Outline

Hardware Description
Language Overview
Verilog Introduction
Announcements

- Problem Set one posted. Start soon!
  - Due next Monday (another out Friday)
- Contact me if you are still trying to get enrolled.
Hardware Description
Languages
Design Entry

- Schematic entry/editing used to be the standard method in industry and universities.
  
 😊 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 on large designs.

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

- Basic Idea:
  - Language constructs describe circuits with two basic forms:
    - **Structural descriptions**: connections of components. Nearly one-to-one correspondence to with schematic diagram.
    - **Behavioral descriptions**: use high-level constructs (similar to conventional programming) to describe the circuit function.
  - Originally invented for simulation.
    - “logic synthesis” tools exist to automatically convert to gate level representation.
    - High-level constructs greatly improves designer productivity.
    - However, this may lead you to falsely believe that hardware design can be reduced to writing programs*

```
“Structural” example:
Decoder(output x0,x1,x2,x3; inputs a,b)
{
    wire abar, bbar;
    inv(bbar, b);
    inv(abar, a);
    and(x0, abar, bbar);
    and(x1, abar, b   );
    and(x2, a,   bbar);
    and(x3, a,   b   );
}
```

```
“Behavioral” example:
Decoder(output x0,x1,x2,x3; inputs a,b)
{
    switch [a b]
        case 00: [x0 x1 x2 x3] = 0x8;
        case 01: [x0 x1 x2 x3] = 0x4;
        case 10: [x0 x1 x2 x3] = 0x2;
        case 11: [x0 x1 x2 x3] = 0x1;
    endswitch;
}
```

*New tools and languages exist for this - called “high level synthesis”.

Warning: this is a fake HDL!
Hierarchically defines structure and/or function of circuit.

**HDL Specification**

**Simulation**
Verification: Does the design behave as required with regards to function, timing, and power consumption?

**Synthesis**
Maps specification to resources of implementation platform (FPGA or ASIC).

Note: This is not the entire story. Other tools are useful for analyzing HDL specifications. More on this later.
Some Hardware Description Languages

Verilog:
- Simple C-like syntax for structural and behavior hardware constructs
- Mature set of commercial tools for synthesis and simulation
- Used in EECS 151 / 251A

VHDL:
- Semantically very close to Verilog
- More syntactic overhead
- Extensive type system for “synthesis time” checking

System Verilog:
- Enhances Verilog with strong typing along with other additions
- Somewhat less mature tool-flow

BlueSpec:
- Invented at MIT
- Originally built within the Haskell programming language
- Now available commercially: bluespec.com

Chisel:
- Developed at UC Berkeley
- Used in CS152, and research projects
- Available at: www.chisel-lang.org
Verilog
Introduction
Verilog Introduction

- A **module** definition describes a component in a circuit

- Two ways to describe module contents:
  - Structural Verilog
    - List of sub-components and how they are connected
    - Just like schematics, but using text
    - You get precise control over circuit details
    - May be necessary to map to special resources of the FPGA/ASIC
  - Behavioral Verilog
    - Describe what a component does, not how it does it
    - May be simpler to write than structural description
    - Synthesized into a circuit that has this behavior
    - Result is only as good as the tools

- Build up a hierarchy of modules. Top-level module is your entire design (or the environment to test your design).

- Common approach is to use behavioral descriptions for “leaf cells” and structural to build hierarchy.
Verilog Modules and Instantiation

- Modules define circuit components.
- Instantiation defines hierarchy of the design.

```
module addr_cell (a, b, cin, s, cout);
input     a, b, cin;
output    s, cout;
endmodule
```

Note: A module is not a function in the C sense. There is no call and return mechanism. Think of it more like a hierarchical data structure.

```
module adder (A, B, S);
addr_cell ac1 (              );
endmodule
```

Instance of `addr_cell`

... connections ...

Note: Verilog syntax allows port declarations in port list.
module xor_gate (out, a, b);
input a, b;
output out;
wire aBar, bBar, t1, t2;
not invA (aBar, a);
not invB (bBar, b);
and and1 (t1, a, bBar);
and and2 (t2, b, aBar);
or or1 (out, t1, t2);
endmodule

- Notes:
  - The instantiated gates are not “executed”. They are active always.
  - xor gate already exists as a built-in (so really no need to define it).
/* 2-input multiplexor in gates */
module mux2 (in0, in1, select, out);
    input in0, in1, select;
    output out;
    wire s0, w0, w1;
    not (s0, select);
    and (w0, s0, in0), (w1, select, in1);
    or (out, w0, w1);
endmodule // mux2

A structural example of a 2-to1 multiplexor (mux) using gates. The code is written in a C++ style, with comments and instance names. Built-in gates can have more than 2 inputs. Multiple instances can share the same "master" name.
module mux4 (in0, in1, in2, in3, select, out);
input in0, in1, in2, in3;
input [1:0] select;
output out;
wire w0, w1;
mux2 m0 (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
m1 (.select(select[0]), .in0(in2), .in1(in3), .out(w1)),
m3 (.select(select[1]), .in0(w0), .in1(w1), .out(out));
endmodule // mux4

/* 2-input multiplexor in gates */
module mux2 (in0, in1, select, out);
input in0, in1, select;
output out;
wire s0, w0, w1;
not (s0, select);
and (w0, s0, in0),
and (w1, select, in1);
or (out, w0, w1);
endmodule // mux2

Signal array. Declares select[1], select[0]

Named ports. Highly recommended.
module foo (out, in1, in2);
  input       in1, in2;
  output      out;
assign out = in1 & in2;
endmodule

“continuous assignment”
Connects out to be the “and” of in1 and in2.

The assignment continuously happens, therefore any change on the rhs is reflected in out immediately (except for the small delay associated with the implementation of the &).

Not like an assignment in C that takes place when the program counter gets to that place in the program.
module FullAdder(a, b, ci, r, co);
  input a, b, ci;
  output r, co;

  assign r = a ^ b ^ ci;
  assign co = a&ci | a&b | b&ci;
endmodule

module Adder(A, B, R);
  input [3:0] A;
  input [3:0] B;
  output [4:0] R;

  wire c1, c2, c3;

  FullAdder
    add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1), .r(R[0]) ),
    add1(.a(A[1]), .b(B[1]), .ci(c1), .co(c2), .r(R[1]) ),
    add2(.a(A[2]), .b(B[2]), .ci(c2), .co(c3), .r(R[2]) ),
    add3(.a(A[3]), .b(B[3]), .ci(c3), .co(R[4]), .r(R[3]) );
endmodule
# Verilog Operators

<table>
<thead>
<tr>
<th>Verilog Operator</th>
<th>Name</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>bit-select or part-select</td>
<td></td>
</tr>
<tr>
<td>()</td>
<td>parenthesis</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical negation</td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>negation</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>reduction AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reduction OR</td>
</tr>
<tr>
<td>&amp;=</td>
<td>reduction NAND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reduction NOR</td>
</tr>
<tr>
<td>^</td>
<td>reduction XOR</td>
<td></td>
</tr>
<tr>
<td>^ or ^~</td>
<td>reduction XNOR</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>unary (sign) plus</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>unary (sign) minus</td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td>concatenation</td>
<td></td>
</tr>
<tr>
<td>{{}}</td>
<td>replication</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>binary plus</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>binary minus</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>logical equality</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>logical inequality</td>
<td></td>
</tr>
<tr>
<td>===</td>
<td>case equality</td>
<td></td>
</tr>
<tr>
<td>!==</td>
<td>case inequality</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise AND</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bit-wise XOR</td>
<td></td>
</tr>
<tr>
<td>^ or ^~</td>
<td>bit-wise XNOR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit-wise OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>conditional</td>
<td></td>
</tr>
</tbody>
</table>

Relational operators:

- >
- >=
- <
- <=

Equality operators:

- ==
- !=
- ===
- !==

Bit-wise operators:

- &
- ^
- ^ or ^~
- | |
- ||

Logical operators:

- &&
- ||
- ?:
Verilog Numbers

Constants:

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

Signal Values:

By default, Values are unsigned

if A = 0110 (6) and B = 1010 (treated as 10 not -6)
\[ C = 10000 \text{ not } 00000 \]
i.e., B is zero-padded, not sign-extended

wire signed [31:0] x;
Declares a signed (2’s complement) signal array.
Verilog Assignment Types
Continuous Assignment Examples

assign $R = X \mid (Y \& \sim Z)$;  
example use of bit-wise Boolean operators
assign $r = \& X$;  
example reduction operator
assign $R = (a == 1'b0) ? X : Y$;  
example conditional operator
assign $P = 8'hff$;  
example constants
assign $P = X \ast Y$;  
arithmetic operators (use with care!)
assign $P[7:0] = \{4\{X[3]\}, X[3:0]\}$;  
(example: sign-extension)
assign $\{cout, R\} = X + Y + cin$;  
bit field concatenation
assign $Y = A \ll 2$;  
bit shift operator
assign $Y = \{A[1], A[0], 1'b0, 1'b0\}$;  
equivalent bit shift
Non-continuous Assignments

A bit strange from a hardware specification point of view. Shows off Verilog roots as a simulation language.

“always” block example:

```verilog
module and_or_gate (out, in1, in2, in3);
    input    in1, in2, in3;
    output   out;
    reg      out;

    always @(in1 or in2 or in3) begin
        out = (in1 & in2) | in3;
    end
endmodule
```

Isn’t this just: `assign out = (in1 & in2) | in3;`?

Why bother?
Always Blocks

Always blocks give us some constructs that are impossible or awkward in continuous assignments.

**case statement example:**

```verilog
module mux4 (in0, in1, in2, in3, select, out);
    input in0,in1,in2,in3;
    input [1:0] select;
    output      out;
    reg         out;

    always @(in0 in1 in2 in3 select)
        case (select)
            2'b00: out=in0;
            2'b01: out=in1;
            2'b10: out=in2;
            2'b11: out=in3;
        endcase
endmodule // mux4
```

**keyword**

The statement(s) corresponding to whichever constant matches “select”, get applied.

**Couldn’t we just do this with nested “if”s?**

*Well yes and no!*
Nested if-else example:

```verilog
module mux4 (in0, in1, in2, in3, select, out);
    input in0, in1, in2, in3;
    input [1:0] select;
    output      out;
    reg         out;

    always @(in0 in1 in2 in3 select)
        if (select == 2'b00) out=in0;
        else if (select == 2'b01) out=in1;
        else if (select == 2'b10) out=in2;
        else out=in3;
endmodule // mux4
```

Nested if structure leads to “priority logic” structure, with different delays for different inputs (in3 to out delay > than in0 to out delay). Case version treats all inputs the same.
Modularity is essential to the success of large designs.

High-level primitives enable direct synthesis of behavioral descriptions (functions such as additions, subtractions, shifts (<< and >>), etc.

**Example: A 32-bit ALU**

**Function Table**

<table>
<thead>
<tr>
<th>F2</th>
<th>F1</th>
<th>F0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A + B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A + 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A - B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A - 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>A * B</td>
</tr>
</tbody>
</table>
Module Definitions

2-to-1 MUX

```verilog
module mux32two(i0, i1, sel, out);
    input [31:0] i0, i1;
    input sel;
    output [31:0] out;

    assign out = sel ? i1 : i0;

endmodule
```

3-to-1 MUX

```verilog
module mux32three(i0, i1, i2, sel, out);
    input [31:0] i0, i1, i2;
    input [1:0] sel;
    output [31:0] out;
    reg [31:0] out;

    always @(i0 or i1 or i2 or sel)
        begin
            case (sel)
                2'b00: out = i0;
                2'b01: out = i1;
                2'b10: out = i2;
                default: out = 32'bx;
            endcase
        end

endmodule
```

32-bit Adder

```verilog
module add32(i0, i1, sum);
    input [31:0] i0, i1;
    output [31:0] sum;

    assign sum = i0 + i1;

endmodule
```

32-bit Subtractor

```verilog
module sub32(i0, i1, diff);
    input [31:0] i0, i1;
    output [31:0] diff;

    assign diff = i0 - i1;

endmodule
```

16-bit Multiplier

```verilog
module mul16(i0, i1, prod);
    input [15:0] i0, i1;
    output [31:0] prod;

    assign prod = i0 * i1;

endmodule
```
module alu(a, b, f, r);
input [31:0] a, b;
input [2:0] f;
output [31:0] r;
wire [31:0] addmux_out, submux_out;
wire [31:0] add_out, sub_out, mul_out;
mux32two adder_mux(.io(b), .i1(32'd1), .sel(f[0]), .out(addmux_out));
mux32two sub_mux(.io(b), .i1(32'd1), .sel(f[0]), .out(submux_out));
add32 our_adder(.i0(a), .i1(addmux_out), .sum(add_out));
sub32 our_subtractor(.i0(a), .i1(submux_out), .diff(sub_out));
mul16 our_multiplier(.i0(a[15:0]), .i1(b[15:0]), .prod(mul_out));
mux32three output_mux(.i0(add_out), .i1(sub_out), .i2(mul_out), .sel(f[2:1]), .out(r));
endmodule
Top-Level ALU Declaration, take 2

- No Hierarchy:
- Declaration of the ALU Module:

```verbatim
module alu(a, b, f, r);
    input [31:0] a, b;
    input [2:0] f;
    output [31:0] r;
    always @ (a or b or f)
        case (f)
            3'b000:  r = a + b;
            3'b001:  r = a + 1'b1;
            3'b010:  r = a – b;
            3'b011:  r = a – 1'b1;
            3'b100:  r = a * b;
            default: r = 32'bx;
        endcase
    endmodule
```

Will this synthesize into 2 adders and 2 subtractors or 1 of each?
module FullAdder(a, b, ci, r, co);
  input a, b, ci;
  output r, co;
  assign r = a ^ b ^ ci;
  assign co = a&ci + a&b + b&cin;
endmodule

module Adder(A, B, R);
  input [3:0] A;
  input [3:0] B;
  output [4:0] R;
  wire c1, c2, c3;
  FullAdder
    add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1), .r(R[0]) ),
    add1(.a(A[1]), .b(B[1]), .ci(c1), .co(c2), .r(R[1]) ),
    add2(.a(A[2]), .b(B[2]), .ci(c2), .co(c3), .r(R[2]) ),
    add3(.a(A[3]), .b(B[3]), .ci(c3), .co(R[4]), .r(R[3]) );
endmodule
Parameters give us a way to generalize our designs. A module becomes a “generator” for different variations. Enables design/module reuse. Can simplify testing.

```verilog
module Adder(A, B, R);
    parameter N = 4;
    input [N-1:0] A;
    input [N-1:0] B;
    output [N:0] R;
    wire [N:0] C;
    genvar i;
    generate
        for (i=0; i<N; i=i+1) begin:bit
            FullAdder add(.a(A[i], .b(B[i]), .ci(C[i]), .co(C[i+1]), .r(R[i]));
        end
    endgenerate
    assign C[0] = 1'b0;
    assign R[N] = C[N];
endmodule
```

Declare a parameter with default value.

Note: this is not a port. Acts like a “synthesis-time” constant.

Replace all occurrences of “4” with “N”.

Variable exists only in the specification - not in the final circuit.

Keyword that denotes synthesis-time operations

For-loop creates instances (with unique names)

Adder adder4 ( ... );
Adder #(N(64)) adder64 ( ... );

Overwrite parameter N at instantiation.
More on Generate Loop

Permits variable declarations, modules, user defined primitives, gate primitives, continuous assignments, initial blocks and always blocks to be instantiated multiple times using a for-loop.

```vhdl
// Gray-code to binary-code converter
module gray2bin1 (bin, gray);
parameter SIZE = 8;
output [SIZE-1:0] bin;
input [SIZE-1:0] gray;

genvar i;

generate for (i=0; i<SIZE; i=i+1)
begin
  assign bin[i] = ^gray[SIZE-1:i];
end endgenerate
endmodule
```

- **Generate** if-else-if based on an expression that is deterministic at the time the design is synthesized.
- **Generate** case: selecting case expression must be deterministic at the time the design is synthesized.

Keywords that denotes synthesis-time operations

- `generate` creates instances of assignments
- Loop must have constant bounds

variable exists only in the specification - not in the final circuit.
Verilog in EECS 151/251A

- We use behavioral modeling at the bottom of the hierarchy
- Use instantiation to 1) build hierarchy and, 2) map to FPGA and ASIC resources not supported by synthesis.
- Favor continuous assign and avoid always blocks unless:
  - no other alternative: ex: state elements, case
  - helps readability and clarity of code: ex: large nested if else
- Use named ports.

Verilog is a big language. This is only an introduction.
- Complete IEEE Verilog-Standard document (1364-2005) linked to class website.
- Harris & Harris book chapter 4 is a good source.
- Be careful of what you read on the web. Many bad examples out there.
- We will be introducing more useful constructs throughout the semester. Stay tuned!