

Intro to Digital Design

L3: Karnaugh Maps

Instructor: Naomi Alterman

Teaching Assistants:

Derek de Leuw

Isabel Froelich

Kevin Hernandez

Aarjav Jain

Packard Stephenson

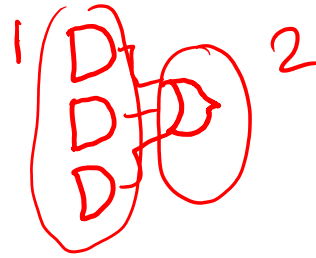
Administrivia

- ❖ Lab 3 out – Logic simplification with K-maps and Verilog
 - Due a week from tomorrow (4/22 @ 2:30p)
 - Full credit for minimal logic

Lecture Outline

- ❖ **Karnaugh Maps (K-maps)**
- ❖ Avoiding Hazards
- ❖ Design Examples
- ❖ More Verilog

On and Off Sets



❖ *On Set* is the set of input patterns where the function is TRUE

- Here on set = $\{\bar{A}\bar{B}C, \bar{A}BC, A\bar{B}\bar{C}, A\bar{B}C\}$

❖ *Off Set* is the set of input patterns where the function is FALSE

- Here off set = $\{\bar{A}\bar{B}\bar{C}, \bar{A}B\bar{C}, A\bar{B}\bar{C}, ABC\}$

❖ **Recall:** Use the On Set for Sum of Products (SoP) and the Off Set for Product of Sums (PoS)

- Considered two-level Boolean expressions

3 input fn.

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

Two-Level Simplification

- ❖ Using Sum of Products, “neighboring” input combinations simplify
 - “Neighboring”: inputs that differ by a single signal
 - *The Uniting Theorem*: $A(\bar{B} + B) = A$
 - e.g., $\cancel{A}B + \cancel{A}\bar{B} = B$, $\bar{A}\cancel{B}C + \bar{A}\cancel{B}\bar{C} = \bar{A}B$
- ❖ **Goal**: Find neighboring subsets of the On Set to eliminate variables and simplify the expression
- ❖ **Technique**: A **Karnaugh map** (“K-map”) is a method of rearranging a truth table to *visualize* the terms that can be simplified with the uniting theorem

Karnaugh Maps

- ❖ Rearrange truth table into a 2D grid with inputs on the edges and outputs in the grid cells
- ❖ If more than 2 inputs, group them along the edges and write out all combinations neighboring cells change only by 1 input ("Gray code")

Binary integer order

A	B	F
0	0	😊
0	1	😞
1	0	😏
1	1	😡

Gray code order

A	B	F
0	0	😊
0	1	😞
1	0	😏
1	1	😡

AB

C	00	01	11	10
0				✓
1				

CD

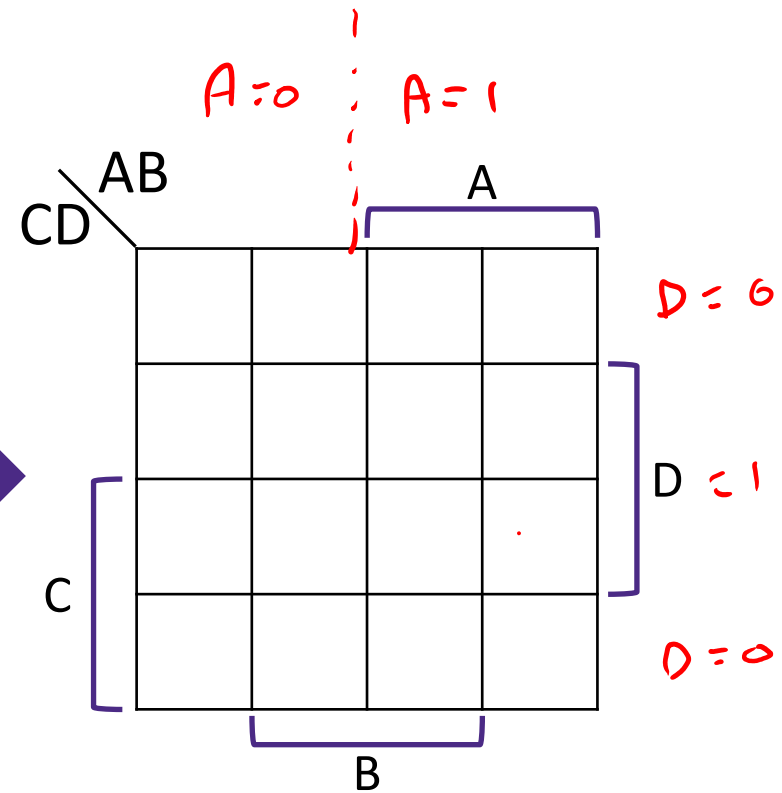
CD	00	01	11	10
00				
01				
11				
10				

Karnaugh Maps

- ❖ Can also draw it using brackets instead of bits. A bracket means the bit is “asserted”/1 in those columns or rows

4 Inputs:

CD \ AB	00	01	11	10
00				
01				
11				
10				



K-map Example: Majority Circuit

❖ Filling in a Karnaugh map:

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

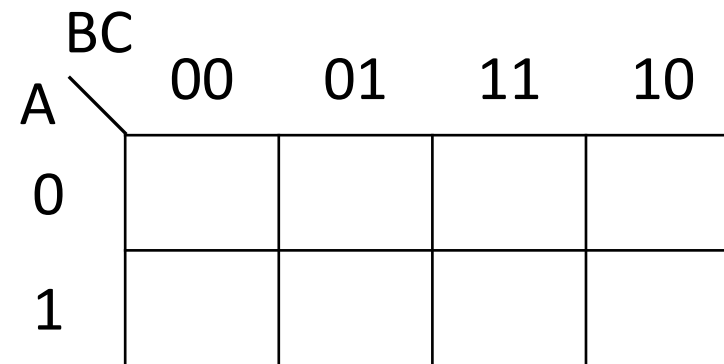
		AB			
		00	01	11	10
C	0	0	0	1	0
	1	0	1	1	1

- ❖ Each row of truth table corresponds to ONE cell of Karnaugh map
- ❖ Note the jump when you go from input 011 to 100
(most mistakes made here)

K-map Example: Majority Circuit

- ❖ Filling in alternate Karnaugh map:

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

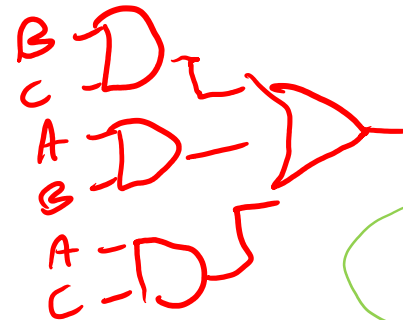


- ❖ Each row of truth table corresponds to ONE cell of Karnaugh map
- ❖ Note the jump when you go from input 001 to 010 and 101 to 110 (*most mistakes made here*)

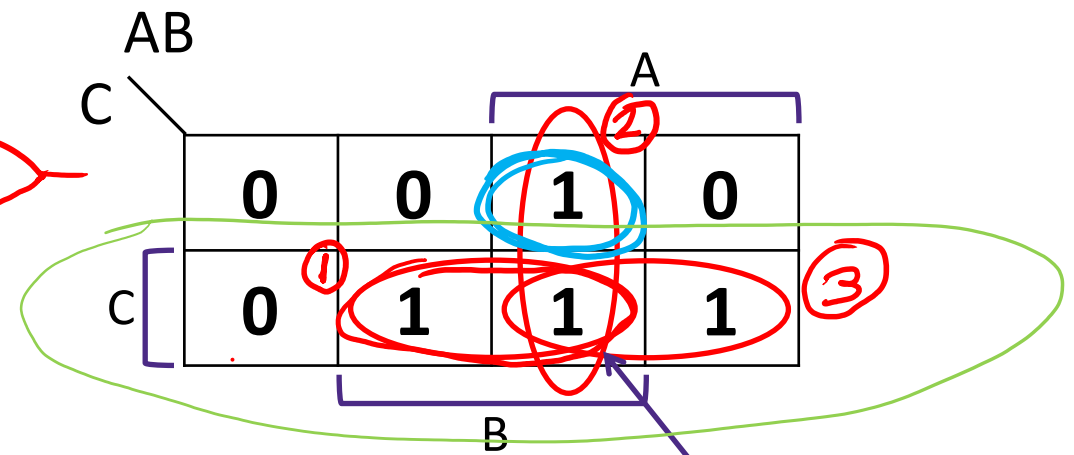
K-map Simplification

- ❖ Group neighboring 1's with circles
 - Each circle becomes an equation term
 - Make sure **all** 1's are circled

❖ $F = \overset{\textcircled{1}}{BC} + \overset{\textcircled{2}}{AB} + \overset{\textcircled{3}}{AC}$



- ❖ Larger groups become smaller terms
 - The single 1 in top row → $ABC\bar{C}$
 - Vertical group of two 1's → AB
 - If entire lower row was 1's → C



Single cell can be part of many groups

General K-map Rules

- ❖ Only group in powers of 2
 - Grouping should be of size $2^i \times 2^j$
 - Applies for both directions
- ❖ Wraps around in all directions
 - “Corners” case is extreme example
- ❖ Always choose largest groupings possible
 - Avoid single cells whenever possible
- ❖ $F = BD + \overline{B}\overline{D} + ACD$

	AB			
	00	01	11	10
CD	00	01	11	10
00	1	0	0	1
01	0	1	1	0
11	0	1	1	1
10	1	0	0	1

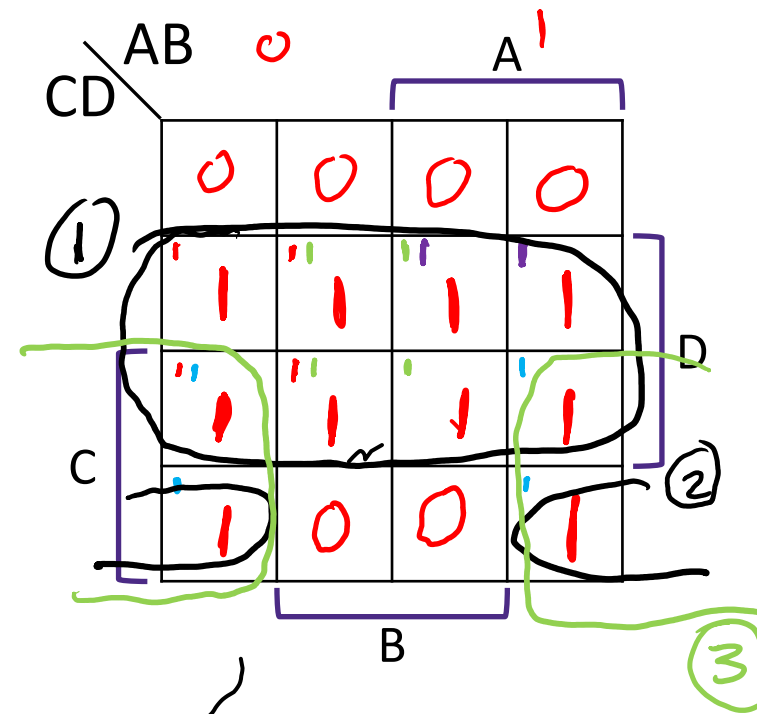
- 1) NOT a valid group
- 2) IS a valid group
- 3) IS a valid group
- 4) “Corners” case
- 5) 1 of 2 good choices here

K-Map Example

$$\diamond F = \underline{\bar{A}D} + \underline{BD} + \underline{\bar{B}C} + A\bar{B}D$$

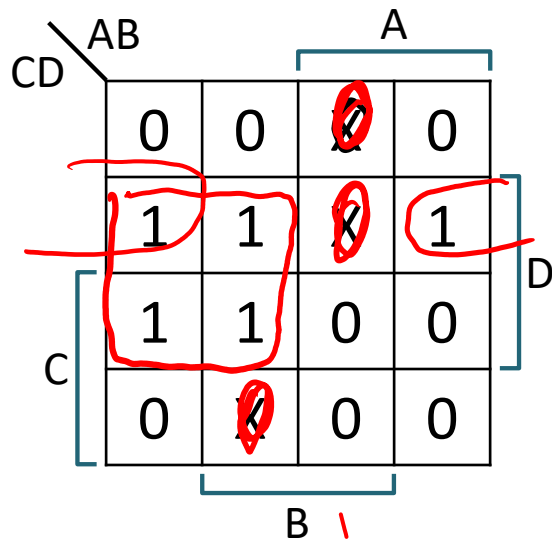
$$= D^{(1)} + \bar{B}C\bar{D}^{(2)}$$

$$= D^{(1)} + \bar{B}C^{(3)}$$



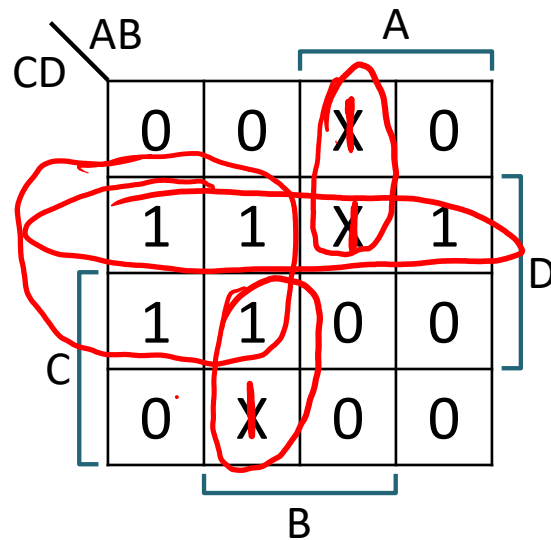
Don't Cares

- ❖ Use symbol 'X' to mean it can be either a 0 or 1
 - Make choice to simplify final expression



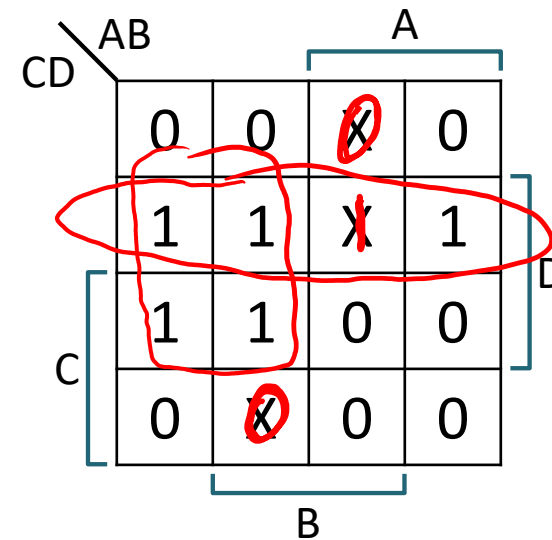
Let all X = 0:

F =



Let all X = 1:

F =



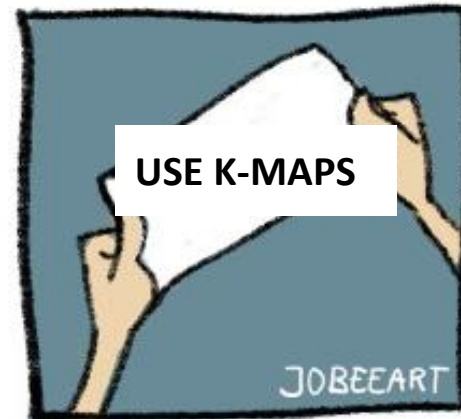
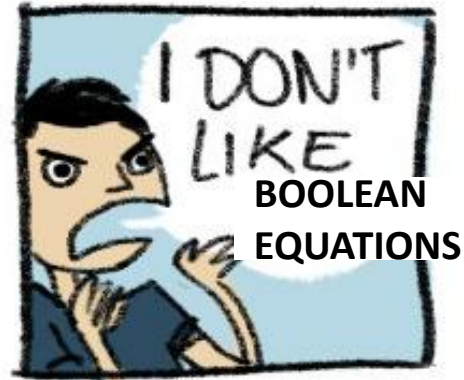
Choose wisely:

F =

Two-Level Simplification (at SCALE!)

- ❖ As an aside: K-Maps are for ≤ 4 dimensions
- ❖ For more, we use a computer implementation of the same technique called the “Quine-McCluskey algorithm”
 - This is part of what Quartus is doing when you hit the “synthesize” button





Lecture Outline

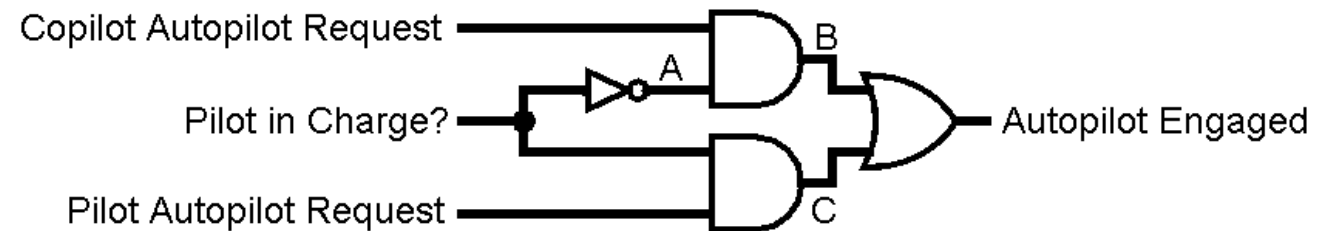
- ❖ Karnaugh Maps (K-maps)
- ❖ **Avoiding Hazards**
- ❖ Design Examples
- ❖ More Verilog

Timing Diagrams Matter



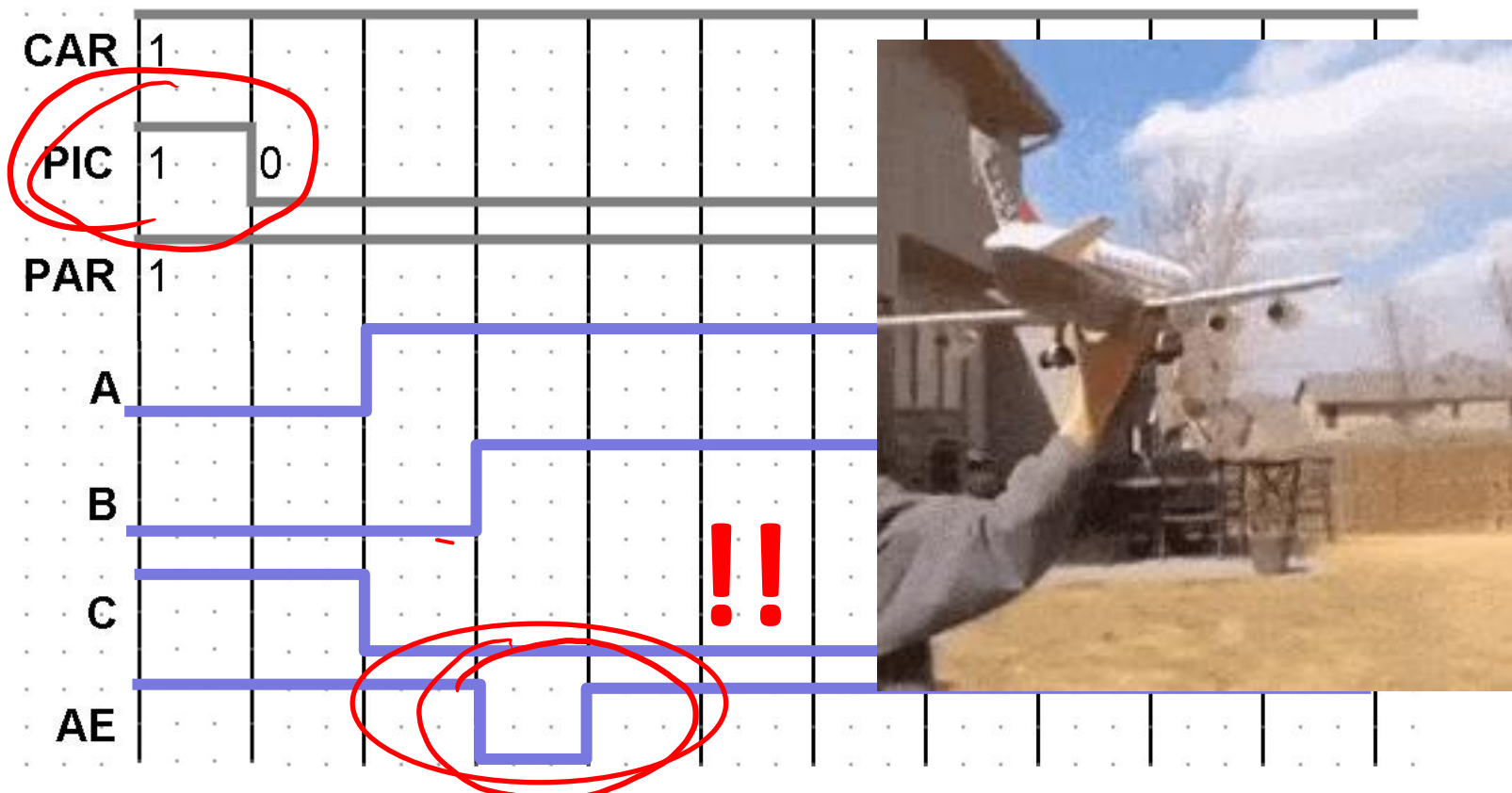
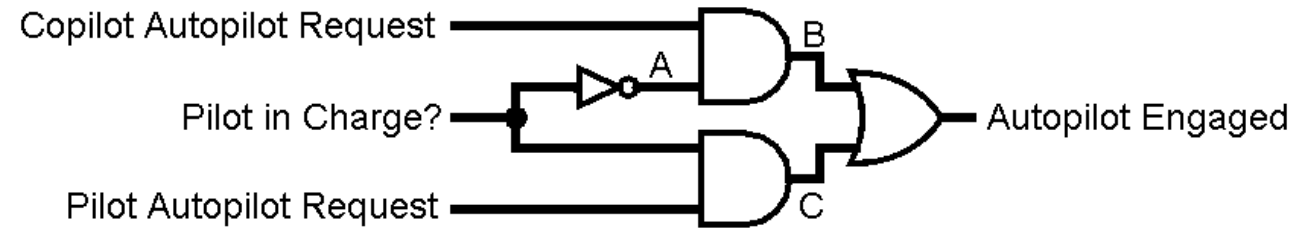
Who is in charge?

- Pilot?
- Copilot?
- Autopilot?



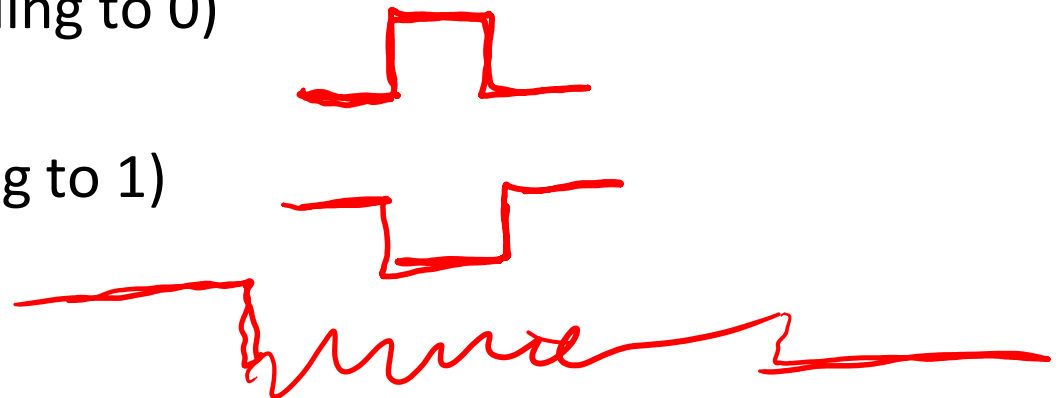
Circuit Timing: Hazards/Glitches

- ❖ Circuits can temporarily go to incorrect states!
- Assuming gate delay of 1 ns / 3 ticks

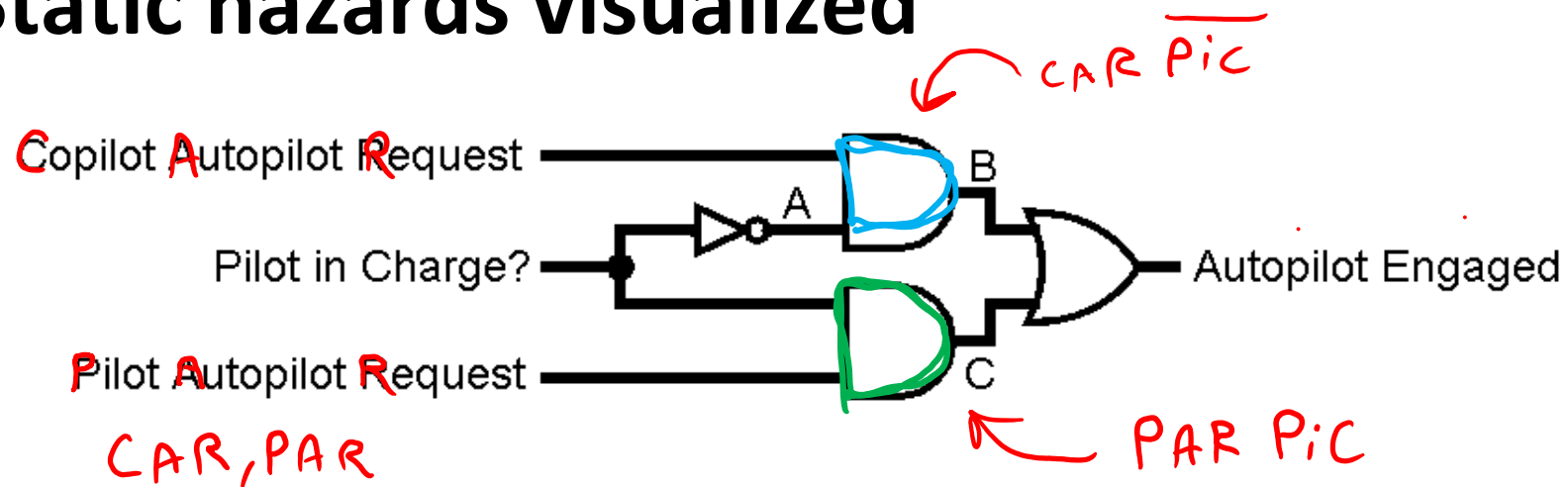


Circuit Hazards

- ❖ Momentary wiggles in output of combinational logic before it “settles” on the right answer
 - Mostly doesn't matter (we'll see why next week)
 - But when it does matter, it can make the plane crash 🐱
 - Can be avoided with redundant gates (eg, not fully simplified equations)
- ❖ Classification:
 - “Static-0” (output spikes to 1 before settling to 0)
 - Only a problem for PoS equations
 - “Static-1” (output dips to 0 before settling to 1)
 - Only a problem for SoP equations
 - “Dynamic” (lots of wiggles)



Static hazards visualized



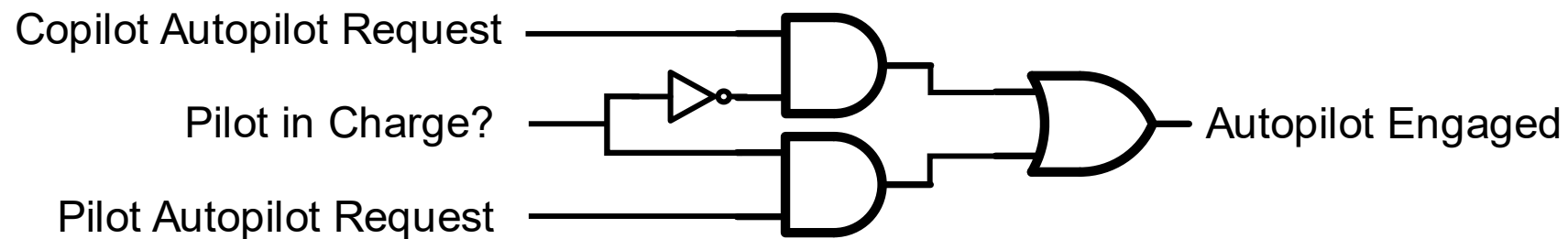
Handwritten annotations: CAR, PAR (pointing to the top row of the K-map); Pic (pointing to the left column of the K-map).

	00	01	11	10
0	0	0	1	1
1	0	1	1	0

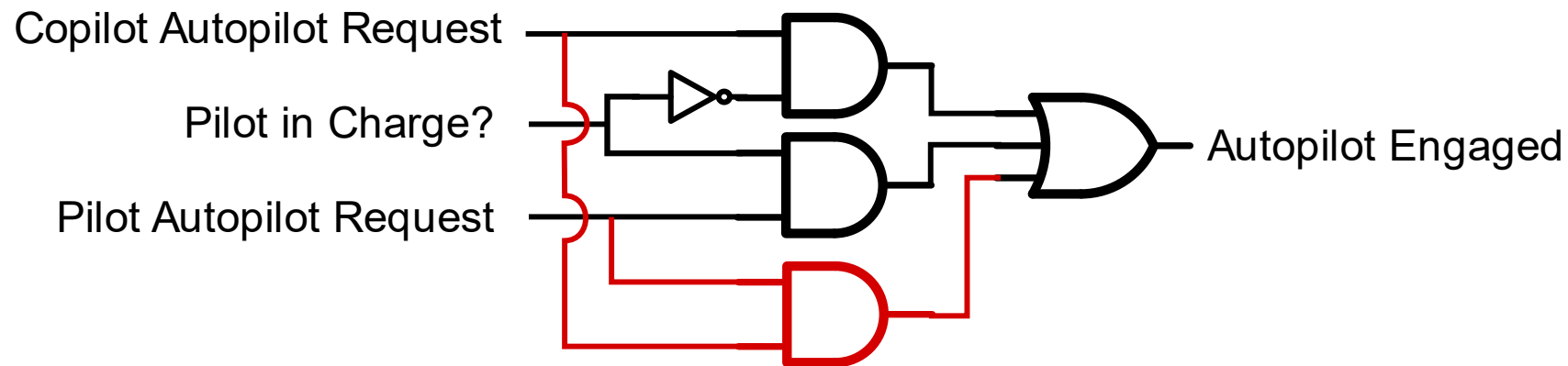
The K-map shows a static hazard at the transition between (0,1) and (1,1) for the input combination (1,1). The hazard is highlighted with a purple circle around the '1' in the (1,1) cell. A green circle highlights the '1' in the (1,1) cell and the '1' in the (1,0) cell. A blue circle highlights the '1' in the (0,1) cell and the '1' in the (0,0) cell.

Design Tradeoffs

Minimal logic



No static hazards



Cool band names

- ❖ From the “See Also” section of the [Wikipedia article](#) for logic hazards:

See also [\[edit\]](#)

- [Don't care](#)
- [Floating body effect](#), a probably cause for
- [Glitch](#)
- [Hazard \(computer architecture\)](#)
- [Logic redundancy](#)
- [Race condition](#)

Miso Moment

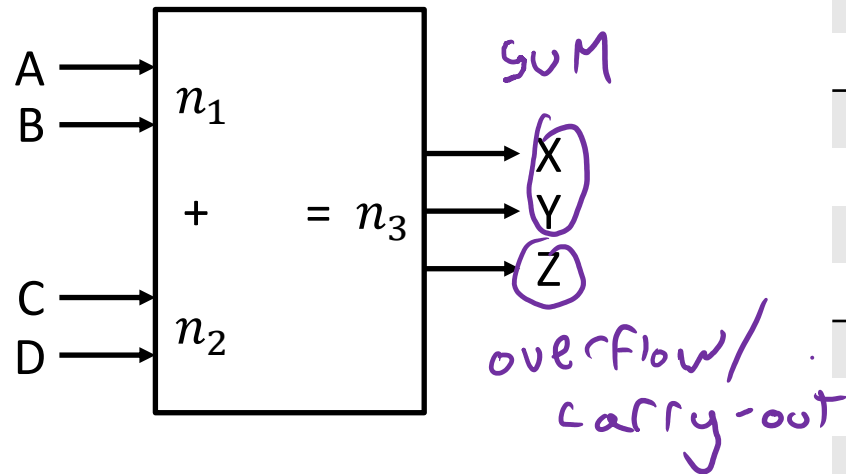


Lecture Outline

- ❖ Karnaugh Maps (K-maps)
- ❖ Avoiding Hazards
- ❖ **Design Examples**
- ❖ More Verilog

Design Example: 2-bit Adder

❖ Block Diagram and Truth Table:

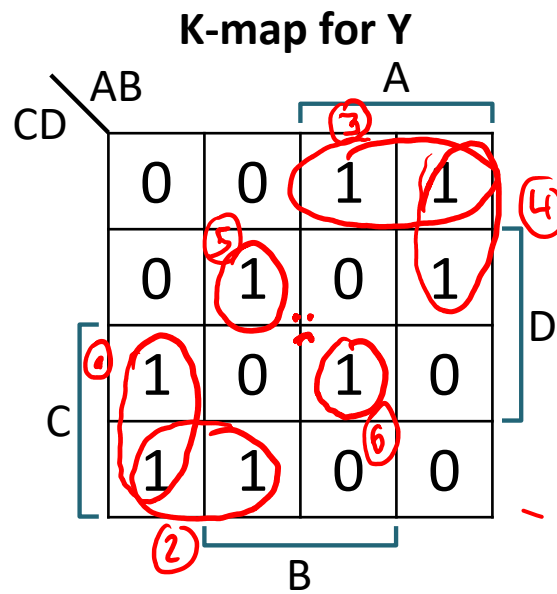
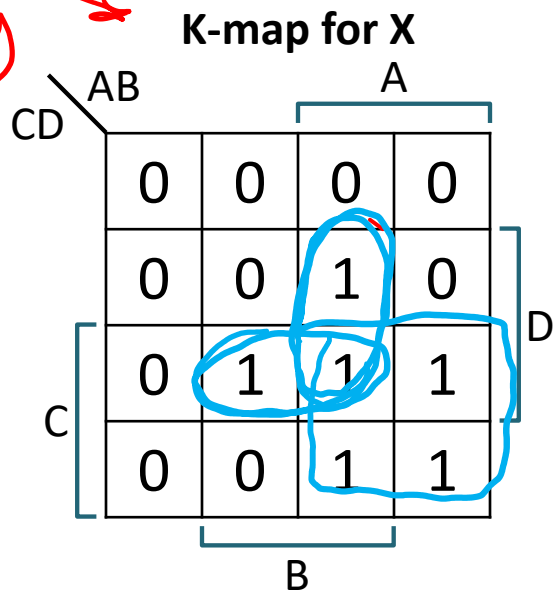


We'll need a new k-map for **each** output

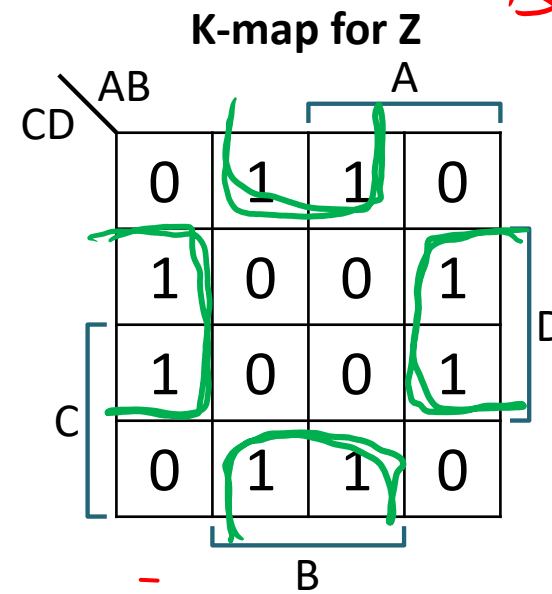
A	B	C	D	X	Y	Z
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Design Example: 2-bit Adder

EVEN B-DAY →



ODD BDAY ↙



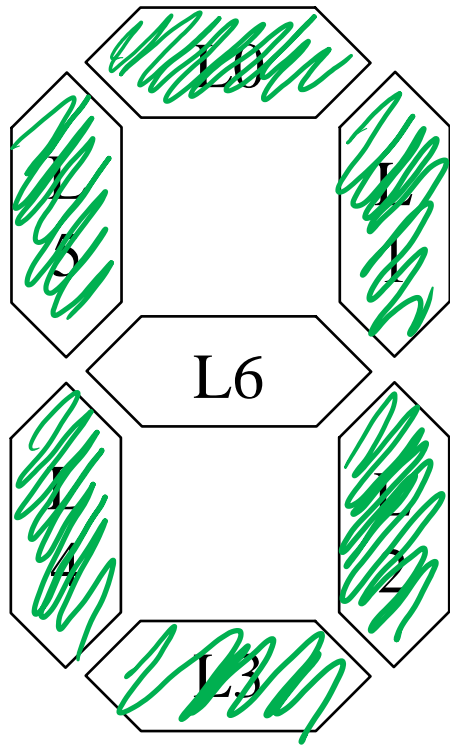
$$X = DAB + BCD + CA$$

$$Y = \bar{A}\bar{B}C + \bar{A}C\bar{D} + A\bar{C}\bar{D} + A\bar{B}\bar{C} + \bar{A}B\bar{C}D + ABCD$$

$$Z = \bar{B}D + B\bar{D} = B \text{ xor } D = B \oplus D$$

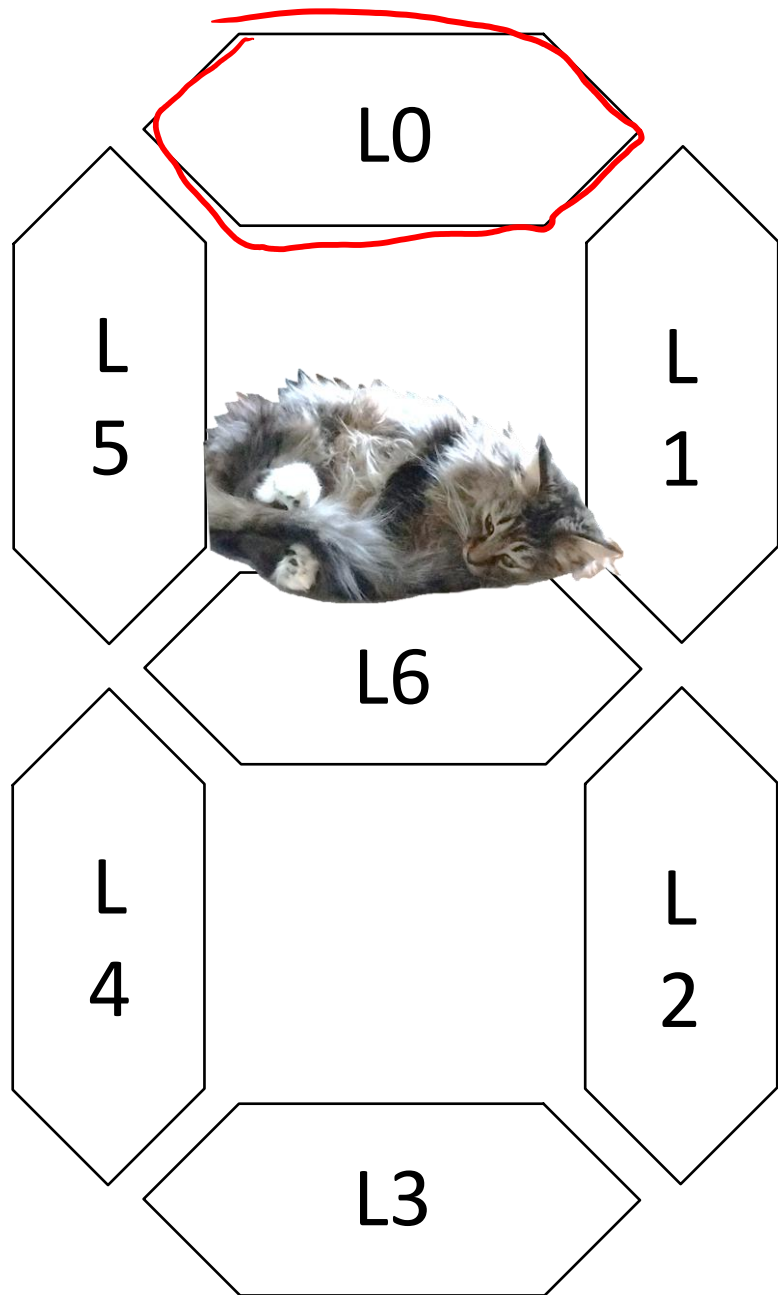
Case Study: Seven-Segment Display

- ❖ Chip to drive digital display

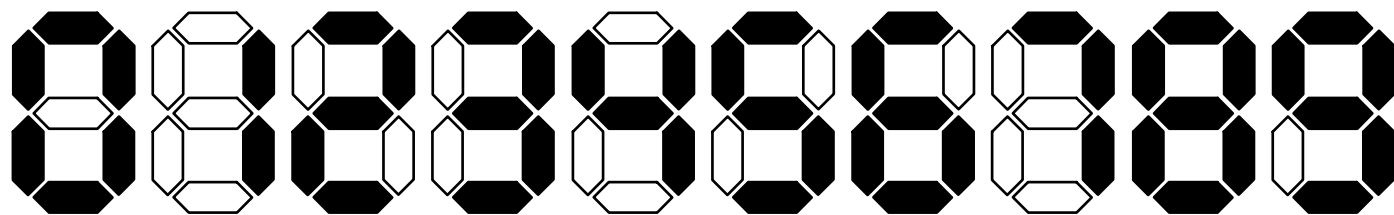


	B3	B2	B1	B0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0

...



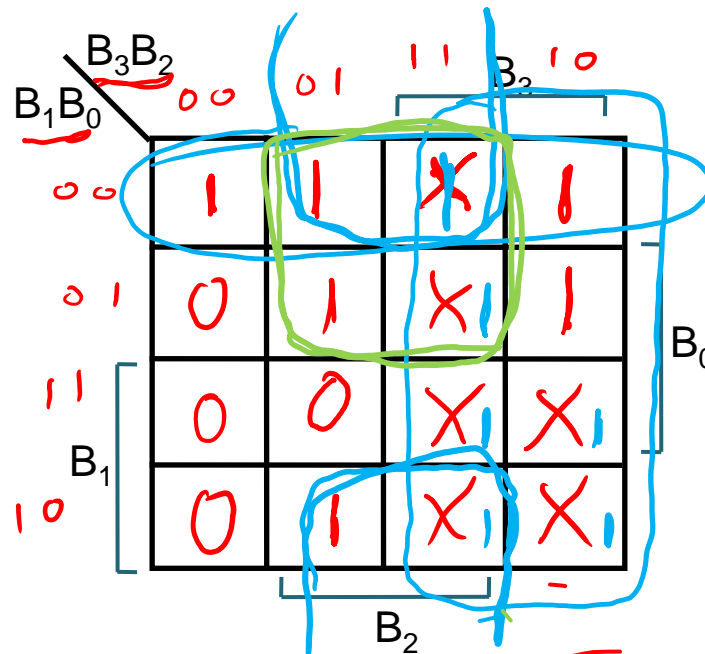
B3	B2	B1	B0	Val	L0	L1	L2	L3	L4	L5	L6
0	0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	1	0	1	1	0	0	0	0
0	0	1	0	2	1	1	0	1	1	0	1
0	0	1	1	3	1	1	1	1	0	0	1
0	1	0	0	4	0	1	1	0	0	1	1
0	1	0	1	5	1	0	1	1	0	1	1
0	1	1	0	6	1	0	1	1	1	1	1
0	1	1	1	7	1	1	1	0	0	0	0
1	0	0	0	8	1	1	1	1	1	1	1
1	0	0	1	9	1	1	1	1	0	1	1



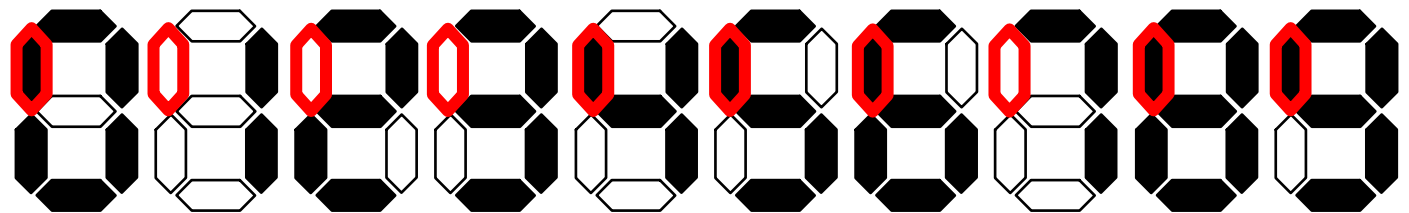
Case Study: Seven-Segment Display

❖ Implement **L5**:

B3	B2	B1	B0	L5
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1

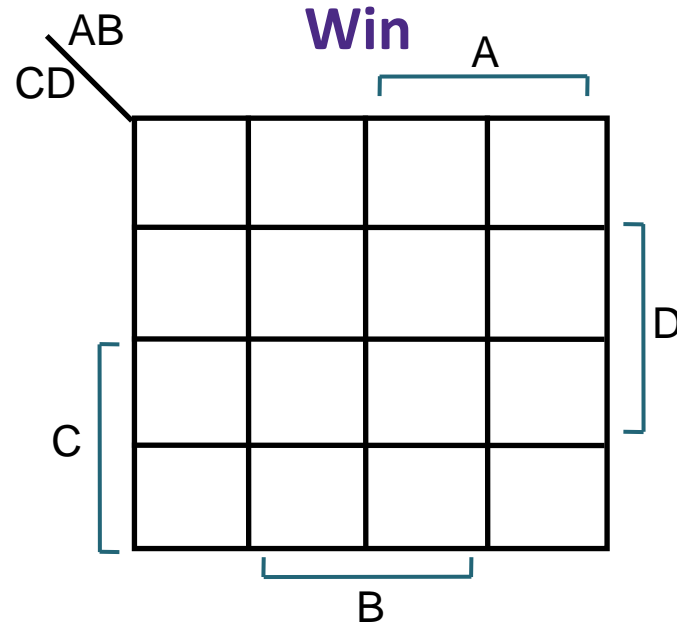


$$B_3 + \bar{B}_1 \bar{B}_0 + B_2 \bar{B}_0 + \bar{B}_1 B_2$$



Practice Problem: Rock-Paper-Scissors

- ❖ Rock (00), Paper (01), Scissors (10) for two players P0 and P1
- ❖ **Output:** Win = Winner's ID (0/1)
Tie = 1 if Tie, 0 else



P1		P0		Win	Tie
A	B	C	D		
0	0	0	0		
0	0	0	1		
0	0	1	0		
0	0	1	1		
0	1	0	0		
0	1	0	1		
0	1	1	0		
0	1	1	1		
1	0	0	0		
1	0	0	1		
1	0	1	0		
1	0	1	1		
1	1	0	0		
1	1	0	1		
1	1	1	0		
1	1	1	1		

Lecture Outline

- ❖ Karnaugh Maps (K-maps)
- ❖ Avoiding Hazards
- ❖ Design Examples
- ❖ **More Verilog**

Verilog “control flow”

- ❖ Verilog has synthesizable if and case statements
 - In hardware, each “clause” exists in parallel and is always computing its output whether the condition is true or not
 - A multiplexer (“mux”) is placed physically after these blocks to electrically connect one of them to the output signal
- ❖ Verilog requires us to wrap these statements in an always_comb block
 - Block defines the full set of circuits that may drive the value on a logic variable
 - Idea: the last assignment in an always block to a given variable is the result that gets used
- ❖ There are several other kinds of “always” block.
 - We’ll learn about them next week!

```
if ( . . . ) {  
    . . .  
} else {  
    . . .  
}
```

7-Seg Display in Verilog

```

module seg7 (bcd, leds);
  input logic [3:0] bcd;
  output logic [6:0] leds;

  always comb
  case (bcd)
    // 3210          6543210
    0 4'b0000: leds = 7'b0111111;
    1 4'b0001: leds = 7'b0000110;
    2 4'b0010: leds = 7'b1011011;
    3 4'b0011: leds = 7'b1001111;
    4 4'b0100: leds = 7'b1100110;
    5 4'b0101: leds = 7'b1101101;
    6 4'b0110: leds = 7'b1111101;
    7 4'b0111: leds = 7'b0000111;
    8 4'b1000: leds = 7'b1111111;
    9 4'b1001: leds = 7'b1101111;
    default: leds = 7'bX;
  endcase
endmodule

```

busses!
Multi-bit signals

no Fall-through!

4 bits
in binary

Verilog Signal Manipulation

- ❖ Bus definition: $[n-1:0]$ is an n-bit bus
 - Good practice to follow bit numbering notation
 - Access individual bit/wire using “array” syntax (e.g., `bus[1]`)
 - Can slice busses using similar notation (e.g., `bus[4:2]`)
- ❖ Multi-bit constants: $n'r###\dots$
 - n is bit width, b is radix/base, #s are the actual digits
 - e.g., `4'd12`, `4'b1100`, `4'hC`
MSB \longrightarrow LSB
- ❖ Concatenation: $\{sig, \dots, sig\}$
 - Ordering matters; result will have combined widths of all signals
- ❖ Replication operator: $\{n\{m\}\}$
 - repeats value m, n times

Let's check our understanding

```

logic [4:0] apple; 5 bit
logic [3:0] pear; 4 bit
logic [9:0] orange;
assign apple = 5'd20;
assign pear = {1'b0, apple[2:1], apple[4]};
    
```

4 3 2 1 0
 5'b 1 0 1 0 0
 1 bit + 2 bits + 1 bit = 4 bit

❖ What's the value of pear?

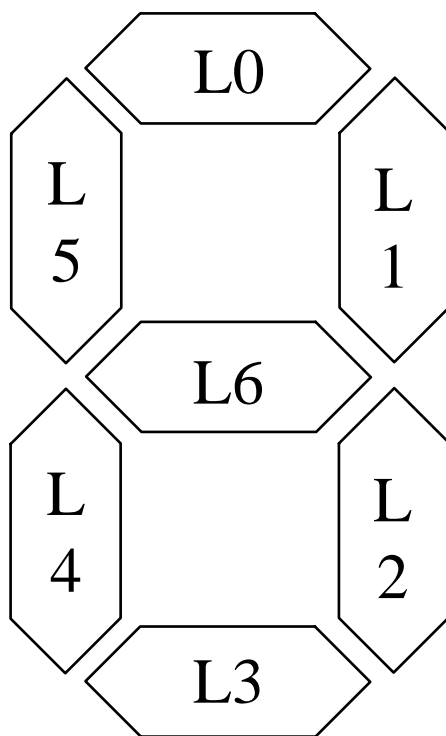
$$\begin{array}{r}
 4'b \ 0 \ 1 \ 0 \ 1 \\
 \hline
 \# \ 3 \ 2 \ 1 \ 0
 \end{array}
 = 5$$

❖ If we want orange to be the *sign-extended* version of apple, what is the appropriate Verilog statement?

```
assign orange =
```

Practice Problem: Extend 7-Seg to Hex

- ❖ Show “A” on 0b1010 (ten) to “F” on 0b1111 (fifteen)



```

module seg7_hex (bcd, leds);
  input logic [3:0] bcd;
  output logic [6:0] leds;

  always_comb
  case (bcd)
    // 3210                6543210
    4'b0000: leds = 7'b0111111;
    4'b0001: leds = 7'b0000110;
    4'b0010: leds = 7'b1011011;
    4'b0011: leds = 7'b1001111;
    4'b0100: leds = 7'b1100110;
    4'b0101: leds = 7'b1101101;
    4'b0110: leds = 7'b1111101;
    4'b0111: leds = 7'b0000111;
    4'b1000: leds = 7'b1111111;
    4'b1001: leds = 7'b1101111;
    // TODO: What else??
    default: leds = 7'bX;
  endcase
endmodule

```