Hardware Description Languages

- **Basic idea:**
  - Language constructs describe circuits with two basic forms:
    - **Structural descriptions:** connections of components (gates & flip-flops). Nearly one-to-one correspondence with schematic diagram (circuit structure).
    - **Behavioral descriptions:** use statements (assignments and tests of logical conditions) to describe the relationships between inputs and outputs (circuit function).
- **Originally invented for simulation**
  - Now “logic synthesis” tools exist to automatically convert from HDL source to circuits.
  - High-level constructs greatly improve designer productivity.
  - However, this may lead to a false belief that hardware design is the same as writing programs!

* Describing hardware with a language is similar, however, to writing a parallel program.

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

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

Hierarchically defines structure and/or function of circuit.

- HDL Specification
  - Simulation
  - Synthesis

Verification: Checks if the design behaves as required with regards to function, timing, and power consumption.
Maps specification to resources of the implementation platform (FPGA or custom silicon).

Top-Down Architecture (1)

- Top Down Refinement Process
- Start Here:

Inputs → Project → Outputs
History of the Verilog HDL

- **1984**: Gateway Design Automation introduced Verilog-XL
  - digital logic simulator
  - The Verilog language was part of the Verilog-XL simulator
  - The language was mostly created by 1 person, Phil Moorby
  - The language was intended to be used with only 1 product
- **1989**: Gateway merged into Cadence Design Systems
- **1990**: Cadence made the Verilog HDL public domain
- **1995**: The IEEE standardized the Verilog HDL (IEEE 1364)
- **2001**: The IEEE enhanced the Verilog HDL for modeling scalable designs, deep sub-micron accuracy, etc.
- **2005**: The IEEE added minor corrections, spec clarifications, and a few new language features
- **2009**: The IEEE standardized SystemVerilog, with many new features and capabilities to aid design verification and design modeling
Verilog Introduction

- A module definition describes a component in a circuit
- Two ways to describe module contents:
  - Structural Verilog
    - Lists sub-components and how they are connected
    - Just like schematics, but using text
    - Tedious to write, hard to understand
    - You get precise control over circuit details
    - May be necessary to map to special resources of the FPGA
  - Behavioral Verilog
    - Describes what a component does, not how it does it
    - 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 test environment for your design).

Verilog Modules

- Verilog modules are the building blocks for Verilog designs
- One module
- Modules instantiating other modules hierarchically
Contents of a Verilog Module

- Modules may represent:
  - An entire design
  - Major hierarchical blocks within a design
  - Individual components within a design

- Modules are completely self contained
  - The only things "global" in Verilog are the names of modules and primitives
  - Verilog does not have global variables or functions

```
module name (ports);
  port declarations
  data type declarations
  functionality
  timing
endmodule
```

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

module adder (A, B, S);
...
addr_cell ac1 ( ... connections ... );
addr_cell ac2 ( ... connections ... );
...
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.
Verilog Modules and Instantiation

- Verilog supports ANSI C style port declarations
  - The port direction and data type of the signal can be included in the port list

```
module addr_cell (input wire a, input wire b, input wire cin, output reg sum, cout);
endmodule
```

```
module adder (A, B, S);
...
addr_cell ac1 ( ... connections ... );
addr_cell ac2 ( ... connections ... );
...
endmodule
```

Verilog Logical System

- Verilog uses four-valued logic system
- A 1-bit signal can take on one of 4 values:
  0  Logical 0, or false
  1  Logical 1, or true
  x  An unknown/undefined logical value
  z  High impedance (floating), as in three-state logic
- Verilog has built-in bitwise boolean operators (see table in a later slide)
Verilog Nets and Wires

- Verilog has two classes of signals: Nets and Variables
- A **net** corresponds to a wire in a physical circuit and provides connectivity between modules
  - **wire** is the default Net type
    - 'wire' is any signal name that appears in a module’s input/output port list, but not in module’s net declaration
    - 'wire' can be a **scalar** (single connection) or a **vector** (multiple connection)
- **Verilog net types**: wire, tri, triand, trior, tri0, tri1, trireg, wand, wor, supply0, supply1
  - supply0, supply1 are considered to be permanently wired to the power rail
  - 'wire' is conventionally used when a single driver is present
  - 'tri' is used when multiple drivers are present
    - When a 'tri' net is driven to a single value by ≥1 drivers, it takes on that value
    - When a 'tri' net is undriven, it floats (value 'z')
    - When it's driven to different values (0, 1, or x) by different drivers, it is in contention (value 'x')
- 'wire' is obsolete in SystemVerilog; instead, use the **logic** signal type

Verilog Internal Variables

- Internal variables store values during a Verilog module’s execution
  - They are neither inputs nor outputs, but are used only **internal to the module**
  - Don’t have physical significance in a circuit
  - Used when describing circuit’s behavior, in “procedural code” when we need to break a complex function into intermediate steps
- A variable can be assigned value in one Verilog statement; retains this value until overwritten in a later statement
  - Unlike a Net, a variable’s value can be changed only within procedural code in a module, not from outside the module
  - Input & inout ports of a module cannot be variables; they must be 'net' types (e.g., 'wire')
  - Output ports can be either be 'net' or variable ('reg') types
- Two common types of variables:
  - **reg** (in old Verilog, but **logic** in SystemVerilog)
  - **integer** (used as loop control variables, e.g., **for** loops)
- 'reg' is **NOT** a register or flip-flop
  - It’s just a variable used on the left hand side of <= or = assignment statements
  - It’s replaced with **logic** in SystemVerilog
Verilog Numbers

- **Constants / Literals**
  - **14** ordinary decimal number
  - **-14** 2’s complement representation
  - **12'b0000_0100_0110** 12-bit binary number ("_" is ignored)
  - **12'h046** 12-bit hexadecimal number
  - **4'bx** 4-bit binary number with unknown value **xxxx**
  - **8'hfx** 8-bit hexadecimal number, equivalent to **8'b1111_xxxx**

- Parameter declaration for defining named constants
  - **parameter BUS_SIZE = 32, MSB = BUS_SIZE-1;**

- **Signal values**
  - By default, Values are unsigned
    - e.g., **C[4:0] = A[3:0] + B[3:0];**
    - if **A = 0110 (6) and B = 1010(–6)** then **C = 10000 not 00000**
    - i.e., B is zero-padded, not sign-extended
  - **wire signed [31:0] x;**
    - Declares a signed (2’s complement) signal array.

Vectors and Bit Selection

- **Vector** is a group of individual 1-bit signals
  - Nets, variables, and constants can all be vectors
  - Examples:
    - **reg [7:0] byte1, byte2, byte3;**
    - **reg [15:0] word1, word2;**
    - **reg [1:16] Zbus;**
  - Note: in SystemVerilog use **logic** instead of **reg**

- **Bit select** syntax to select individual bits
  - Example: **byte1[7]** selects the leftmost bit

- **Part select** selects a range of bits
  - Example: **byte1[5:2]** selects the middle 4 bits
Verilog Operators (1)

- **Concatenation operator** `{ }` joins together two or more bits or vectors into a single vector
  - Example: `{2'b00, 2'b11}` produces `{4'b0011}`

- **Replication operator** `n{ }` replicates a bit or vector `n` times
  - Example: `{2{byte1}, 2{byte2}}` produces a 32-bit vector `{byte1, byte1, byte2, byte2}
  - “Bit swizzling”: using bit/part select and concatenation to form busses
    - Example: `{c[2:1], {3{d[0]}}, c[0], 3'b101}` forms a 9-bit bus `c2c1d0d0d0c0101`

- See next-slide table for more operators

- **Padding**: vectors of different sizes are aligned on their rightmost bits and padded with zeros at left
  - Example: `2'b11 & 4'b0101` produces `4'b0001`

## Verilog Operators (2)

<table>
<thead>
<tr>
<th>Verilog Operator</th>
<th>Name</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{ }</code></td>
<td>bit-select or part-select</td>
<td></td>
</tr>
<tr>
<td><code>( )</code></td>
<td>parenthesis</td>
<td></td>
</tr>
<tr>
<td><code>!</code></td>
<td>logical negation</td>
<td>Logical</td>
</tr>
<tr>
<td><code>-</code></td>
<td>negation</td>
<td>Logical</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>reduction AND</td>
<td>Reduction</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>reduction OR</td>
</tr>
<tr>
<td><code>~</code></td>
<td>reduction NAND</td>
<td>Reduction</td>
</tr>
<tr>
<td><code>^</code></td>
<td>reduction XOR</td>
<td>Reduction</td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td>arithmatic shift left</td>
<td>Arithmetic</td>
</tr>
<tr>
<td><code>&gt;&gt;&gt;</code></td>
<td>arithmatic shift right</td>
<td>Arithmetic</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater than</td>
<td>Relational</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>greater than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>less than</td>
<td>Relational</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>less than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td><code>==</code></td>
<td>logical equality</td>
<td>Equality</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>logical inequality</td>
<td>Equality</td>
</tr>
<tr>
<td><code>? :</code></td>
<td>case equality</td>
<td>Equality</td>
</tr>
<tr>
<td><code>===</code></td>
<td>case inequality</td>
<td>Equality</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>bit-wise AND</td>
<td>Bit-wise</td>
</tr>
<tr>
<td><code>~&amp;</code></td>
<td>bit-wise XOR</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>bit-wise OR</td>
</tr>
<tr>
<td>`~</td>
<td>`</td>
<td>bit-wise XOR</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>logical AND</td>
<td>Logical</td>
</tr>
<tr>
<td><code>~&amp;</code></td>
<td>logical AND</td>
<td>Logical</td>
</tr>
<tr>
<td><code>^</code></td>
<td>logical XOR</td>
<td>Logical</td>
</tr>
<tr>
<td><code>~^</code></td>
<td>logical XOR</td>
<td>Logical</td>
</tr>
<tr>
<td><code>?:</code></td>
<td>conditional</td>
<td>Conditional</td>
</tr>
</tbody>
</table>
### Verilog Operators (3)

- **Built-in arithmetic operators** treat vectors as unsigned integers; leftmost bit of a vector is MSB
- **Shift operator** shifts the 1\textsuperscript{st} operand by a number of positions given by the 2\textsuperscript{nd} operand
  - Example: `8'b11010011<<3` gives `8'b10011000`
- **Boolean reduction operators** take a single vector operand and collapse it to a 1-bit result
  - Reduction operators combine all bits in the vector and return a 1-bit result
  - Example: `^word` produces `1'b1` if odd number of bits of 'word' are 1 (parity calculation using XOR operation `^`)

### Verilog Operators (4)

**Arithmetic Shift Operators**

- The `>>>` token does an **arithmetic shift right**, filling with the value of the sign bit
  - Different than the `>>` bit shift right operator, which always fills with zero
- The `<<<` token does an **arithmetic shift left**, filling with zeros
  - Same functionality as the `<<` bit shift left operator

**Example:**
  - Given: `in = 8'b11001010;
    assign out = in >> 3; // bit shift right results in 8'b00011001
    assign out = in >>> 3; // arithmetic shift right results in 8'b11111001
  ```
"case equality" operator  ===

- == tests **logical equality**
  (tests for 1 and 0, all other will result in x)
- === tests 4-state logical equality
  (tests for 1, 0, z and x)
- Example, after executing `dataoutput = 52'bx;`
  - if `{dataoutput[7:0] == 8'bx} begin ...
    versus
  - if `{dataoutput[7:0] === 8'bx} begin ...
- the second gives 1, but the first gives 0.
  - The result of `dataoutput == 8'bx` is not really "0", it is "x".
    However, both "0" and "x" are false values, meaning the body of
    the if will not be executed.
  - For the === and !== operators, bits with x and z are included in the
    comparison and must match for the result to be true.
  - So, a == b is 'a equals b' and a === b is 'a really equals b'