Combinational Logic Design with Verilog

ECE 152A – Winter 2012
Reading Assignment

- Brown and Vranesic
  - 2 Introduction to Logic Circuits
    - 2.10 Introduction to Verilog
      - 2.10.1 Structural Specification of Logic Circuits
      - 2.10.2 Behavioral Specification of Logic Circuits
      - 2.10.3 How Not to Write Verilog Code
Reading Assignment

- Brown and Vranesic (cont) 1st edition only!
  - 4 Optimized Implementation of Logic Functions
    - 4.12 CAD Tools
      - 4.12.1 Logic Synthesis and Optimization
      - 4.12.2 Physical Design
      - 4.12.3 Timing Simulation
      - 4.12.4 Summary of Design Flow
      - 4.12.5 Examples of Circuits Synthesized from Verilog Code
Programmable Logic

- Provides low cost and flexibility in a design
  - Replace multiple discrete gates with single device
  - Logical design can be changed by reprogramming the device
    - No change in board design
  - Logical design can be changed even after the part has been soldered onto the circuit board in modern, In-system programmable device
- Inventory can focus on one part
  - Multiple uses of same device
Programmable Logic

- Evolution of Programmable Logic
  - Both in time and complexity
  - ROM’s and RAM’s
    - Not strictly programmable logic, but useful in implementing combinational logic and state machines
  - PAL’s
    - PAL’s – Programmable Array Logic
    - PLA’s – Programmable Logic Array
    - GAL’s – Generic Logic Array
Programmable Logic

- **PLD’s**
  - Programmable Logic Device
  - PLDs are (in general) advanced PALs

- **CPLD’s**
  - Complex Programmable Logic Device
  - Multiple PLDs on a single chip

- **FPGA’s**
  - Field Programmable Gate Array
Design Entry

- In previous examples, design entry is schematic based
  - TTL implementation using standard, discrete integrated circuits
  - PLD implementation using library of primitive elements
- Code based design entry uses a hardware description language (HDL) for design entry
  - Code is synthesized and implemented on a PLD
Verilog Design

- Structural Verilog
  - Looks like the gate level implementation
    - Specify gates and interconnection
  - Text form of schematic
    - Referred to as “netlist”
  - Allows for “bottom – up” design
    - Begin with primitives, instantiate in larger blocks
Verilog Design

- RTL (Register Transfer Level) Verilog
  - Allows for “top – down” design
  - No gate structure or interconnection specified
  - Synthesizable code (by definition)
    - Emphasis on synthesis, not simulation
      - vs. high level behavioral code and test benches
    - No timing specified in code
    - No initialization specified in code
      - Timing, stimulus, initialization, etc. generated in testbench (later)
Recall Half Adder description from schematic based design example

- Operation
- Truth table
- Circuit
- Graphical symbol
Verilog Syntax

- Modules are the basic unit of Verilog models
  - Functional Description
    - Unambiguously describes module’s operation
      - Functional, i.e., without timing information
  - Input, Output and Bidirectional ports for interfaces
  - May include instantiations of other modules
    - Allows building of hierarchy
Verilog Syntax

- Module declaration
  - `module ADD_HALF (s,c,x,y);`
  - Parameter list is I/O Ports

- Port declaration
  - Can be input, output or inout (bidirectional)
    - `output s,c;`
    - `input x,y;`
Verilog Syntax

- Declare nodes as wires or reg
  - Wires assigned to declaratively
  - Reg assigned to procedurally
    - More on this later
  - In a combinational circuit, all nodes can, but don’t have to be, declared wires
    - Depends on how code is written
  - Node defaults to wire if not declared otherwise
  - `wire s,c,x,y;`
Verilog Syntax

- Gates and interconnection
  - xor G1(s,x,y);
  - and G2(c,x,y);
  - Verilog gate level primitive
    - Gate name
  - Internal (local) name
    - Instance name
  - Parameter list
    - Output port, input port, input port…
**Gate Instantiation**

- **Verilog Gates**
  - **Note:** *notif* and *bufif* are tri-state gates

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>$f = (a \cdot b \cdot \ldots)$</td>
<td>and $(f, a, b, \ldots)$</td>
</tr>
<tr>
<td>nand</td>
<td>$f = (a \cdot b \cdot \ldots)$</td>
<td>nand $(f, a, b, \ldots)$</td>
</tr>
<tr>
<td>or</td>
<td>$f = (a + b + \ldots)$</td>
<td>or $(f, a, b, \ldots)$</td>
</tr>
<tr>
<td>nor</td>
<td>$f = (a + b + \ldots)$</td>
<td>nor $(f, a, b, \ldots)$</td>
</tr>
<tr>
<td>xor</td>
<td>$f = (a \oplus b \oplus \ldots)$</td>
<td>xor $(f, a, b, \ldots)$</td>
</tr>
<tr>
<td>xnor</td>
<td>$f = (a \odot b \odot \ldots)$</td>
<td>xnor $(f, a, b, \ldots)$</td>
</tr>
<tr>
<td>not</td>
<td>$f = \overline{a}$</td>
<td>not $(f, a)$</td>
</tr>
<tr>
<td>buf</td>
<td>$f = a$</td>
<td>buf $(f, a)$</td>
</tr>
<tr>
<td>notif0</td>
<td>$f = (</td>
<td>e \cdot \overline{a} : 'bz)$</td>
</tr>
<tr>
<td>notif1</td>
<td>$f = (e \cdot \overline{a} : 'bz)$</td>
<td>notif1 $(f, a, e)$</td>
</tr>
<tr>
<td>bufif0</td>
<td>$f = (</td>
<td>e \cdot a : 'bz)$</td>
</tr>
<tr>
<td>bufif1</td>
<td>$f = (e \cdot a : 'bz)$</td>
<td>bufif1 $(f, a, e)$</td>
</tr>
</tbody>
</table>
Verilog Syntax

- Close the module definition with `endmodule`
- Comments begin with `//`
module ADD_HALF (s, c, x, y);

output s, c;
input x, y;

wire s, c, x, y;
   // this line is optional since nodes default to wires

xor G1 (s, x, y);  // instantiation of XOR gate
and G2 (c, x, y);  // instantiation of AND gate

dendmodule
# Half Adder - PLD Implementation

## Functional Simulation

<table>
<thead>
<tr>
<th>Input</th>
<th>0+0</th>
<th>0+1</th>
<th>1+0</th>
<th>1+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name:</td>
<td></td>
<td>20.0ns</td>
<td>40.0ns</td>
<td>60.0ns</td>
</tr>
<tr>
<td>[I] x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[I] y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[O]s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[O]c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>00</td>
<td>01</td>
<td>01</td>
<td>10</td>
</tr>
</tbody>
</table>
Full Adder – Structural Verilog Design

- Recall Full Adder description from schematic based design example
  - Truth table
  - Karnaugh maps
  - Circuit

![Truth Table][Truth Table]

![Karnaugh Maps][Karnaugh Maps]

![Circuit][Circuit]
Full Adder from 2 Half Adders

(a) Block diagram

(b) Detailed diagram

Figure 5.5  A decomposed implementation of the full-adder circuit.
module ADD_FULL (s,cout,x,y,cin);

output s,cout;
input x,y,cin;

//internal nodes also declared as wires
wire cin,x,y,s,cout,s1,c1,c2;

ADD_HALF HA1(s1,c1,x,y);
ADD_HALF HA2(s,c2,cin,s1);
or (cout,c1,c2);

endmodule
### Full Adder - PLD Implementation

#### Functional Simulation

<table>
<thead>
<tr>
<th>Input</th>
<th>0+0+0</th>
<th>0+1+0</th>
<th>1+0+0</th>
<th>1+1+0</th>
<th>1+1+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0+0+1</td>
<td>0+1+1</td>
<td>1+0+1</td>
<td>1+1+0</td>
<td>1+1+1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>20.0ns</th>
<th>40.0ns</th>
<th>60.0ns</th>
<th>80.0ns</th>
<th>100.0ns</th>
<th>120.0ns</th>
<th>140.0ns</th>
<th>160.0ns</th>
<th>180.0ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ci</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>00</td>
<td>01</td>
<td>01</td>
<td>10</td>
<td>01</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>co</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Verilog language includes a large number of logical and arithmetic operators. Bit length column indicates width of result.

### Verilog Operators

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
<th>Bit Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwise</td>
<td>$\sim A$, $+A$, $-A$, $A &amp; B$, $A \vert B$, $A \wedge B$, $A \wedge B$, $A \oplus B$</td>
<td>$L(A)$, $MAX(L(A), L(B))$</td>
</tr>
<tr>
<td>Logical</td>
<td>$\land$, $\lor$, $A | B$</td>
<td>1 bit</td>
</tr>
<tr>
<td>Reduction</td>
<td>$&amp; A$, $\sim &amp; A$, $\vert A$, $\sim \vert A$, $\wedge \vert A$, $\sim \wedge A$</td>
<td>1 bit</td>
</tr>
<tr>
<td>Relational</td>
<td>$A \equiv B$, $A' = B$, $A &gt; B$, $A &lt; B$</td>
<td>1 bit</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>$A + B$, $A - B$, $A \times B$, $A / B$</td>
<td>$MAX(L(A), L(B))$</td>
</tr>
<tr>
<td>Shift</td>
<td>$A &lt; B$, $A &gt;&gt; B$</td>
<td>$L(A)$</td>
</tr>
<tr>
<td>Concatenate</td>
<td>$A, \ldots, B$</td>
<td>$L(A) + \cdots + L(B)$</td>
</tr>
<tr>
<td>Replication</td>
<td>$[B\vert A]$</td>
<td>$B + L(A)$</td>
</tr>
<tr>
<td>Condition</td>
<td>$A?B:C$</td>
<td>$MAX(L(B), L(C))$</td>
</tr>
</tbody>
</table>
Behavioral Specification of Logic Circuits

Continuous Assignment Operator

- assign sum = a ^ b;
  - “Assign” to a wire (generated declaratively)
  - Equivalent to
    - xor (sum,a,b);
- Continuous and concurrent with other wire assignment operations
  - If a or b changes, sum changes accordingly
  - All wire assignment operations occur concurrently
    - Order not specified (or possible)
module ADD_FULL_RTL (sum, cout, x, y, cin);

    output sum, cout;
    input x, y, cin;

    // declaration for continuous assignment
    wire cin, x, y, sum, cout;

    // logical assignment
    assign sum = x ^ y ^ cin;
    assign cout = x & y | x & cin | y & cin;

endmodule
module ADD_FULL_RTL (sum, cout, x, y, cin);

    output sum, cout;
    input x, y, cin;

    // declaration for continuous assignment
    wire cin, x, y, sum, cout;

    // concatenation operator and addition
    assign {cout, sum} = x + y + cin;

endmodule
Procedural Verilog Statements

Recall:

- Wires assigned to declaratively
  - Continuous / concurrent assignment
- Reg “variables” assigned to procedurally
  - Value is “registered” until next procedural assignment
    - Continuous assignment (wires) occurs immediately on input change
  - Enables clocked (synchronous) timing
Procedural Verilog Statements

- The “always” block
  - Syntax is “always at the occurrence (@) of any event on the sensitivity list, execute the statements inside the block (in order)”

```verilog
always @ (x or y or cin)
{
  cout, sum} = x + y + cin;
```
module ADD_FULL_RTL (sum, cout, x, y, cin);

output sum, cout;
input x, y, cin;

// declaration for behavioral model
wire cin, x, y;
reg sum, cout;

// behavioral specification
always @ (x or y or cin)
    {cout, sum} = x + y + cin;

endmodule
Two-bit, Ripple Carry Adder – Structural Verilog

module TWO_BIT_ADD (S,X,Y,cin,cout);

input cin;
input [1:0]X,Y;  // vectored input
output [1:0]S;   // and output signals
output cout;

wire cinternal;

ADD_FULL AF0(S[0],cinternal,X[0],Y[0],cin);
ADD_FULL AF1(S[1],cout,X[1],Y[1],cinternal);

endmodule
**Two-bit, Ripple Carry Adder – PLD Implementation**

- **Functional Simulation**
  - Base-4 Bus Representation of X, Y and Sum

![Functional Simulation Table]

<table>
<thead>
<tr>
<th>Name</th>
<th>100.0ns</th>
<th>200.0ns</th>
<th>300.0ns</th>
<th>400.0ns</th>
<th>500.0ns</th>
<th>600.0ns</th>
<th>700</th>
</tr>
</thead>
<tbody>
<tr>
<td>cin</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>1 2 3</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0</td>
</tr>
<tr>
<td>cout</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0 1 2</td>
<td>0</td>
</tr>
</tbody>
</table>

- $0 + 1 + 3 = 4 = 10_4$ →
- $1 + 2 + 2 = 5 = 11_4$ →
- $0 + 3 + 0 = 3 = 03_4$ →
- $1 + 3 + 3 = 7 = 13_4$ →
Verilog Test Bench

- Device Under Test (DUT)
  - Circuit being designed/developed
    - Full adder for this example

- Testbench
  - Provides stimulus to DUT
    - Like test equipment on a bench

- Instantiate DUT in testbench
  - Generate all signals in testbench
  - No I/O (parameter list) in testbench
module ADDFULL_TB;

reg a, b, ci;
wire sum, co;

initial begin
    a = 0;
    b = 0;
    ci = 0;
end

always begin
    #5 a = ~a;
end

always begin
    #10 b = ~b;
end

always begin
    #20 ci = ~ci;
end

    ADD_FULL AF1(sum, co, a, b, ci);
endmodule