C
- for statement is executed at compile time (like macro expansion)
- Result is all that matters, not how result is calculated
- Use in testbenches only!

```verilog
module encode (A, Y);
input [7:0] A;
output [2:0] Y;
reg [2:0] Y;
integer i;
reg [7:0] test;
always @(A) begin
  test = 8b'00000001;
  Y = 3'bX;
  for (i = 0; i < 8; i = i + 1) begin
    if (A == test) Y = N;
    test = test << 1;
  end
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

```verilog
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

```verilog
// 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
```

Full-case and parallel-case

- for is similar to C
- for statement is executed at compile time (like macro expansion)
- Result is all that matters, not how result is calculated
- Use in testbenches only!