

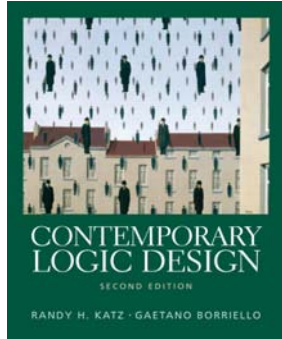
# Administrivia

- Homework 3 due Friday

# CSE 370 Spring 2006

## Introduction to Digital Design

### Lecture 8: Introduction to Verilog



#### Last Lecture

- Design Examples a K-maps
- Minimization Algorithm

#### Today

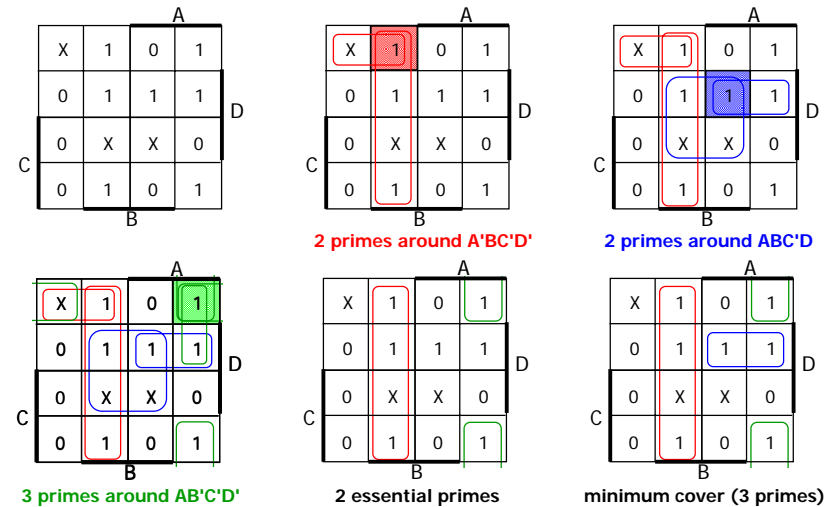
- Introduction to Verilog

## Algorithm for two-level simplification

- Algorithm: minimum sum-of-products expression from a Karnaugh map

- Step 1: choose an element of the ON-set
- Step 2: find "maximal" groupings of 1s and Xs adjacent to that element
  - consider top/bottom row, left/right column, and corner adjacencies
  - this forms prime implicants (number of elements always a power of 2)
- Repeat Steps 1 and 2 to find all prime implicants
- Step 3: revisit the 1s in the K-map
  - if covered by single prime implicant, it is essential, and participates in final cover
  - 1s covered by essential prime implicant do not need to be revisited
- Step 4: if there remain 1s not covered by essential prime implicants
  - select the smallest number of prime implicants that cover the remaining 1s

## Algorithm for two-level simplification (example)



## Visit All in the On Set?

		A	
	0	0	1
	0	1	1
	0	1	1
	0	0	1
C		B	

## Activity

- List all prime implicants for the following K-map:

		A	
	X	0	X
	0	1	X
	0	X	X
	X	1	1
C		B	

- Which are essential prime implicants?
- What is the minimum cover?

## Loose end: POS minimization using k-maps

- Using k-maps for POS minimization
  - Encircle the zeros in the map
  - Interpret indices complementary to SOP form

		A			
	AB	00	01	11	10
	00	1	0	0	1
	01	0	1	0	0
	11	1	1	1	1
	10	1	1	1	1
C		B			

$$F = (B'+C+D)(B+C+D')(A'+B'+C)$$

Check using de Morgan's on SOP

$$F' = BC'D' + B'C'D + ABC'$$

$$(F')' = (BC'D' + B'C'D + ABC')'$$

$$(F')' = (BC'D')' + (B'C'D)' + (ABC')'$$

$$F = (B'+C+D)(B+C+D')(A'+B'+C)$$

## Ways of specifying circuits

- Schematics
  - 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
  - Moved to public domain in 1990
- VHDL (~1987)
  - DoD sponsored
  - Similar to Ada

# Verilog

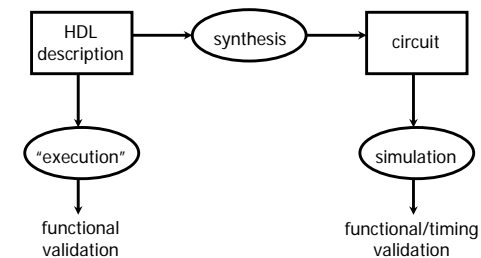
```
module toplevel(clock,reset);  
  
input clock;  
input reset;  
reg flop1;  
reg flop2;  
  
always @ (posedge reset or posedge clock)  
if (reset)  
begin  
flop1 <= 0;  
flop2 <= 1;  
end  
else  
begin  
flop1 <= flop2;  
flop2 <= flop1;  
end  
  
endmodule
```

# Verilog versus VHDL

- Both “IEEE standard” languages
- Most tools support both
- Verilog is “simpler”
  - Less syntax, fewer constructs
- VHDL is more structured
  - Can be better for large, complex systems
  - Better modularization

# Simulation versus synthesis

- Simulation
  - “Execute” a design to verify correctness
- Synthesis
  - Generate a netlist from HDL code



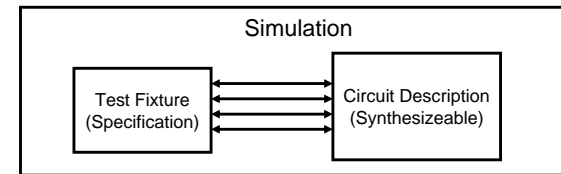
## Simulation versus synthesis (con't)

- Simulation
  - Models what a circuit does
    - Multiply is "\*", ignoring implementation options
  - Can include static timing
  - Allows you to test design options
- Synthesis
  - Converts your code to a netlist
    - Can simulate synthesized design
  - Tools map your netlist to hardware
- Verilog and VHDL simulate and synthesize
  - CSE370: Learn simulation
  - CSE467 (Advanced Digital Design): Learn synthesis

## Simulation

- You provide an environment
  - Using non-circuit constructs
    - Read files, print, control simulation
  - Using Verilog simulation code
    - A "test fixture"

Note: We will ignore timing and test benches until next Verilog lecture



## Levels of abstraction

- Verilog supports 4 description levels
  - Switch
  - Gate  $\left. \begin{array}{l} \text{Switch} \\ \text{Gate} \end{array} \right\} \textit{structural}$
  - Dataflow
  - Algorithmic  $\left. \begin{array}{l} \text{Dataflow} \\ \text{Algorithmic} \end{array} \right\} \textit{behavioral}$
- Can mix & match levels in a design
- Designs that combine dataflow and algorithmic constructs and synthesis are called RTL
  - *Register Transfer Level*

## Structural versus behavioral Verilog

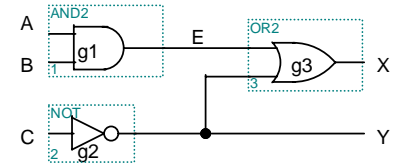
- Structural
  - Describe explicit circuit elements
  - Describe explicit connections between elements
    - Connections between logic gates
  - Just like schematics, but using text
- Behavioral
  - Describe circuit as algorithms/programs
    - What a component does
    - Input/output behavior
  - Many possible circuits could have same behavior
    - Different implementations of a Boolean function

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

# Basic building blocks: Modules

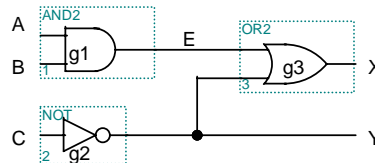
- Instantiated into a design
  - Never called
- Illegal to nest module defs.
- Modules execute in parallel
- Names are case sensitive
- // for comments
- Name can't begin with a number
- Use wires for connections
- and, or, not are keywords
- All keywords are lower case
- Gate declarations (and, or, etc)
  - List outputs first
  - Inputs second



```
// 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 example
- Port types
  - input
  - output
  - inout (tristate)
- Use assign statements for Boolean expressions
  - and ⇔ &
  - or ⇔ |
  - not ⇔ ~

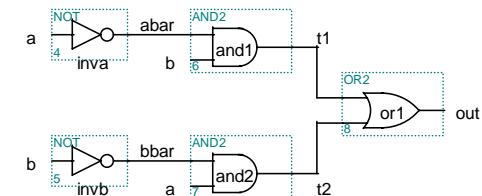


```
// previous example as a
// Boolean expression
module smpl2 (X,Y,A,B,C);
  input A,B,C;
  output X,Y;
  assign X = (A&B) | ~C;
  assign Y = ~C;
endmodule
```

# 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

## 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]

## Data types

- Values on a wire
  - 0, 1, **x** (don't care), **z** (tristate 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
    - 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;
  - Legal syntax: C = &A[6:7]; // logical *and* of bits 6 and 7 of A

## 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)
- 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	
! ~ &   ~& ~  ^ ~^ 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	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

# Continuous assignment

- Assignment is continuously evaluated

- Corresponds to a logic gate
- Assignments execute in parallel

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

Boolean operators  
(~ for bit-wise negation)

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

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

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

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

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

Gate delay (used by simulator)

multiple assignment (concatenation)

arithmetic operator

# Example: A 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
```

Top-down design and bottom-up design are both okay  
 ⇒ module ordering doesn't matter  
 ⇒ because modules 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, A10, A11, A12, A13, B10, B11, B12, B13;

    Compare1 cp0(e0, A10, B10, A4[0], B4[0]);
    Compare1 cp1(e1, A11, B11, A4[1], B4[1]);
    Compare1 cp2(e2, A12, B12, A4[2], B4[2]);
    Compare1 cp3(e3, A13, B13, A4[3], B4[3], );

    assign Equal = (e0 & e1 & e2 & e3);
    assign Alarger = (A13 | (A12 & e3) |
                    (A11 & e3 & e2) |
                    (A10 & e3 & e2 & e1));
    assign Blarger = (~Alarger & ~Equal);
endmodule
```

# 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:**  
 Functions force a result  
 ⇒ Compiler will fail if function does not generate a result

# Sequential Verilog-- Blocking and non-blocking assignments

- Blocking assignments ( $Q = A$ )
  - Variable is assigned immediately
  - New value is used by subsequent statements
- Non-blocking assignments ( $Q <= A$ )
  - Variable is assigned after all scheduled statements are executed
  - Value to be assigned is computed but saved for later
  - Usual use: Register assignment
  - Registers simultaneously take new values after the clock edge

```

Example: Swap
always @(posedge CLK)
begin
  temp = B;
  B = A;
  A = temp;
end

always @(posedge CLK)
begin
  A <= B;
  B <= A;
end

```

# Sequential Verilog-- Assignments- watch out!

- Blocking versus Non-blocking

```

reg B, C, D;

always @(posedge clk)
begin
  B = A;
  C = B;
  D = C;
end

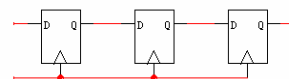
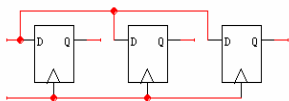
```

```

reg B, C, D;

always @(posedge clk)
begin
  B <= A;
  C <= B;
  D <= C;
end

```



# Summary of two-level combinational-logic

- Logic functions and truth tables
  - AND, OR, Buf, NOT, NAND, NOR, XOR, XNOR
  - Minimal set
- Axioms and theorems of Boolean algebra
  - Proofs by re-writing
  - Proofs by perfect induction (fill in truth table)
- Gate logic
  - Networks of Boolean functions
  - NAND/NOR conversion and de Morgan's theorem
- Canonical forms
  - Two-level forms
  - Incompletely specified functions (don't cares)
- Simplification
  - Two-level simplification (K-maps)



# Solving combinational design problems

- Step 1: Understand the problem
  - Identify the inputs and outputs
  - Draw a truth table
- Step 2: Simplify the logic
  - Draw a K-map
  - Write a simplified Boolean expression
    - SOP or POS
    - Use don't cares
- Step 3: Implement the design
  - Logic gates and/or Verilog