Verilog

- Hardware Description Languages
- Verilog
  - -- structural: modules, instances
  - -- dataflow: continuous assignment
  - -- sequential behavior: always blocks
  - -- pitfalls
  - -- other useful features

  Reminder: Lab #1 due tonight!

Lab Check-off Policies

- Last check-off is at 10:30p on Thursday. Please don't assume that *you* can wait until the last minute!

- No check-offs on Friday, Saturday

- Sunday check-offs are 1 day late, Monday 2 days late, etc.

- 20%/day late penalty
  - On-time check-off: 5 points
  - Sunday check-off: 4 points
  - …

- All labs must be checked off (even if for 0 points) before you can start your final project. We've learned that if you're struggling with the labs, the final project won't go very well.

The Need for HDLs

A specification is an engineering contract that lists all the goals for a project:

• goals include area, power, throughput, latency, functionality, test coverage, costs (NREs and piece costs), … Helps you figure out when you're done and how to make engineering tradeoffs. Later on, goals help remind everyone (especially management) what was agreed to at the outset!

• top-down design: partition the project into modules with well-defined interfaces so that each module can be worked on by a separate team. Gives the SW types a head start too! (Hardware/software codesign is currently all the rage…)

The Need for HDLs (cont'd.)

A <u>behavioral model</u> serves as an executable functional specification that documents the exact behavior of all the individual modules and their interfaces. Since one can run tests, this model can be refined and finally verified through simulation.

We need a way to talk about what hardware should do without actually designing the hardware itself, i.e., we need to separate behavior from implementation. We need a

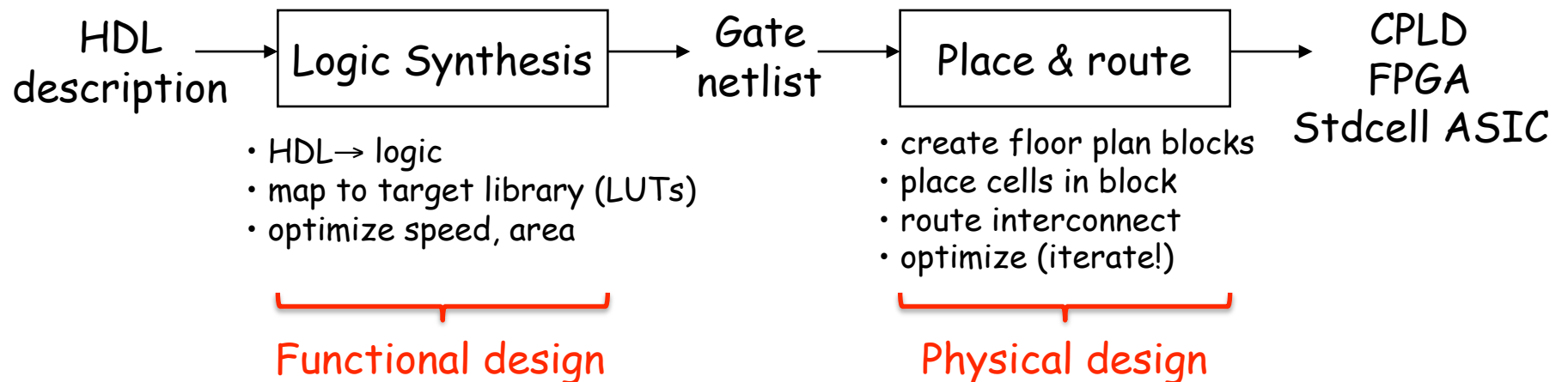<u>H</u>ardware <u>D</u>escription <u>L</u>anguage

If we were then able to <u>synthesize</u> an implementation directly from the behavioral model, we'd be in good shape!

Using an HDL description

So, we have an executable functional specification that
- documents exact behavior of all the modules and their interfaces
- can be tested & refined until it does what we want

An HDL description is the first step in a mostly automated process to build an implementation directly from the behavioral model

HDL description → Logic Synthesis → Gate netlist → Place & route → CPLD FPGA Stdcell ASIC

Logic Synthesis:
- HDL→ logic
- map to target library (LUTs)
- optimize speed, area

Place & route:
- create floor plan blocks
- place cells in block
- route interconnect
- optimize (iterate!)

Functional design

Physical design

A Tale of Two HDLs

VHDL

ADA-like verbose syntax, lots of redundancy (which can be good!)

Extensible types and simulation engine. Logic representations are not built in and have evolved with time (IEEE-1164).

Design is composed of entities each of which can have multiple architectures. A configuration chooses what architecture is used for a given instance of an entity.

Behavioral, dataflow and structural modeling. Synthesizable subset...

Harder to learn and use, not technology-specific, DoD mandate

Verilog

C-like concise syntax

Built-in types and logic representations. Oddly, this led to slightly incompatible simulators from different vendors.

Design is composed of modules.

Behavioral, dataflow and structural modeling. Synthesizable subset...

Easy to learn and use, fast simulation, good for hardware design

Verilog data values

Since we're describing hardware, we'll need to represent the values that can appear on wires. Verilog uses a 4-valued logic:

Value	Meaning
0	Logic zero, "low"
1	Logic one, "high"
Z or ?	High impedance (tri-state buses)
X	Unknown value (simulation)

"X" is used by simulators when a wire hasn't been initialized to a known value or when the predicted value is an illegitimate logic value (e.g., due to contention on a tri-state bus).

Verilog also has the notion of "drive strength" but we can safely ignore this feature for our purposes.

Numeric Constants

Constant values can be specified with a specific width and radix:

```
123            // default: decimal radix, unspecified width
'd123          // 'd = decimal radix
'h7B           // 'h = hex radix
'o173          // 'o = octal radix
'b111_1011     // 'b = binary radix, "_" are ignored
'hxx           // can include X, Z or ? in non-decimal constants
16'd5          // 16-bit constant 'b0000_0000_0000_0101
11'h1X?        // 11-bit constant 'b001_XXXX_ZZZZ
```

By default constants are unsigned and will be extended with 0's on left if need be (if high-order bit is X or Z, the extended bits will be X or Z too). You can specify a signed constant as follows:

```
8'shFF         // 8-bit twos-complement representation of -1
```

To be absolutely clear in your intent it's usually best to explicitly specify the width and radix.

Wires

We have to provide declarations* for all our named wires (aka "nets"). We can create buses – indexed collections of wires – by specifying the allowable range of indices in the declaration:

```
wire a,b,z;              // three 1-bit wires
wire [31:0] memdata;     // a 32-bit bus
wire [7:0] b1,b2,b3,b4;  // four 8-bit buses
wire [W-1:0] input;      // parameterized bus
```

Note that [0:7] and [7:0] are both legitimate but it pays to develop a convention and stick with it. Common usage is [MSB:LSB] where MSB > LSB; usually LSB is 0. Note that we can use an expression in our index declaration but the expression's value must be able to be determined at compile time. We can also build unnamed buses via concatenation:

```
{b1,b2,b3,b4}  // 32-bit bus, b1 is [31:24], b2 is [23:16], …
{4{b1[3:0]},16'h0000}  // 32-bit bus, 4 copies of b1[3:0], 16 0's
```

* Actually by default undeclared identifiers refer to a 1-bit wire, but this means typos get you into trouble. Specify "`default_nettype none" at the top of your source files to avoid this bogus behavior.

Basic building block: modules

In Verilog we design modules, one of which will be identified as our top-level module. Modules usually have named, directional ports (specified as input, output or inout) which are used to communicate with the module.

Don't forget this ";"

```
// 2-to-1 multiplexer with dual-polarity outputs
module mux2(input a,b,sel, output z,zbar);
  wire selbar,z1,z2;     // wires internal to the module
  // order doesn't matter – all statements are
  // executed concurrently!
  not i1(selbar,sel);    // inverter, name is "i1"
  and a1(z1,a,selbar);   // port order is (out,in1,in2,…)
  and a2(z2,b,sel);
  or  o1(z,z1,z2);
  not i2(zbar,z);
endmodule
```

In this example the module's behavior is specified using Verilog's built-in Boolean modules: not, buf, and, nand, or, nor, xor, xnor. *Just say no!* We want to specify behavior, not implementation!

Continuous assignments

If we want to specify a behavior equivalent to combinational logic, use Verilog's operators and continuous assignment statements:

```verilog
// 2-to-1 multiplexer with dual-polarity outputs
module mux2(input a,b,sel, output z,zbar);
  // again order doesn't matter (concurrent execution!)
  // syntax is "assign LHS = RHS" where LHS is a wire/bus
  // and RHS is an expression
  assign z = sel ? b : a;
  assign zbar = ~z;
endmodule
```

Conceptually assign's are evaluated continuously, so whenever a value used in the RHS changes, the RHS is re-evaluated and the value of the wire/bus specified on the LHS is updated.

This type of execution model is called "dataflow" since evaluations are triggered by data values flowing through the network of wires and operators.

Boolean operators

- Bitwise operators perform bit-oriented operations on vectors
  - ~(4'b0101) = {~0,~1,~0,~1} = 4'b1010
  - 4'b0101 & 4'b0011 = {0&0, 1&0, 0&1, 1&1} = 4'b0001

- Reduction operators act on each bit of a single input vector
  - &(4'b0101) = 0 & 1 & 0 & 1 = 1'b0

- Logical operators return one-bit (true/false) results
  - !(4'b0101) = 1'b0

Bitwise

~a	NOT
a & b	AND
a | b	OR
a ^ b	XOR
a ~^ b a ^~ b	XNOR

Reduction

&a	AND
~&a	NAND
|a	OR
~|a	NOR
^a	XOR
~^a ^~a	XNOR

Logical

!a	NOT
a && b	AND
a || b	OR
a == b a != b	[in]equality returns x when x or z in bits. Else returns 0 or 1
a === b a !== b	case [in]equality returns 0 or 1 based on bit by bit comparison

Note distinction between ~a and !a
when operating on multi-bit values

Other operators

Conditional

a ? b : c	If a then b else c

Relational

a > b	greater than
a >= b	greater than or equal
a < b	Less than
a <= b	Less than or equal

Arithmetic

-a	negate
a + b	add
a - b	subtract
a * b	multiply
a / b	divide
a % b	modulus
a ** b	exponentiate
a << b	logical left shift
a >> b	logical right shift
a <<< b	arithmetic left shift
a >>> b	arithmetic right shift

Hierarchy: module instances

Our descriptions are often hierarchical, where a module's behavior is specified by a circuit of module instances:

```
// 4-to-1 multiplexer
module mux4(input d0,d1,d2,d3, input [1:0] sel, output z);
  wire z1,z2;
  // instances must have unique names within current module.
  // connections are made using .portname(expression) syntax.
  // once again order doesn't matter…
  mux2 m1(.sel(sel[0]),.a(d0),.b(d1),.z(z1));  // not using zbar
  mux2 m2(.sel(sel[0]),.a(d2),.b(d3),.z(z2));
  mux2 m3(.sel(sel[1]),.a(z1),.b(z2),.z(z));
  // could also write "mux2 m3(z1,z2,sel[1],z,)" NOT A GOOD IDEA!
endmodule
```

Connections to a module's ports are made using a syntax that specifies both the port name and the wire(s) that connects to it, so ordering of the ports doesn't have to be remembered.

This type of hierarchical behavioral model is called "structural" since we're building up a structure of instances connected by wires. We often mix dataflow and structural modeling when describing a module's behavior.

Parameterized modules

```
// 2-to-1 multiplexer, W-bit data
module mux2 #(parameter W=1)   // data width, default 1 bit
            (input [W-1:0] a,b,
             input sel,
             output [W-1:0] z);
  assign z = sel ? b : a;
  assign zbar = ~z;
endmodule

// 4-to-1 multiplexer, W-bit data
module mux4 #(parameter W=1)   // data width, default 1 bit
            (input [W-1:0] d0,d1,d2,d3,
             input [1:0] sel,
             output [W-1:0] z);
  wire [W-1:0] z1,z2;

  mux2 #(.W(W)) m1(.sel(sel[0]),.a(d0),.b(d1),.z(z1));
  mux2 #(.W(W)) m2(.sel(sel[0]),.a(d2),.b(d3),.z(z2));
  mux2 #(.W(W)) m3(.sel(sel[1]),.a(z1),.b(z2),.z(z));
endmodule
```

*could be an expression evaluable at compile time;
if parameter not specified, default value is used*

Sequential behaviors

There are times when we'd like to use sequential semantics and more powerful control structures – these are available inside sequential `always` blocks:

```verilog
// 4-to-1 multiplexer
module mux4(input a,b,c,d, input [1:0] sel, output reg z,zbar);
  always @(*) begin
    if (sel == 2'b00) z = a;
    else if (sel == 2'b01) z = b;
    else if (sel == 2'b10) z = c;
    else if (sel == 2'b11) z = d;
    else z = 1'bx;    // when sel is X or Z
    // statement order matters inside always blocks
    // so the following assignment happens *after* the
    // if statement has been evaluated
    zbar = ~z;
  end
endmodule
```

`always` @(*) blocks are evaluated whenever any value used inside changes. Equivalently we could have written

```verilog
always @(a, b, c, d, sel) begin … end    // careful, prone to error!
```

reg vs wire

We've been using `wire` declarations when naming nets (ports are declared as wires by default). However nets appearing on the LHS of assignment statements inside of `always` blocks *must* be declared as type `reg`.

I don't why Verilog has this rule! I think it's because traditionally `always` blocks were used for sequential logic (the topic of next lecture) which led to the synthesis of hardware registers instead of simply wires. So this seemingly unnecessary rule really supports historical usage – the declaration would help the reader distinguish registered values from combinational values.

We can add the `reg` keyword to `output` or `inout` ports (we wouldn't be assigning values to `input` ports!), or we can declare nets using `reg` instead of `wire`.

```
output reg [15:0] result      // 16-bit output bus assigned in always block
reg flipflop;                 // declaration of 1-bit net of type reg
```

Case statements

Chains of if-then-else statements aren't the best way to indicate the intent to provide an alternative action for every possible control value. Instead use case:

```verilog
// 4-to-1 multiplexer
module mux4(input a,b,c,d, input [1:0] sel, output reg z,zbar);
  always @(*) begin
    case (sel)
      2'b00: z = a;
      2'b01: z = b;
      2'b10: z = c;
      2'b11: z = d;
      default: z = 1'bx;   // in case sel is X or Z
    endcase
    zbar = ~z;
  end
endmodule
```

case looks for an exact bit-by-bit match of the value of the case expression (e.g., sel) against each case item, working through the items in the specified order. casex/casez statements treat X/Z values in the selectors as don't cares when doing the matching that determines which clause will be executed.

Unintentional creation of state

Suppose there are multiple execution paths inside an always block, i.e., it contains if or case statements, and that on some paths a net is assigned and on others it isn't.

```verilog
// 3-to-1 multiplexer ????
module mux3(input a,b,c, input [1:0] sel, output reg z);
  always @(*) begin
    case (sel)
      2'b00: z = a;
      2'b01: z = b;
      2'b10: z = c;
      // if sel is 2'b11, no assignment to z!!??
    endcase
  end
endmodule
```

So sometimes z changes and sometimes it doesn't (and hence keeps its old value). That means the synthesized hardware has to have a way of remembering the state of z (i.e., it's old value) since it's no longer just a combinational function of sel, a, b, and c. We'll talk about such hardware in the next lecture; meanwhile that wasn't what we intended here.

Keeping logic combinational

To avoid the unintentional creation of state, ensure that each variable that's assigned in an `always` block always gets assigned a new value at least once on every possible execution path.

```verilog
// 3-to-1 multiplexer
module mux3(input a,b,c, input [1:0] sel, output reg z);
  always @ (*) begin
    z = 1'bx;      // a second assignment may happen below
    case (sel)
      2'b00: z = a;
      2'b01: z = b;
      2'b10: z = c;
    endcase
  end
endmodule
```

It's good practice when writing combinational `always` blocks to provide a `default:` clause for each `case` statement and an `else` clause for each `if` statement.

Other useful Verilog features

- Additional control structures: `for`, `while`, `repeat`, `forever`
- Procedure-like constructs: functions, tasks
- One-time-only initialization: `initial` blocks
- Compile-time computations: `generate`, `genvar`
- System tasks to help write simulation test jigs
  - Stop the simulation: `$finish(…)`
  - Print out text, values: `$display(…)`
  - Initialize memory from a file: `$readmemh(…)`, `$readmemb(…)`
  - Capture simulation values: `$dumpfile(…)`, `$dumpvars(…)`
  - Explicit time delays: *#nnn*
- Compiler directives
  - Macro definitions: `` `define ``
  - Conditional compilation: `` `ifdef ``, …
  - Include other source files: `` `include ``
  - Control simulation time units: `` `timescale ``
  - No implicit net declarations: `` `default_nettype none ``