Lecture 8: Combinational Verilog

CSE 370, Autumn 2007
Benjamin Ylvisaker

Where We Are

- Last lecture: Minimization with K-maps
- This lecture: Combinational Verilog
- Next lecture: ROMs, PLAs and PALs, oh my!
- Homework 3 ongoing
- Lab 2 done; lab 3 next week

Specifying Circuits

- Schematics
  - Structural description
  - Build more complex circuits using hierarchy
  - Large circuits are unreadable
- HDLs (Hardware description languages)
  - Not conventional programming languages
  - Very restricted parallel languages
  - Synthesize code to produce a circuit

Quick History Lesson

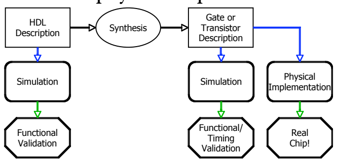
- Abel (~1983)
  - Developed by Data-I/O
  - Targeted to PLDs
- Verilog (~1985)
  - Developed by Gateway (now part of Cadence)
  - **Syntax** similar to C
  - Moved to public domain in 1990
- VHDL (~1987)
  - DoD sponsored
  - **Syntax** similar to Ada

Verilog and VHDL Dominant

- Both "IEEE standard" languages
- Most tools support both
- Verilog is "simpler"
  - Less, more concise syntax
- VHDL is more structured
  - More sophisticated type system
  - Better modularity features

Simulation and Synthesis

- Simulation
  - "Execute" a design with some test data
- Synthesis
  - Generate a physical implementation

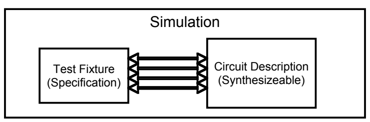


Simulation and Synthesis (cont'd)

- Simulation
  - Model circuit behavior
  - Can include timing estimates
  - Allows for easier design exploration
- Synthesis
  - Converts HDL code to "netlists"
  - Can still simulate the generated netlists
- Simulation and synthesis in the CSE curriculum
  - 370: Learn simulation
  - 467: Learn something about synthesis

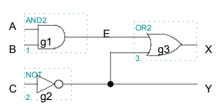
Simulation

- You provide an environment
  - Use non-circuit constructs (Active-HDL waveforms, random number generators, etc)
  - Can write arbitrary Verilog code



Specifying Circuits in Verilog

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



"Structural" "Behavioral"

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

```
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 (unconnected)
- Vectors
  - A[3:0] vector of 4 bits: A[3], A[2], A[1], A[0]
    - Interpreted as an unsigned binary number
    - Indices must be constants
  - Concatenation
    - B = {A[3], A[3], A[3], A[3], A[3:0]};
    - B = {4{A[3]}, A[3:0]};
  - Style: good to use unnecessary size specs sometimes
    - a[7:0] = b[7:0] + c[7:0];
  - Built-in reductions: C = &A[5:7];

Data Types That Do Not Exist

- structures (records)
- Pointers
- Objects
- Recursive types
- (Remember, Verilog is not C or Java or Lisp or ...)

Numbers

- Format: <sign><size><base format><number>
- 14
  - Decimal
- -4'b11
  - 4-bit 2's complement of 0011
- 12'b000_0100_0110
  - 12 bit binary number (_'s ignored)
- 12'h4Ab
  - 12 bit hexadecimal number

Operators

<table>
<tr><th>Verilog Operator</th><th>Name</th><th>Functional Group</th></tr>
<tr><td>[ ]</td><td>bit-select or part-select</td><td></td></tr>
<tr><td>( )</td><td>parenthesis</td><td></td></tr>
<tr><td>!
~
&
|
~&
~|
^
~^ or ^~</td><td>logical negation
negation
reduction AND
reduction OR
reduction NAND
reduction NOR
reduction XOR
reduction XNOR</td><td>Logical
Bit-wise
Reduction
Reduction
Reduction
Reduction
Reduction
Reduction</td></tr>
<tr><td>+
-</td><td>unary (sign) plus
unary (sign) minus</td><td>Arithmetic
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
divide
modulus</td><td>Arithmetic
Arithmetic
Arithmetic</td></tr>
<tr><td>+
-</td><td>binary plus
binary minus</td><td>Arithmetic
Arithmetic</td></tr>
<tr><td><<
>></td><td>shift left
shift right</td><td>Shift
Shift</td></tr>
</table>

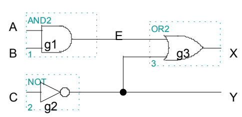
<table>
<tr><td>>
>=
<
<=</td><td>greater than
greater than or equal to
less than
less than or equal to</td><td>Relational
Relational
Relational
Relational</td></tr>
<tr><td>==
!=</td><td>logical equality
logical inequality</td><td>Equality
Equality</td></tr>
<tr><td>===
!==</td><td>case equality
case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

Similar to C operators

Two Abstraction Mechanisms

- Modules
  - More structural
  - Heavily used in 370 and "real" Verilog code
- Functions
  - More behavioral
  - Used to some extent in "real" Verilog, but not much in 370

Basic Building Blocks: Modules

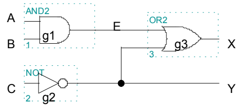


- Instantiated, not called
- Illegal to nest module defs
- Instances "execute" in parallel
- Wires are used for connections
- and, or, not built-in primitive modules
  - List output first
  - Arbitrary number of inputs next
- Names are case sensitive
  - Cannot begin with number
- // for comments

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

Module Ports

- Modules interact with the rest of a design through ports
  - input
  - output
  - inout



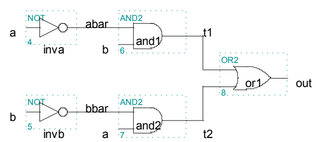
- Same example with continuous assignments:

```
// first simple example
module smpl(X,Y,A,B,C);
  input A,B,C;
  output X,Y;
  assign X = (A&B)|~C;
  assign Y = ~C;
endmodule
```

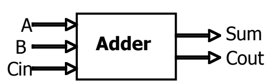

Bigger Structural Example

```
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 built-in gates:
and, or, nand, nor,
buf, not, xor, xnor



Behavioral Full Adder




```
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

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
```


Continuous Assignment

- Continuously evaluated
  - Think of them as collections of logic gates
  - Evaluated 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;
```


Hierarchy Example: Comparator

•

```
module Compare1 (Equal, Alarger, Blarger, A, B);
  input  A, B;
  output Equal, Alarger, Blarger;
  assign Equal = (A & B) | (~A & ~B);
  assign Alarger = (A & ~B);
  assign Blarger = (~A & B);
endmodule
```

4-bit Comparator

```
// 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, B10, B11, B12, B13;

  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 Equal = (e0 & e1 & e2 & e3);
  assign Alarger = (Al3 | (Al2 & e3) |
                   (Al1 & e3 & e2) |
                   (Al0 & e3 & e2 & e1));
  assign Blarger = (~Alarger & ~Equal);
endmodule
```


Sequential assigns don't make any sense

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

assign B = W | A;

assign A = Y & Z;
```

- You can't reassign a variable with continuous assignments

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 control path!!! (otherwise you get the dreaded "inferred latch")

Functions

- Functions can be used for combinational logic that you want to reuse

```
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
```


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
  - And in today's reading assignment
  - "Starter's Guide to Verilog 2001" by Michael Ciletti
    - copies for borrowing in hardware lab

Thank You for Your Attention

Thank You for Your Attention

- Read lab 2
- Continue homework 2
- Continue reading the book