Ways of specifying circuits

- Schematic
  - Structural description
  - Describe circuit as interconnected elements
    - Build complex circuits using hierarchy
    - Large circuits are unreadable

- HDLs
  - Hardware description languages
    - Not programming languages
    - Parallel languages tailored to digital design
  - Synthesize code to produce a circuit
Hardware description languages (HDLs)

- **Abel (~1983)**
  - Developed by Data-I/O
  - Targeted to PLDs
  - Limited capabilities (can do state machines)

- **Verilog (~1985)**
  - Developed by Gateway (now part of Cadence)
  - Similar to C

- **VHDL (~1987)**
  - DoD sponsored
  - Similar to Ada
Verilog versus VHDL

- Both “IEEE standard” languages
- Most tools support both
- Verilog is “simpler”
  - Less syntax, fewer constructs
- VHDL is structured for large, complex systems
  - Better modularization
Simulation versus synthesis

-present

- Early HDLs supported execution/simulation only
  - Hand transform code to a schematic

- Current HDLs support direct synthesis to hardware
  - A “synthesizable subset” of the language

![Diagram of HDL description, synthesis, and circuit with functional and timing validation.]
Simulation versus synthesis (con’t)

✨ Simulation
- Models what a circuit does, not how it does it
  - e.g. multiply
    - Just say “*”, ignoring the implementation possibilities
- Includes functions and timing
- Allows you to quickly test design options

✨ Synthesis
- Converts your code to a netlist
  - Description of interconnected circuit elements
  - No timing
- Tools map your netlist to hardware

✨ Verilog and VHDL both simulate and synthesize
Simulation

- You provide a circuit environment
  - Using misc non-circuit constructs
    - Read files, print, control simulation
  - Using Verilog simulation code
    - A “test fixture”
      - A specification
      - Tests if circuit behavior (I/O) is correct
Structural versus behavioral Verilog

- **Structural**
  - Describe explicit circuit elements
  - Describe explicit connections between elements
    - e.g., logic gates are instantiated and connected to others
  - Just like schematics, but using text

- **Behavioral**
  - Describes circuit as algorithms/programs
    - What a component does
    - Input/output behavior
  - Many possible circuits could have same behavior
    - e.g., different implementation of a Boolean function
Levels of abstraction

Verilog supports 4 description levels:
- Switch
- Gate
- Dataflow
- Algorithmic

Can mix & match levels in a design

Designs that combine dataflow and algorithmic constructs and synthesize are called RTL
- Register Transfer Level
What you will do...

- Use a synthesizeable subset of Verilog
  - e.g. no “initial” blocks
  - Using structural and “synthesizeable” behavioral Verilog

- **Will** simulate your Verilog code
  - Use ActiveHDL

- **Will** synthesize your code
  - Use SimplifyPro

- **Will** map your netlist
  - ISE

- **Will** simulate your netlist
  - After synthesis
  - All your code will synthesize (by necessity)
Verilog tips

- **Do not** write C-code
  - Don’t write algorithms
    - You will not get efficient circuits
    - Compilers don’t map algorithms to circuits well

- **Do** describe hardware circuits
  - Don’t start coding until you have a complete dataflow diagram

- **References**
Modules

✧ The basic building block
  ➔ Instance into a design
  ➔ Illegal to nest module defs
  ➔ Example: 4-bit adder

module add4 (A, B, SUM, OVER) ;

    input [3:0] A ;
    input [3:0] B ;
    output [3:0] SUM ;
    output OVER ;


endmodule
Modules

- Modules are circuit components
  - “parameter list” is a list of external connections
    - A list of ports
  - Port types: “input”, “output” or “inout”
    - inout are used on tri-state buses

```verilog
module full_addr (A, B, Cin, S, Cout);
  input A, B, Cin;
  output S, Cout;
  assign {Cout, S} = A + B + Cin;
endmodule
```
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 orl (out, t1, t2);
endmodule

Note: Verilog is case sensitive
[] All keywords are lowercase
module full_addr (A, B, Cin, S, Cout);
    input    A, B, Cin;
    output   S, Cout;

    assign {Cout, S} = A + B + Cin;
endmodule

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
module and_gate (out, in1, in2);
    input         in1, in2;
    output        out;

    assign out = in1 & in2;
endmodule

Note: AND is a Verilog primitive, so you can also write

and and_gate(out, in1, in2);
Data types

- Values on a wire
  - 0, 1, x (don’t care), z (tristate)

- Vectors
    - An unsigned integer value

  - Concatenating bits/vectors
    - e.g. sign extend
    - Style: Use \( a[7:0] = b[7:0] + c; \)
      - \( Not \) \( a = b + c; \)
Numbers

✧ 14
  ➢ Decimal number

✧ –14
  ➢ 2’s complement binary of the decimal number

✧ 12’b0000_0100_0110
  ➢ 12 bit binary number (_ is ignored)

✧ 3’h046
  ➢ 12 bit hexadecimal number

✧ Verilog values are unsigned
    ✧ if A = 0110 (6) and B = 1010 (–6), then C = 10000 (not 00000)
    ✧ B is zero-padded, not sign-extended
## 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>Logical</td>
</tr>
<tr>
<td>~</td>
<td>negation</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>&amp;</td>
<td>reduction AND</td>
<td>Reduction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reduction OR</td>
</tr>
<tr>
<td>-&amp;</td>
<td>reduction NAND</td>
<td>Reduction</td>
</tr>
<tr>
<td>^</td>
<td>reduction NOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>^</td>
<td>reduction XOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>^ or ^=</td>
<td>reduction XNOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>+</td>
<td>unary (sign) plus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>unary (sign) minus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>{ }</td>
<td>concatenation</td>
<td>Concatenation</td>
</tr>
<tr>
<td>{{ }}</td>
<td>replication</td>
<td>Replication</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>+</td>
<td>binary plus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>binary minus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>Shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>Shift</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>Relational</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td>==</td>
<td>logical equality</td>
<td>Equality</td>
</tr>
<tr>
<td>!=</td>
<td>logical inequality</td>
<td>Equality</td>
</tr>
<tr>
<td>===</td>
<td>case equality</td>
<td>Equality</td>
</tr>
<tr>
<td>!==</td>
<td>case inequality</td>
<td>Equality</td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise AND</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>^</td>
<td>bit-wise XOR</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>^ or ^=</td>
<td>bit-wise XNOR</td>
<td>Bit-wise</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit-wise OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>Logical</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>conditional</td>
<td>Conditional</td>
</tr>
</tbody>
</table>
Variables

- **wire**
  - Connects components together

- **reg**
  - Saves a value
    - Part of a behavioral description
    - Usually corresponds to a wire
  - Does *NOT* necessarily become a register when you synthesize

- **The rule**
  - Declare a variable as reg if it is a target of an assignment statement
    - Continuous assign doesn’t count