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)

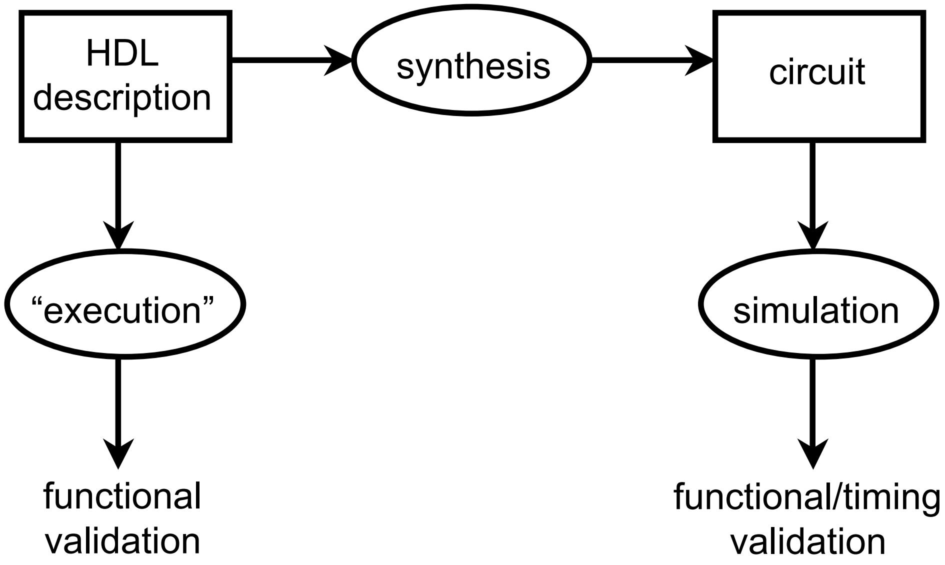
- Abel (~1983)
  - Developed by Data-I/O
  - Targeted to PLDs
  - Limited capabilities (can do state machines)
- Verilog (~1985)
  - Developed by Gateway (now part of Cadence)
  - Similar to C
- VHDL (~1987)
  - DoD sponsored
  - Similar to Ada

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

- Early HDLs supported execution/simulation only
  - Hand transform code to a schematic
- Current HDLs support direct synthesis to hardware
  - A "synthesizeable subset" of the language



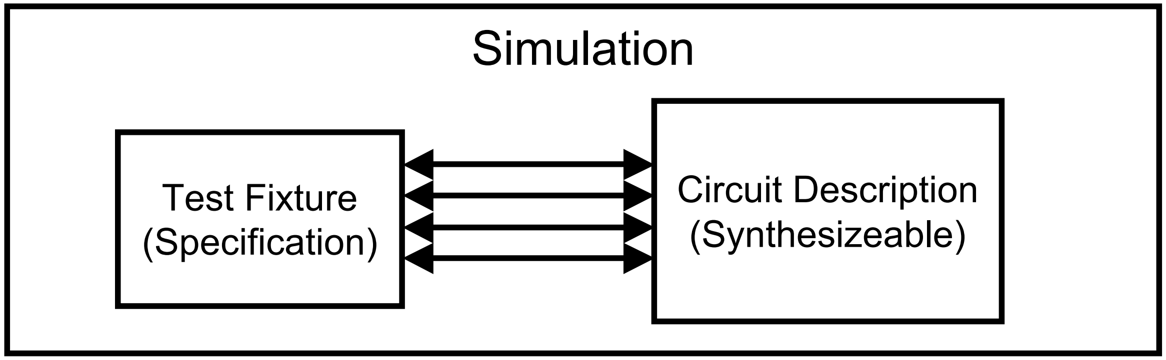
CSE467, Combinational Verilog

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

    structural
  - Dataflow
  - Algorithmic

    behavioral
- 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...

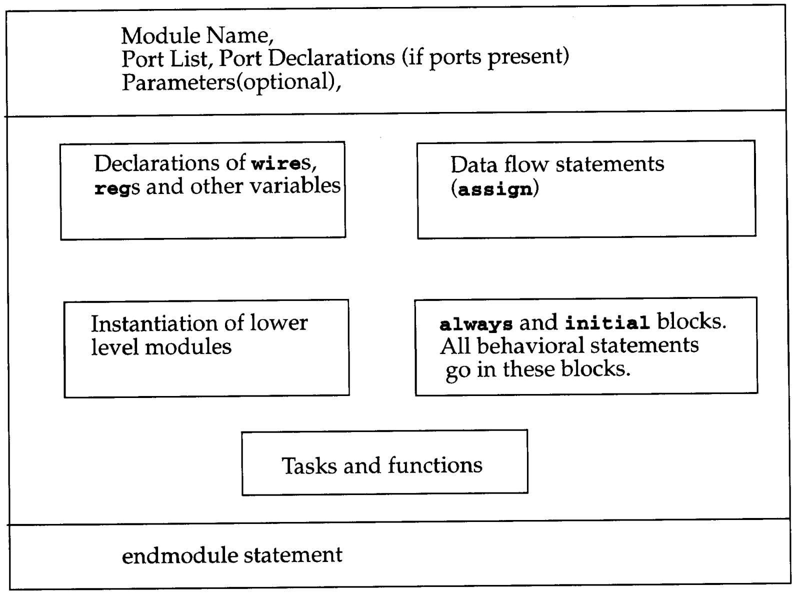
- Use a synthesizeable subset of Verilog
  - e.g. no “initial” blocks
  - Using structural and “synthesizeable” behavioral Verilog
- Will simulate your Verilog code
  - Use ActiveHDL
- Will synthesize your code
  - Use SimplifyPro
- Will map your netlist
  - ISE
- Will simulate your netlist
  - After 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
  - http://www.cs.washington.edu/education/courses/467/99au/admin/HardwareLab.html
  - http://www.europa.com/~celiac/ver_eda.html

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 ;

  assign {OVER, SUM[3:0]} = A[3:0] + B[3:0];

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

module name

ports

```
module full_addr (A, B, Cin, S, Cout);
  input     A, B, Cin;
  output    S, Cout;

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

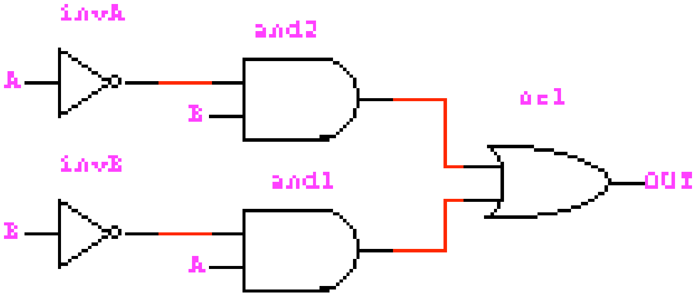
inputs/outputs

Structural Verilog

```
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  or1 (out, t1, t2);

endmodule
```



Note: Verilog is case sensitive
□ All keywords are lowercase

Structural full adder

```
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
```

Behavioral Verilog

```
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);
```

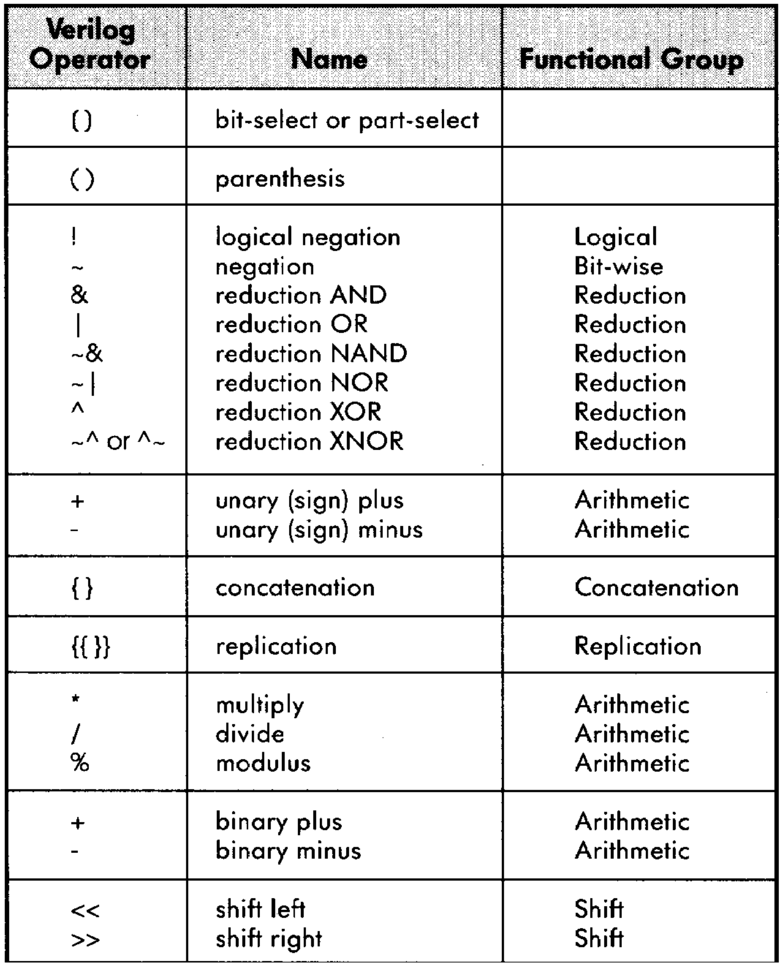
Data types

- Values on a wire
  - 0, 1, ***x*** (don’t care), ***z*** (tristate)
- Vectors
  - A[3:0]   vector of 4 bits: A[3], A[2], A[1], A[0]
    - An unsigned integer value
  - Concatenating bits/vectors
    - e.g. sign extend
      - B[7:0] = {A[3], A[3], A[3], A[3], A[3:0]};
      - B[7:0] = {4{A[3]}, A[3:0]};
  - Style: Use   a[7:0] = b[7:0] + c;
    Not   a = b + c;

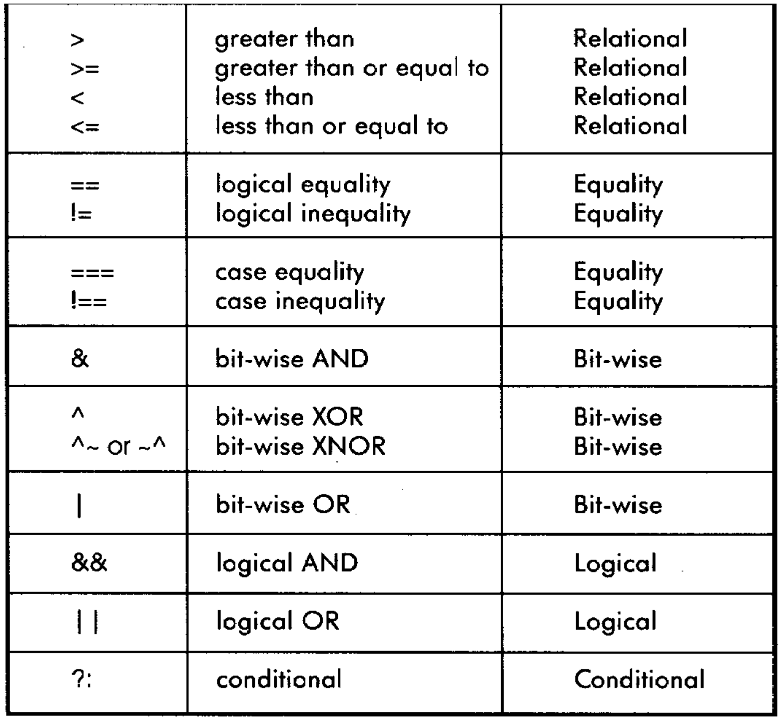
Numbers

- 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
  - C[4:0] = A[3:0] + B[3:0];
    - if A = 0110 (6) and B = 1010(–6), then C = 10000 (*not* 00000)
    - B is zero-padded, *not* sign-extended

Operators



Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
! ~ & | ~& ~| ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	Logical Bit-wise Reduction Reduction Reduction Reduction Reduction Reduction
+ -	unary (sign) plus unary (sign) minus	Arithmetic Arithmetic
{ }	concatenation	Concatenation
{{ }}	replication	Replication
* / %	multiply divide modulus	Arithmetic Arithmetic Arithmetic
+ -	binary plus binary minus	Arithmetic Arithmetic
<< >>	shift left shift right	Shift Shift



> >= < <=	greater than greater than or equal to less than less than or equal to	Relational Relational Relational Relational
== !=	logical equality logical inequality	Equality Equality
=== !==	case equality case inequality	Equality Equality
&	bit-wise AND	Bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	Bit-wise Bit-wise
|	bit-wise OR	Bit-wise
&&	logical AND	Logical
||	logical OR	Logical
?:	conditional	Conditional

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