Specifying digital circuits

- Schematics (what we've done so far)
  - Structural description
  - Describe circuit as interconnected elements
    - Build complex circuits using hierarchy
    - Large circuits are unreadable
- Hardware description languages (HDLs)
  - Structural and behavioral descriptions
    - Not programming languages
    - They are parallel languages tailored to digital design
  - Synthesize code to produce a circuit from descriptions
    - Easier to modify designs
    - Details of realization take care of by compiler

Hardware description languages

- Describe hardware at varying levels of abstraction
- Structural description
  - textual replacement for schematic
  - hierarchical composition of modules from primitives
- Behavioral/functional description
  - describe what module does, not how
  - synthesis generates circuit for module
- Simulation semantics

Hardware description languages (HDLs)

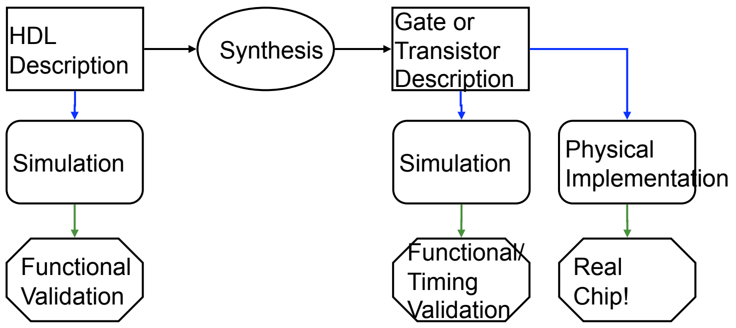
- Abel (~1983)
  - Developed by Data-I/O
  - Targeted to two-level logic (most popular at the time)
  - Limited capabilities
- Verilog (~1985) – IEEE standard
  - Developed by Gateway (now part of Cadence)
  - **Syntax** similar to C
  - Supports both two-level and multi-level logic
  - Moved to public domain in 1990
- VHDL (~1987) – IEEE standard
  - DoD-sponsored
  - Supports both two-level and multi-level logic
  - **Syntax** similar to Ada
- SystemC (late 90s) – developing into IEEE standard
  - Support mixed hardware/software systems
  - Not as widely accepted yet

Verilog

- Supports structural and behavioral descriptions
- Structural
  - explicit structure of the circuit
  - e.g., each logic gate instantiated and connected to others
- Behavioral
  - program describes input/output behavior of circuit
  - many structural implementations could have same behavior
  - e.g., different implementation of one Boolean function
- We'll mostly be using behavioral Verilog in Aldec ActiveHDL
  - rely on schematic when we want structural descriptions

Simulation and synthesis

- Simulation
  - "Execute" a design to verify correctness
- Synthesis
  - Generate a physical implementation from HDL code

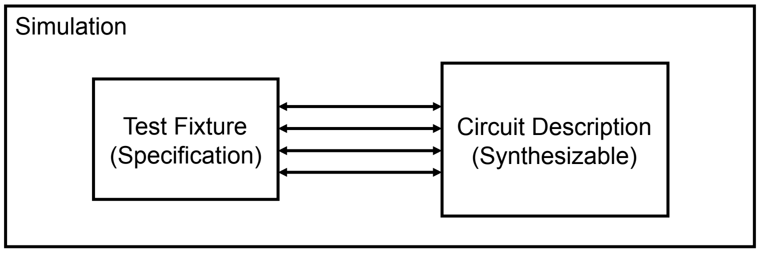


Simulation and synthesis (con't)

- Simulation
  - Models what a circuit does
    - Ignore implementation options
  - Can include static timing
  - Allows you to test design options at an abstract level
- Synthesis
  - Converts your code to a netlist (circuit)
    - Can simulate synthesized design
  - Tools map your netlist to the hardware you'll be using
- Simulation and synthesis in the CSE curriculum
  - CSE370: Learn simulation
  - CSE467: Learn synthesis

Simulation

- You provide an environment in which to test your circuit
  - Using non-circuit constructs
    - Active-HDL waveforms, read files, print
  - Using Verilog simulation code
    - A "test fixture"


Simulation
Test Fixture (Specification)
Circuit Description (Synthesizable)

Structural model

```
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
```

Simple behavioral model

- Continuous assignment

```
module xor_gate (out, a, b);
  input         a, b;
  output        out;
  reg           out;

  assign #6 out = a ^ b;

endmodule
```

simulation register - keeps track of value of signal

NOTE: anything on the left side of an assignment must have a "reg" declaration

delay from input change to output change

Simple behavioral model

- always block

```
module xor_gate (out, a, b);
  input         a, b;
  output        out;
  reg           out;

  always @(a or b) begin
     #6 out = a ^ b;
  end

endmodule
```

specifies when block is executed ie. triggered by which signals

NOTE: this "or" is not a Boolean OR, it just says: re-evaluate this expression whever a or b change

Driving a simulation through a “testbench”

```
module testbench (x, y);
  output        x, y;
  reg [1:0]     cnt;

  initial begin
    cnt = 0;
    repeat (4) begin
      #10 cnt = cnt + 1;
      $display ("@ time=%d, x=%b, y=%b, cnt=%b",
        $time, x, y, cnt); end
    #10 $finish;
  end

  assign x = cnt[1];
  assign y = cnt[0];
endmodule
```

2-bit vector

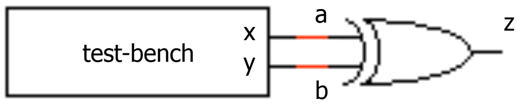
initial block executed only once at start of simulation

print to a console

directive to stop simulation

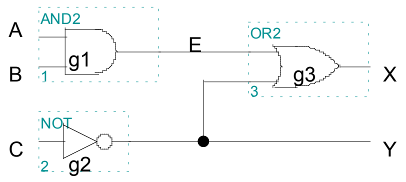
Complete simulation

- Instantiate stimulus component and device to test in a schematic



Specifying circuits in Verilog

- There are three major styles
  - Instances 'n wires
  - Continuous assignments
  - "always" blocks



"Structural"

```
wire E;
and g1(E,A,B);
not g2(Y,C);
or  g3(X,E,Y);
```

"Behavioral"

```
wire E;
assign E = A & B;
assign Y = ~C;
assign X = E | Y;
```

```
reg E, X, Y;
always @ (A or B or C)
begin
  E = A & B;
  Y = ~C;
  X = E | Y;
end
```

Data types

- Values on a wire
  - 0, 1, ***x*** (unknown or conflict), ***z*** (tri-state or unconnected)
- Vectors
  - A[3:0] vector of 4 bits: A[3], A[2], A[1], A[0]
    - Unsigned integer value
    - Indices must be constants
  - Concatenating bits/vectors (curly brackets on left or right side)
    - e.g. sign-extend
      - B[7:0] = {A[3], A[3], A[3], A[3], A[3:0]};
      - {4{A[3]}, A[3:0]} = B[7:0];
  - Style: Use a[7:0] = b[7:0] + c;
    Not a = b + c;
  - Bad style but legal syntax: C = &A[6:7]; // *and* of bits 6 and 7 of A

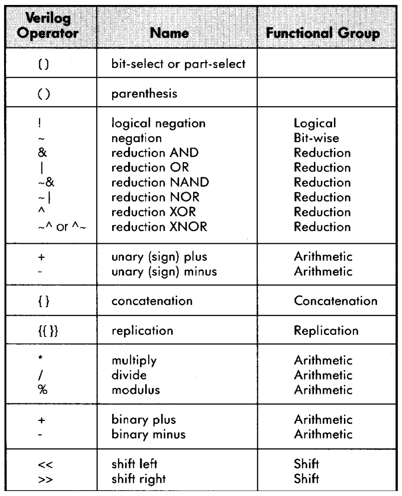
Data types that do not exist

- Structures
- Pointers
- Objects
- Recursive types
- (Remember, Verilog is not C or Java or Lisp or …!)

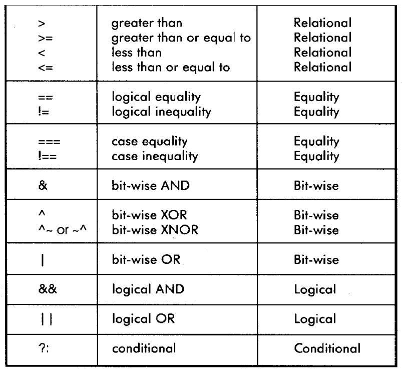
Numbers

- Format: <sign><size><base format><number>
- 14
    - Decimal number
- –4’b11
    - 4-bit 2’s complement binary of 0011 (is 1101)
- 12’b0000_0100_0110
    - 12-bit binary number (_ is ignored, just used for readibility)
- 3’h046
    - 3-digit (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	
!	logical negation	Logical
~	negation	Bit-wise
&	reduction AND	Reduction
|	reduction OR	Reduction
~&	reduction NAND	Reduction
~|	reduction NOR	Reduction
^	reduction XOR	Reduction
~^ or ^~	reduction XNOR	Reduction
+	unary (sign) plus	Arithmetic
-	unary (sign) minus	Arithmetic
{ }	concatenation	Concatenation
{{ }}	replication	Replication
*	multiply	Arithmetic
/	divide	Arithmetic
%	modulus	Arithmetic
+	binary plus	Arithmetic
-	binary minus	Arithmetic
<<	shift left	Shift
>>	shift right	Shift



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

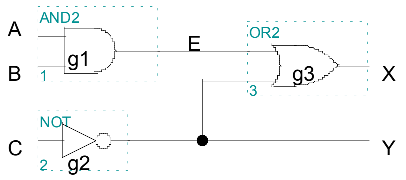
Similar to C operators

Two abstraction mechanisms

- Modules
  - More structural
  - Heavily used in 370 and “real” Verilog code
- Functions
  - More behavioral
  - Used often in “real” Verilog
  - Just used in test fixtures in 370

Basic building blocks: Modules

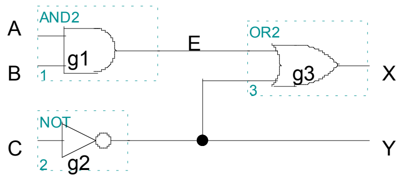
- Instantiated into a design
  - Not called like a procedure/method
- Illegal to nest module definitions
- Modules execute in parallel
- Names are case sensitive
- // for comments
- Name can't begin with a number
- Use wires for connections
- *and, or, not* are keywords
- All keywords are lower case
- Gate declarations (and, or, etc.)
  - List outputs first (convention), then inputs



```
// first simple example
module smpl (X,Y,A,B,C);
  input A,B,C;
  output X,Y;
  wire E
  and g1(E,A,B);
  not g2(Y,C);
  or  g3(X,E,Y);
endmodule
```

Modules are circuit components

- Module has ports
  - External connections
  - A, B, C, X, Y in this example
- Port types
  - input (A, B, C)
  - output (X, Y)
  - inout (tri-state) – more later
- Use assign statements for Boolean expressions
  - and ⇔ &
  - or ⇔ |
  - not ⇔ ~

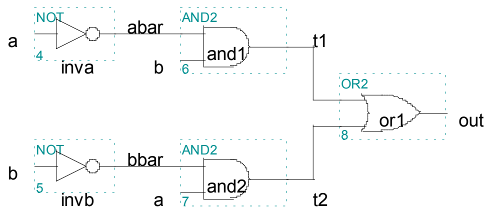


```
// previous example as a
// Boolean expression
module smpl2 (X,Y,A,B,C);
  input A,B,C;
  output X,Y;
  assign X = (A&B)|~C;
  assign Y = ~C;
endmodule
```

Structural Verilog

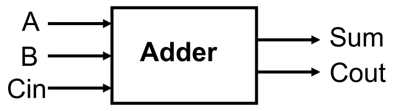
```
module xor_gate (out,a,b);
  input     a,b;
  output    out;
  wire      abar, bbar, t1, t2;
  not       inva (abar,a);
  not       invb (bbar,b);
  and       and1 (t1,abar,b);
  and       and2 (t2,bbar,a);
  or        or1 (out,t1,t2);
endmodule
```

8 basic gates (keywords):
and, or, nand, nor
buf, not, xor, xnor



Behavioral Verilog

- Describe circuit behavior
  - Not implementation



```
module full_addr (Sum,Cout,A,B,Cin);
  input     A, B, Cin;
  output    Sum, Cout;
  assign   {Cout, Sum} = A + B + Cin;
endmodule
```

{Cout, Sum} is a concatenation of 2 1-bit signals

Behavioral 4-bit adder

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

"[3:0] A" is a 4-wire bus labeled "A"
Bit 3 is the MSB
Bit 0 is the LSB

Can also write "[0:3] A"
Bit 0 is the MSB
Bit 3 is the LSB

Buses are implicitly connected
If you write BUS[3:2], BUS[1:0]
They become part of BUS[3:0]

Continuous assignment

- Assignment is continuously evaluated
  - Corresponds to a logic gate
  - Assignments execute in parallel

```
assign A = X | (Y & ~Z);

assign B[3:0] = 4'b01XX;

assign C[15:0] = 4'h00ff;

assign #3 {Cout, Sum[3:0]} = A[3:0] + B[3:0] + Cin;
```

Boolean operators (~ for bit-wise negation)

bits can assume four values (0, 1, X, Z)

variables can be n-bits wide (MSB:LSB)

arithmetic operator

Gate delay (used by simulator)

multiple assignment (concatenation)

Example: A comparator

```
module Compare1 (Equal, Alarger, Blarger, A, B);
  input     A, B;
  output    Equal, Alarger, Blarger;

  assign #5 Equal   = (A & B) |
                      (~A & ~B);
  assign #3 Alarger = (A & ~B);
  assign #3 Blarger = (~A & B);
endmodule
```

assign statement ordering doesn’t matter because they execute in parallel

Comparator example (con’t)

```
// Make a 4-bit comparator from 4 1-bit comparators

module Compare4(Equal, Alarger, Blarger, A4, B4);
  input [3:0] A4, B4;
  output Equal, Alarger, Blarger;
  wire E0, E1, E2, E3, AL0, AL1, AL2, AL3, BL0, BL1, BL2, BL3;

  Compare1 cp0(E0, AL0, BL0, A4[0], B4[0]);
  Compare1 cp1(E1, AL1, BL1, A4[1], B4[1]);
  Compare1 cp2(E2, AL2, BL2, A4[2], B4[2]);
  Compare1 cp3(E3, AL3, BL3, A4[3], B4[3]);

  assign #5  Equal   = (E0 & E1 & E2 & E3);
  assign #10 Alarger = (AL3 | (AL2 & e3) |
                       (AL1 & E3 & E2) |
                       (AL0 & E3 & E2 & E1));
  assign #3  Blarger = (~Alarger & ~Equal);
endmodule
```

Sequential `assign`s don't make any sense

```
assign A = X | (Y & ~Z);

assign B = W | A;

assign A = Y & Z;
```

"Reusing" a variable on the LHS in multiple assign statements is not allowed – they execute in parallel – indeterminate result

Always Blocks

```
reg A, B, C;

always @ (W or X or Y or Z)
begin
  A = X | (Y & ~Z);
  B = W | A;
  A = Y & Z;
  if (A & B) begin
    B = Z;
    C = W | Y;
  end
end
```

Variables that appear on the left hand side in an always block must be declared as "reg"s

Sensitivity list

Statements in an always block are executed in sequence

All variables must be assigned on every possible path through the code!!!
- otherwise, the simulator gets confused and decides it needs memory (the dreaded "inferred latch") so it can remember the old value it had

Functions

- Use functions for complex combinational logic

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

  assign out = myfunction(in1, in2);

  function myfunction;
   input in1, in2;
   begin
     myfunction = in1 & in2;
   end
  endfunction

endmodule
```

Benefit:
Compiler will fail if function does not generate a result

Verilog tips

- **Do not** write C-code
  - Think hardware, not algorithms
    - Verilog is **inherently parallel**
    - Compilers don't map algorithms to circuits well
- Do describe hardware circuits
  - First draw a dataflow diagram
  - Then start coding
- References
  - Tutorial and reference manual are found in ActiveHDL help
  - Wikipedia – search for "Verilog" – tutorials and external resources
    - Sutherland quick reference guide
  - "Starter's Guide to Verilog 2001" by Michael Ciletti – copies for borrowing in hardware lab

Hardware description languages vs. programming languages

- Program structure
  - instantiation of multiple components of the same type
  - specify interconnections between modules via schematic
  - hierarchy of modules (only leaves can be HDL)
- Assignment
  - continuous assignment (logic always computes)
  - propagation delay (computation takes time)
  - timing of signals is important (when does computation have its effect)
- Data structures
  - size explicitly spelled out - no dynamic structures
  - no pointers
- Parallelism
  - hardware is naturally parallel (must support multiple threads)
  - assignments can occur in parallel (not just sequentially)

Hardware description languages and combinational logic

- Modules - specification of inputs, outputs, bidirectional, and internal signals
- Continuous assignment - a gate’s output is a function of its inputs at all times (doesn’t need to wait to be "called")
- Propagation delay- concept of time and delay in input affecting gate output
- Composition - connecting modules together with wires
- Hierarchy - modules encapsulate functional blocks