Introduction

- Netlist
- Design flow
- What is a HDL?
- Verilog
- Announcements
- Structural models
- Behavioral models
- Elements of the language
- Lots of examples

Outline

- Netlist
- Design flow
- What is a HDL?
- Verilog
- Announcements
- Structural models
- Behavioral models
- Elements of the language
- Lots of examples

Design Flow

Design Entry

High-level Analysis

Technology Mapping

Low-level Analysis

Netlist

- A key data structure (or representation) in the design process is the "netlist":
  - Network List
  - A netlist lists components and connects them with nodes:
    - g1 "and" n1 n2 n3
    - g2 "and" n4 n5 n6
    - g3 "or" n5 n6 n7
  - Netlist is needed for simulation and implementation.
  - Could be at the transistor level, gate level, ...
  - Could be hierarchical or flat.
  - How do we generate a netlist?

Review

- Advancing technology changes the trade-offs and design techniques
  - 2x transistors per chip every 18 months
- ASIC, Programmable Logic, Microprocessor
- Programmable logic invests chip real-estate to reduce design time & time to market
- FPGA:
  - programmable interconnect,
  - configurable logic blocks
  - LUT + storage
  - Block RAM
  - IO Blocks

Remember: to design is to represent

- How do we represent digital designs?
- Components
  - Logic symbol, truth table
  - Storage symbol, timing diagram
- Connections
  - Schematics

Human readable or machine readable???
Design Flow

- Circuit is described and represented:
  - Graphically (Schematics)
  - Textually (HDL)
- Result of circuit specification (and compilation) is a netlist of:
  - Generic primitives - logic gates, flip-flops, or
  - Technology specific primitives - LUTs/CLBs, transistors, discrete gates, or
  - Higher level library elements - adders, ALUs, register files, decoders, etc.

- High-level Analysis is used to verify:
  - Correct function
  - Rough:
  - Timing
  - Power
  - Cost
- Common tools used are:
  - Simulator - check functional correctness and
  - Static timing analyzer - estimates circuit delays based on timing model and delay parameters for library elements (or primitives).

- Design Flow:

  - Technology Mapping:
    - Converts netlist to implementation technology dependent details
    - Expands library elements,
    - Performs:
      - Place/Route,
      - Plan,
      - Route,
    - Low-level Analysis:
      - Simulation and Analysis Tools perform low-level checks with:
    - Accurate timing models,
    - Wire delay
    - For FPGAs this step could also use the actual device.

- Design Flow:

  - Technology Mapping:
  - Converts netlist to implementation technology dependent details
  - Expands library elements,
  - Performs:
    - Place/Route,
    - Plan,
    - Route,
  - Low-level Analysis:
    - Simulation and Analysis Tools perform low-level checks with:
  - Accurate timing models,
  - Wire delay
  - For FPGAs this step could also use the actual device.

- Design Flow:

  - Technology Mapping:
    - Converts netlist to implementation technology dependent details
    - Expands library elements,
    - Performs:
      - Place/Route,
      - Plan,
      - Route,
    - Netlist is used between and internally for all steps.

Design Entry

- Schematic entry/editing used to be the standard method in industry
- Used in EEC5150 until recently
  - Schematics are intuitive. They match our use of gate-level or block diagrams.
  - Somewhat physical. They imply a physical implementation.
  - Require a special tool (editor).
  - Unless hierarchy is carefully designed, schematics can be confusing and difficult to follow.

- Hardware Description Languages (HDLs) are the new standard
  - Except for PC board design, where schematics are still used.

- Basic Idea:
  - Language constructs describe circuits with two basic forms:
  - Structural descriptions similar to hierarchical netlist.
  - Behavioral descriptions use higher-level constructs (similar to conventional programming).
- Originally designed to help in abstraction and simulation.
  - Now “logic synthesis” tools exist to automatically convert from behavioral descriptions to gate netlist.
  - Greatly improves designer productivity.

  "Structural" example:
  ```
  Decade(output, a, b, c, d, e);
  ```
  "Behavioral" example:
  ```
  Case (a, b)
  00: [x0 x1 x2 x3] = 0x0;
  01: [x0 x1 x2 x3] = 0x2;
  10: [x0 x1 x2 x3] = 0x4;
  11: [x0 x1 x2 x3] = 0x6;
  ```

HDLs

- Basic Idea:
  - Language constructs describe circuits with two basic forms:
  - Structural descriptions similar to hierarchical netlist.
  - Behavioral descriptions use higher-level constructs (similar to conventional programming).
- Originally designed to help in abstraction and simulation.
  - Now “logic synthesis” tools exist to automatically convert from behavioral descriptions to gate netlist.
  - Greatly improves designer productivity.

  "Structural" example:
  ```
  Decade(output, a, b, c, d, e);
  ```
  "Behavioral" example:
  ```
  Case (a, b)
  00: [x0 x1 x2 x3] = 0x0;
  01: [x0 x1 x2 x3] = 0x2;
  10: [x0 x1 x2 x3] = 0x4;
  11: [x0 x1 x2 x3] = 0x6;
  ```
Design Methodology

- HDL Specification
  - Structure and Function (Behavior) of a Design
  - Simulation
  - Synthesis

Verification: Design
- Behave as Required?
- Functional I/O Behavior
- Register-Level (Architectural)
- Logic-Level (Gates)
- Timing: Waveform Behavior

Generation: Map
- Specification to Implementation

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, originally developed for simulation
  - Fairly efficient and easy to write
  - SDS Berkeley develops synthesis tools
  - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
  - Similar to Ada (emphasis on re-use and maintainability)
  - Simulation semantics viable
  - Very general but verbose

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
    - Behavioral Verilog
      - Do not list
      - Tedious to write, hard to decode
      - Essential without integrated design tools
      - Synthesized into a circuit that has this behavior
    - Result is only as good as the tools
  - Build up a hierarchy of modules

Structural Model - XOR

- Composition of primitive gates to form more complex module
- Note use of wire declaration!
  - By default, identifiers are wires

Structural Model: 2-to1 mux

//2-input multiplexer in gates
module mux2 (in0, in1, select, out);  
input in0,in1,select;  
output out;  
wire a0,w0,w1;  
not (a0, select);  
and (w0, a0, in0);  
or (out, w0, w1);  
endmodule // mux2  

Notes:
- comments
- “module”
- port list
- declarations
- wire type
- primitive gates
- instance names?
- list per type
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
```

2-to-1 mux behavioral description

- Notes:
  - behavioral descriptions use the keyword always followed by blocking procedural assignments
  - Target output of procedural assignments must be of type reg (not a real register)
  - Unlike wire types where the target output of an assignment may be continuously updated, a reg type retains its value until a new value is assigned (the assigning statement is executed).
  - Optional initial statement

```verilog
// Behavioral model of 2-to-1 multiplexer.
module mux2 (in0, in1, select, out);
  input in0, in1, select;
  output out;
  //
  reg out;
  always @ (in0 or in1 or select)
    if (select) out = in1;
  else out = in0;
endmodule // mux2
```

Verilog Help

- The lecture notes only cover the very basics of Verilog and mostly the conceptual issues.
- Textbook has examples.
- The Bhasker book is a good tutorial.
- On reserve in the Engineering library (starting Friday).

- The complete language specification from the IEEE is available on the class website under "Refs/Links"
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 < 0
  - Concatenating bits/vectors into a vector
    - e.g., sign extend
      - $b7:0 = [3(A[3]), A[3]:0]$;
  - Style: Use $a[7:0] = b[7:0] + c$; Not: a = b + c; if need to look at declaration

Verilog Numbers

- 14 - ordinary decimal number
- -14 - 2's complement representation
- 12'b0000_0100_0110 - binary number with 12 bits (_ is ignored)
- 12'b046 - hexadecimal number with 12 bits
- Verilog values are *unsigned*
  - If A = 0110 (6) and B = 1010(6)
    - C = 10000 not 00000
    - i.e., B is zero-padded, not sign-extended

Verilog Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
<td>Additive</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
<td>Multipli</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
<td>Divide</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
<td>Subtract</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
<td>Bitwis</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
<td>Bitwis</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>Greater th</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>Less than or eq</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>Greater than or eq</td>
</tr>
<tr>
<td>=</td>
<td>Equal</td>
<td>Equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
<td>Not equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>Less than or eq</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>Greater than or eq</td>
</tr>
<tr>
<td>==</td>
<td>Equal</td>
<td>Equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
<td>Not equal</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
- **usage:**
  - Don't confuse reg assignments with the combinational continuous
    assign statement (more soon)
  - Reg should only be used with always blocks (sequential logic, to be presented ...)

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

```verilog
module full adder (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
  - **Dataflow style**
    - use of Boolean operators (~ for bit-wise, | for logical negation)
    - bits can take on four values (0, 1, X, Z)
    - assign C[15:0] = 0'b0000;
    - use of arithmetic operator
      - multiple assignment (concatenated)
      - delay of performing computation, only used by simulator, not synthesis
Comparator Example

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

“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: 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

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

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, A0, A1, A2, A3, B0, B1, B2, B3;
Comparator cp0(A4[0], B4[0], e0, A10, B10);
Comparator cp1(A4[1], B4[1], e1, A11, B11);
Comparator cp2(A4[2], B4[2], e2, A12, B12);
Comparator cp3(A4[3], B4[3], e3, A13, B13);
assign Equal = (e0 & e1 & e2 & e3);
assign Alarger = (A13 | (A12 & e3) | (A11 & e2 & e1));
assign Blarger = (~Alarger & ~Equal);
endmodule
Behavioral with Bit Vectors

Verilog case

Verilog if

Hierarchy & Bit Vectors

Verilog case
Verilog case (cont)

- Cases are executed sequentially
  - The following implements a priority encoder

```verilog
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
  regvalid;

  always @(A)
  case (A)
    A[6]: Y = 6;
    A[7]: Y = 7;
    default: Y = 3'bX; // Don't care when input is all 0's
endcase
endmodule
```

Verilog cases

- 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!

```verilog
module encode (A, valid, Y);
  input [7:0] A;// 8-bit input vector
  output valid;// Asserted when an input is not all 0's
  reg [2:0] Y; // 3-bit encoded output
  reg regvalid;

  always @(A)
  case (A)
    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 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;// 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

  integer i; // Target of assignment
  always @(A)
  for (i = 0; i < 8; i = i + 1) count = count + neighbors[i];
  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
```

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
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)
    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
```

casex Example

```verilog
// Priority encoder
module encode (A, valid, Y);
  input [7:0] A;// 8-bit input vector
  output [2:0] Y;// 3-bit encoded output
  reg [2:0] Y; // Target of assignment
  reg valid;

  always @(A)
  case (A)
    A[0]: valid = 1;
    A[1]: Y = 0;
    A[2]: Y = 1;
    A[3]: Y = 2;
    A[4]: Y = 3;
    A[5]: Y = 4;
    A[6]: Y = 5;
    A[7]: Y = 6;
    default: valid = 1; // Don't care when input is all 0's
  endcase
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];
    if (count == 3) out = 1;
    else 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

Sequential Logic

// Parallel to Serial converter
module ParToSer(LD, X, out, CLK);
  input [3:0] X;
  output out, CLK; // synthesis full_case
  reg Q[0];
  assign out = Q[0];
  always @ (posedge CLK)
    if (LD) Q<=>X;
    else Q = Q+1;
endmodule // ParToSer

module FF (CLK, Q, D);
  input D, CLK; // synthesis full_case
  output Q;
  assign Q = D; // synthesis forever
  always @ (posedge CLK) Q=D;
endmodule // FF

Testbench

Top-level modules written specifically to test sub-modules.
Generally no ports.

- Notes:
  - module testmux,
    reg a, b, s;
    wire f;
    reg expected;
    
    mux2 myMux (.select(s), .in0(a), .in1(b), .out(f));

    initial
      begin
        a=0; b=0; s=1; expected=0;
        #10 a=1; b=0; expected=1;
        #10 a=1; b=0; expected=1;
        and
        $monitor("select=\%b in0=\%b in1=\%b out=\%b expected=\%b time=\%d",
          a, b, s, expected, $time);
      end
endmodule // testmux

Final thoughts

- Verilog looks like C, but it describes hardware
  - Multiple physical elements, Parallel activities
  - Temporal relationships
  - Basis for simulation and synthesis
  - figure out the circuit you want, then figure out how to express it in Verilog

- Understand the elements of the language
  - Modules, ports, wires, reg, primitive, continuous assignment, blocking statements, sensitivity lists, hierarchy
  - Best done through experience

- Behavioral constructs hide a lot of the circuit details but you as the designer must still manage the structure, data-communication, parallelism, and timing of your design.