Recap: Combinational v.s. sequential logic

• Combinational logic
  • The output is a pure function of its current inputs
  • The output doesn’t change regardless how many times the logic is triggered — Idempotent

• Sequential logic
  • The output depends on current inputs, previous inputs, their history
Recap: Basic Boolean Algebra Concepts

• \{0, 1\}: The only two possible values in inputs/outputs
• Basic operators
  • \text{AND (\(\cdot\)) — } a \cdot b
    • returns 1 only if both \(a\) and \(b\) are 1s
    • otherwise returns 0
  • \text{OR (+) — } a + b
    • returns 1 if \(a\) \textbf{or} \(b\) is 1
    • returns 0 if none of them are 1s
  • \text{NOT (’) — } a’
    • returns 0 if \(a\) is 1
    • returns 1 if \(a\) is 0
Recap: Definitions of Boolean Function Expressions

- Complement: variable with a bar over it or a ‘ — $A', B', C'$
- Literal: variable or its complement — $A, A', B, B', C, C'$
- Implicant (Product term): product of literals — $ABC, AC, BC$
- Implicate (Sum terms): sum of literals — $(A+B+C), (A+C), (B+C)$
- Minterm: AND that includes all input variables — $ABC, A'BC, AB'C$
- Maxterm: OR that includes all input variables — $(A+B+C), (A'+B+C), (A'+B'+C)$
Recap: Boolean operators their circuit “gate” symbols

<table>
<thead>
<tr>
<th>Operator</th>
<th>Circuit Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td><img src="image" alt="AND gate" /></td>
</tr>
<tr>
<td>OR</td>
<td><img src="image" alt="OR gate" /></td>
</tr>
<tr>
<td>NOT</td>
<td><img src="image" alt="NOT gate" /></td>
</tr>
<tr>
<td>NAND</td>
<td><img src="image" alt="NAND gate" /></td>
</tr>
<tr>
<td>NOR</td>
<td><img src="image" alt="NOR gate" /></td>
</tr>
<tr>
<td>XOR</td>
<td><img src="image" alt="XOR gate" /></td>
</tr>
<tr>
<td>NXOR</td>
<td><img src="image" alt="NXOR gate" /></td>
</tr>
</tbody>
</table>

- Represents where we take a compliment value on an input.
- Represents where we take a compliment value on an output.
How to express \( y = e(ab+cd) \)

- **# inputs:** 5
- **# outputs:** 1
- **# gates:** 4
- **# signal nets:** 9
- **# pins:** 12

Diagram showing the logical expression and the count of gates, signal nets, and pins.
Recap: You can also use only NANDs

Now, only 5 gates and 4 transistors each — 20 transistors!
Recap: Canonical form — Sum of “Minterms”

### Truth Table 1: XNOR

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **A minterm**
- **Sum (OR) of “minterms”**

- **f(A,B) = A’B’ + AB**

### Truth Table 2: f(X,Y)

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **A minterm**
- **Sum (OR) of “minterms”**

- **f(X,Y) = XY’ + XY**

---

**Input**

**Output**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**A minterm**

**Sum (OR) of “minterms”**

**f(A,B) = A’B’ + AB**

---

**Input**

**Output**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Outline

• Let’s start designing the first circuit
• Designing circuit with HDL
• Let’s optimize the circuit!
Canonical form — Product of “Maxterms”

A “maxterm”

Input | Output
--- | ---
X | Y |
0 | 0 | 0
0 | 1 | 0
1 | 0 | 1
1 | 1 | 0

f(X, Y) = (X + Y) (X + Y')

Product of maxterms

XNOR

Input | Output
--- | ---
A | B |
0 | 0 | 1
0 | 1 | 0
1 | 0 | 0
1 | 1 | 1

f(A, B) = (A + B') (A' + B)
Sum-of-minterms/product-of-maxterms

- They can be used interchangeably
- Depends on if the truth table has more 0s or 1s in the result
- Neither forms give you the “optimized” equation. By optimized, we mean — minimize the number of operations
Let’s design a circuit!
Binary addition

3 + 2 = 5
\[
\begin{array}{c}
0 & 0 & 1 & 1
\end{array}
+ \begin{array}{c}
0 & 0 & 1 & 0
\end{array}
\hline
0 & 1 & 0 & 1
\]
carry

3 + 3 = 6
\[
\begin{array}{c}
0 & 0 & 1 & 1
\end{array}
+ \begin{array}{c}
0 & 0 & 1 & 1
\end{array}
\hline
0 & 1 & 1 & 0
\]

half adder — adder without a carry as an input
full adder — adder with a carry as an input

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Binary addition

Inputs — two 4-bit binary numbers:
\[ A_3A_2A_1A_0 \]
\[ B_3B_2B_1B_0 \]

Output — one 4-bit binary number
\[ O_3O_2O_1O_0 \]
Half adder

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Out = A'B + AB'
Cout = AB
How many of the following minterms are part of the sum-of-product form of the full adder in generating the output bit?

① A'B'Cin'
② A'BCin'
③ AB'Cin'
④ ABCin'
⑤ A'B'C
⑥ A'BCin
⑦ AB'Cin
⑧ ABCin
A. 0
B. 1
C. 2
D. 3
E. 4
The sum-of-product form of the full adder

- How many of the following minterms are part of the sum-of-product form of the full adder in generating the output bit?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>Out</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Out = A'B'Cin' + AB'Cin' + A'B'Cin + ABCin
Cout = ABCin' + A'BCin + AB'Cin + ABCin

- 0
- 1
- 2
- 3
- 4

E. 4
The full adder

Out = A'BCin' + AB'Cin' + A'B'Cin + ABCin
Cout = ABCin' + A'BCin + AB'Cin + ABCin

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Do we need to perform hardware design in gate-level?

— Not when you can use an HDL!
Turn a design into Verilog
Verilog

• Verilog is a Hardware Description Language (HDL)
  • Used to describe & model the operation of digital circuits.
  • Specify simulation procedure for the circuit and check its response — simulation requires a logic simulator.
  • Synthesis: transformation of the HDL description into a physical implementation (transistors, gates)
    • When a human does this, it is called logic design.
    • When a machine does this, it is called synthesis.

• In this class, we use Verilog to implement and verify your processor.
• C/Java like syntax
Data types in Verilog

• Bit vector is the only data type in Verilog
• A bit can be one of the following
  • 0: logic zero
  • 1: logic one
  • X: unknown logic value, don’t care
  • Z: high impedance, floating
• Bit vectors expressed in multiple ways
  • 4-bit binary: 4'b11_10 ( _ is just for readability)
  • 16-bit hex: 16'h034f
  • 32-bit decimal: 32'd270
# Operators

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Logical</th>
<th>Bitwise</th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>!</td>
<td>~</td>
<td>&gt;</td>
</tr>
<tr>
<td>-</td>
<td>&amp;&amp;</td>
<td>&amp;</td>
<td>&lt;</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>^</td>
<td>^=</td>
<td>&lt;=</td>
</tr>
<tr>
<td>%</td>
<td>~^</td>
<td>==</td>
<td></td>
</tr>
<tr>
<td>**</td>
<td>&lt;&lt;</td>
<td>!=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Don't use:

- Power

## Concatenation

\( \{ \text{e.g., } \{1b'1,1b'0\} \text{ is } 2b'10\} \)

## Replication

\( \{\} \text{ (e.g., } \{4\{1b'0\}\} \text{ is } 4b'0\) \)

## Conditional

\( \text{condition } ? \text{ value if true } : \text{ value if false} \)
Wire and Reg

- wire is used to denote a hardware net — “continuously assigned” values and do not store
  - single wire
    ```
    wire my_wire;
    ```
  - array of wires
    ```
    wire[7:0] my_wire;
    ```
- reg is used for procedural assignments — values that store information until the next value assignment is made.
  - again, can either have a single reg or an array
    ```
    reg[7:0] result; // 8-bit reg
    ```
  - reg is not necessarily a hardware register
  - you may consider it as a variable in C
Revisit the 4-bit adder
Half adder

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Out = A’B + AB’
Cout = AB

```verilog
module HA(input a, input b, output cout, output out);
assign out = (~a & b) | (a & ~b);
assign cout = a & b;
endmodule
```
Full adder

\[
\text{Out} = A'B'Cin' + AB'Cin' + A'B'Cin + ABCin \\
\text{Cout} = ABCin' + A'BCin + AB'Cin + ABCin
\]

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

module FA( input a, 
           input b, 
           input cin, 
           output cout, 
           output out );
assign out = (~a&b&~cin)|(a&~b&~cin)|(~a&~b&cin)|(a&b&cin);
assign cout = (a&b&~cin)|(~a&b&cin)|(a&~b&cin)|(a&b&cin);;
endmodule
The Adder

Connecting ports by name yields clearer and less buggy code.

module adder(
    input[3:0] A,
    input[3:0] B,
    output[3:0] O,
    output cout);
wire [2:0] carries;
HA ha0(.a(A[0]), .b(B[0]), .out(O[0]), .cout(carry[0]));
FA fa1(.a(A[1]), .b(B[1]), .cin(carry[0]), .out(O[1]), .cout(carry[1]));
FA fa2(.a(A[2]), .b(B[2]), .cin(carry[1]), .out(O[2]), .cout(carry[2]));
FA fa3(.a(A[3]), .b(B[3]), .cin(carry[2]), .out(O[2]), .cout(cout));
endmodule
Always block — combinational logic

- Executes when the condition in the sensitivity list occurs

```verilog
module FA( input a, 
    input b, 
    input cin, 
    output cout, 
    output out );

assign out = (~a&b&~cin)|(a&~b&~cin)|(~a&~b&cin)|(a&b&cin);
assign cout = (a&b&~cin)|(~a&b&cin)|(a&~b&cin)|(a&b&cin);;

always@(a or b or cin)
begin  // the following block changes outputs when a, b or cin changes
assign out = (~a&b&~cin)|(a&~b&~cin)|(~a&~b&cin)|(a&b&cin);
assign cout = (a&b&~cin)|(~a&b&cin)|(a&~b&cin)|(a&b&cin);;
end
endmodule
```
Always block — sequential logic

- Executes when the condition in the sensitivity list occurs

```vhdl
always@(posedge clk)// the following block only triggered by a positive clock
begin
  ...
  ...
end
```
Blocking and non-blocking

- Inside an always block, = is a blocking assignment
  - assignment happens immediately and affect the subsequent statements in the always block
- <= is a non-blocking assignment
  - All the assignments happens at the end of the block
- Assignment rules:
  - The left hand side, LHS, must be a reg.
  - The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants.

Initially, a = 2, b = 3

```verilog
reg a[3:0];
reg b[3:0];
reg c[3:0];
always @(posedge clock)
begin
a <= b;
c <= a;
end
Afterwards: a = 3 and c = 2
```

Afterwards: a = 3 and c = 3
"Always blocks" permit more advanced sequential idioms

module mux4( input a,b,c,d, 
       input [1:0] sel, 
       output out );

reg out;
always @( * )
begin
  if ( sel == 2’d0 )
    out = a;
  else if ( sel == 2’d1 )
    out = b
  else if ( sel == 2’d2 )
    out = c
  else if ( sel == 2’d3 )
    out = d
  else
    out = 1’bx;
end
endmodule

module mux4( input a,b,c,d, 
       input [1:0] sel, 
       output out );

reg out;
always @( * )
begin
  case ( sel )
    2’d0 : out = a;
    2’d1 : out = b;
    2’d2 : out = c;
    2’d3 : out = d;
    default : out = 1’bx;
  endcase
end
endmodule

Courtesy of Arvind http://csg.csail.mit.edu/6.375/
Initial block

- Executes only once in beginning of the code

```plaintext
initial
begin
  ...  
  ...
  end
```
`timescale 1ns/1ns // Add this to the top of your file to set time scale
module testbench();
reg [3:0] A, B;
reg C0;
wire [3:0] S;
wire C4;
adder uut (.B(B), .A(A), .sum(S), .cout(C4)); // instantiate adder

initial
begin
A = 4'd0; B = 4'd0; C0 = 1'b0;
#50 A = 4'd3; B = 4'd4; // wait 50 ns before next assignment
#50 A = 4'b0001; B = 4'b0010; // don't use #n outside of testbenches
end
endmodule
How many will get “1”s

- For the following Verilog code snippet, how many of their “output” values will be 1 after the “always” block finishes execution?

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
How many will get “1”s

• For the following Verilog code snippet, how many of their “output” values will be 1 after the “always” block finishes execution?

A. 0
B. 1
C. 2
D. 3
E. 4
Parameterize your module

module adder #(parameter WIDTH=32)(
    input[WIDTH-1:0] A,
    input[WIDTH-1:0] B,
    output[WIDTH-1:0] O,
    output cout);
endmodule
Coding guides

- When modeling sequential logic, use nonblocking assignments.
- When modeling latches, use nonblocking assignments.
- When modeling combinational logic with an always block, use blocking assignments.
- When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.
- Do not mix blocking and nonblocking assignments in the same always block.
- Do not make assignments to the same variable from more than one always block.
- Use $strobe to display values that have been assigned using nonblocking assignments.
- Do not make assignments using #0 delays.

Electrical Computer Engineering
Science 120A