Intro to Digital Design
L3: Karnaugh Maps

Instructor: Naomi Alterman

Teaching Assistants:

Derek de Leuw Isabel Froelich

Kevin Hernandez Sathvik Kanuri

Aadithya Manoj

Administrivia

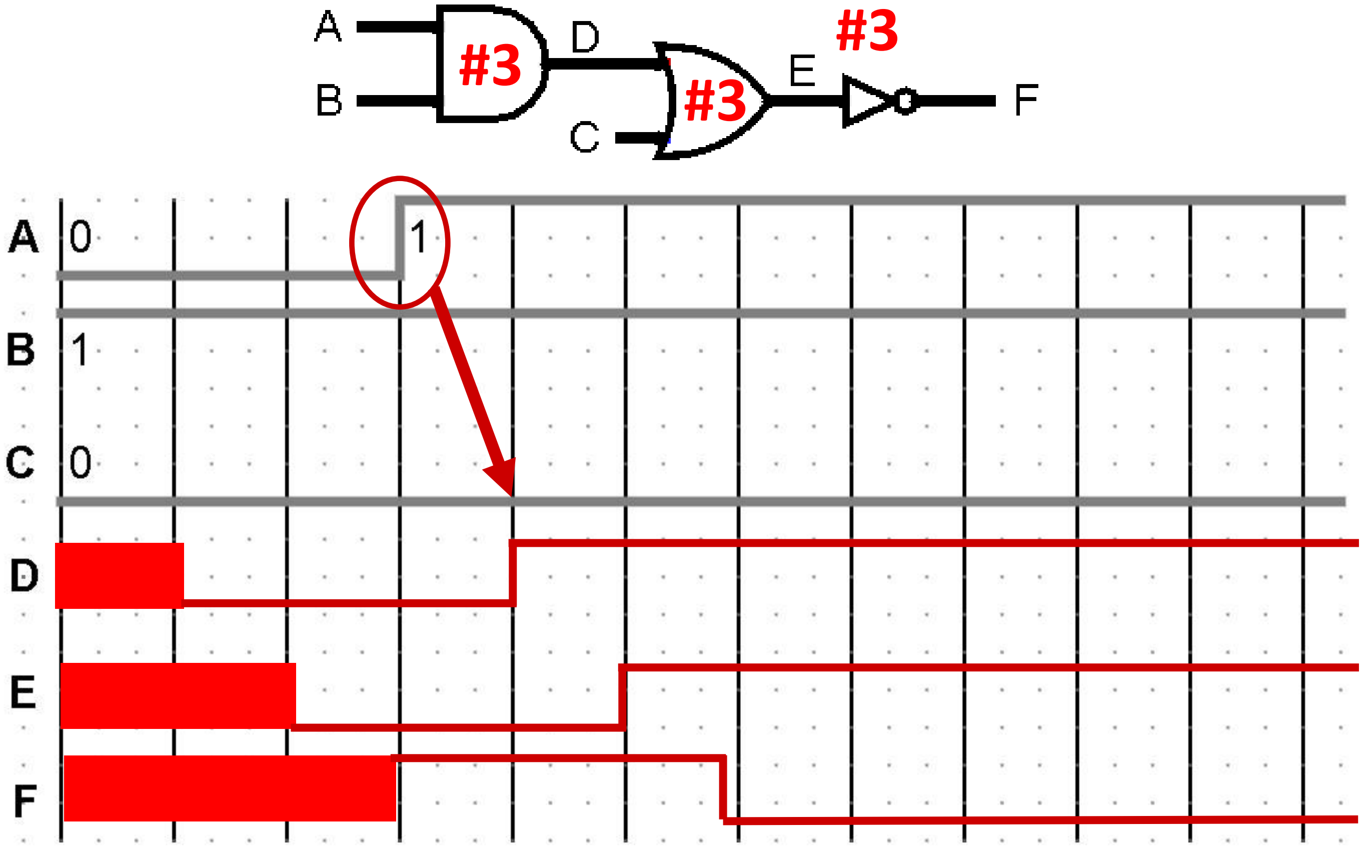
- Lab 1 & 2 approaches!
  - Reports + code due on Gradescope by Wednesday @ 2:30p, *regardless of your demo time!*
  - Demos due during your assigned demo slots
    - If you don’t have a lab kit yet, demo your design to TAs with LabsLand
- Lab 3 out – Logic simplification with K-maps and Verilog
  - Due a week from tomorrow (1/28 @ 2:30p)
  - Full credit for minimal logic
- More lab kits arriving soon 🤞
  - Hopefully in time to give them to you during your lab demo

When last we left off…

Circuit Hazards

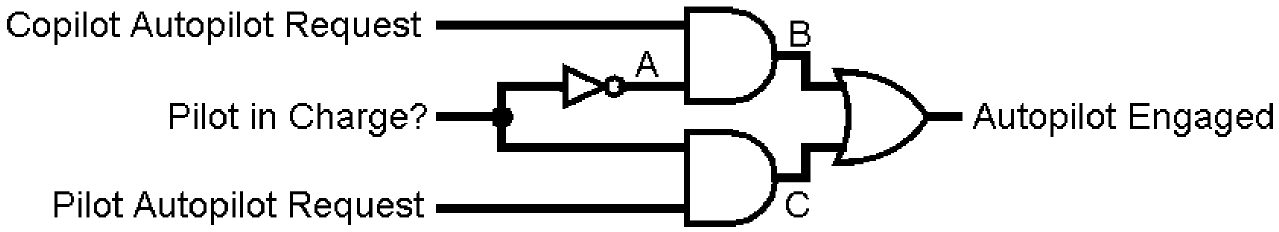
Correction from Lec 2

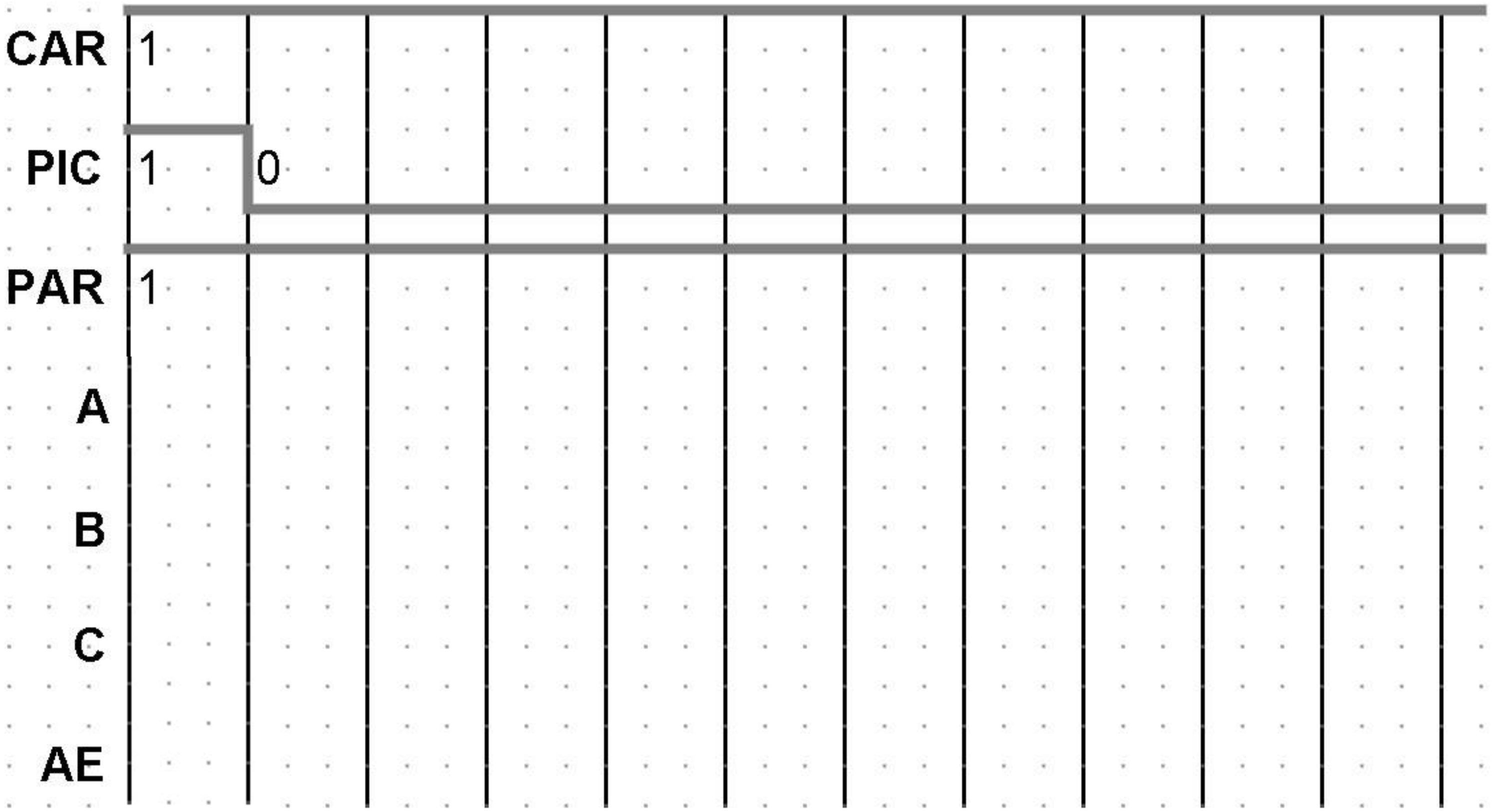
- Corrected diagram from slide 38 last week (sorry!)
  - Assuming gate delay of 1 ns (= 3 ticks)



Circuit Timing: Hazards/Glitches

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





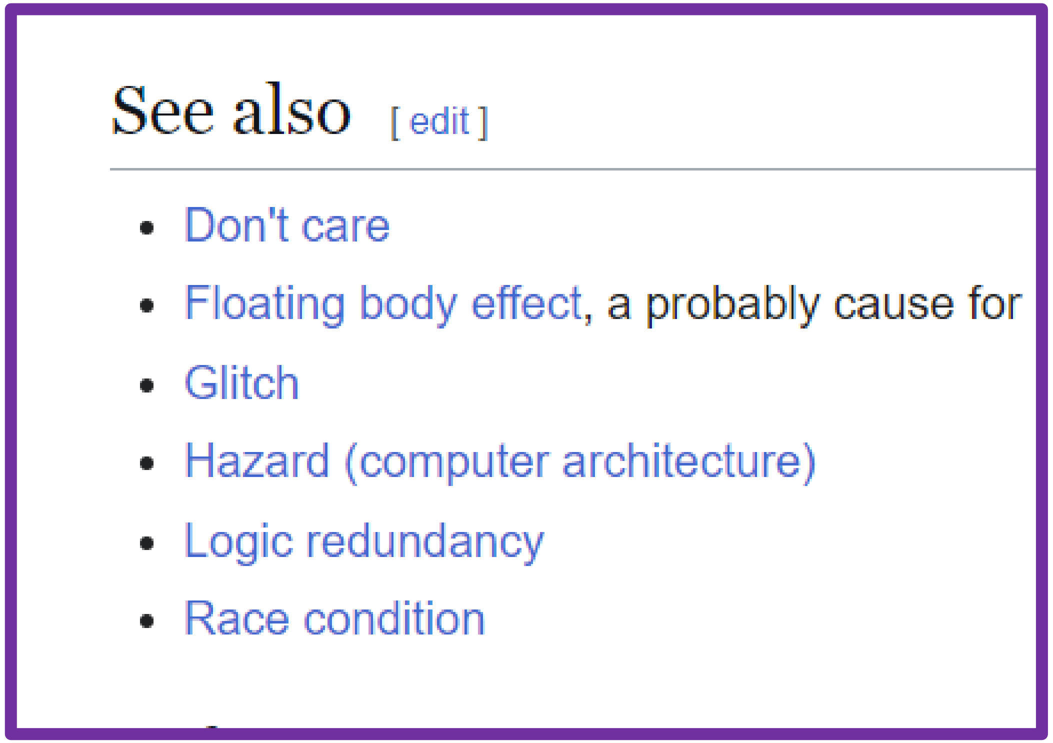
Circuit Hazards

- 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)

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

Lecture Outline

- Hazards
- **Karnaugh Maps (K-maps)**
- Design Examples

On and Off Sets

- *On Set* is the set of input patterns where the function is TRUE
  - Here on set = $\{\overline{A}\,\overline{B}C, \overline{A}BC, A\overline{B}\,\overline{C}, A\overline{B}C\}$

- *Off Set* is the set of input patterns where the function if FALSE
  - Here off set = $\{\overline{A}\,\overline{B}\,\overline{C}, \overline{A}B\overline{C}, AB\overline{C}, ABC\}$

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

- **Recall:** Use the On Set for *Sum of Products* (SoP) and the Off Set for *Product of Sums* (PoS)
  - Considered two-level Boolean expressions

Two-Level Simplification

- Using Sum of Products, “neighboring” input combinations simplify
  - “Neighboring”: inputs that differ by a single signal
  - *The Uniting Theorem:* $\mathrm{A}(\overline{\mathrm{B}}+\mathrm{B})=\mathrm{A}$
  - *e.g.*, $\mathrm{AB}+\overline{\mathrm{A}}\mathrm{B}=\mathrm{B}$, $\overline{\mathrm{A}}\mathrm{BC}+\overline{\mathrm{A}}\mathrm{B}\overline{\mathrm{C}}=\overline{\mathrm{A}}\mathrm{B}$

- **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
  - For ≤ 4 dimensions. For more, we use a computer implementation of the same technique called the “Quine-McCluskey algorithm”

Karnaugh Maps

- Rearrange 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 of bits arranged so that neighboring combinations change only by 1 input ("**Gray code**")

2 Inputs:

A \ B	0	1
0		
1		

3 Inputs:

C \ AB	00	01	11	10
0				
1				

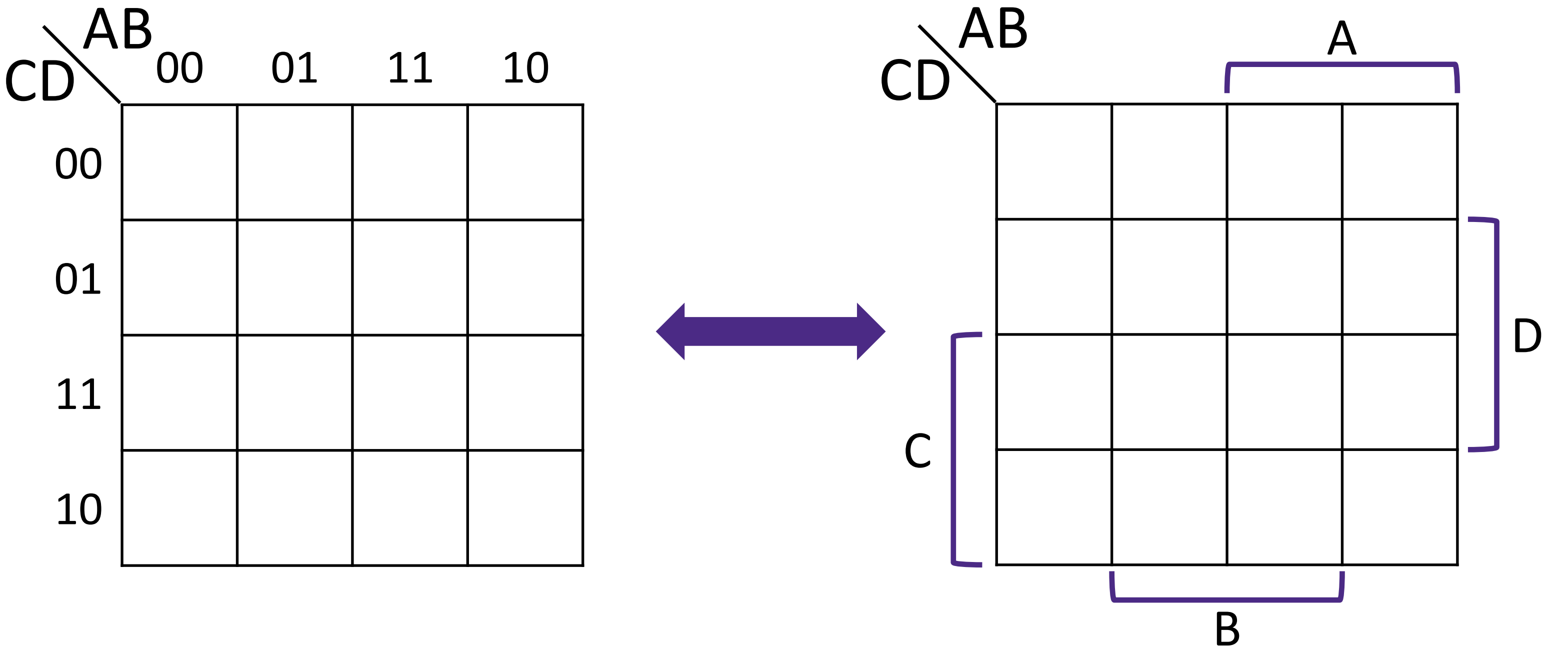
4 Inputs:

CD \ AB	00	01	11	10
00				
01				
11				
10				

Karnaugh Maps

- Also see visualization with brackets for “asserted” simplifications:

4 Inputs:



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

C \ AB	00	01	11	10
0				
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:

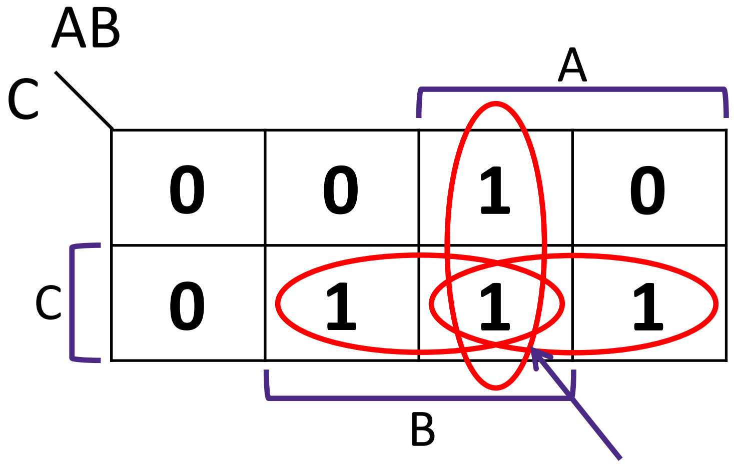
A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

A \ BC	00	01	11	10
0				
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
- $\mathrm{F} = \mathrm{BC} + \mathrm{AB} + \mathrm{AC}$
- Larger groups become smaller terms
  - The single 1 in top row → $\mathrm{AB\overline{C}}$
  - Vertical group of two 1's → AB
  - If entire lower row was 1's → C



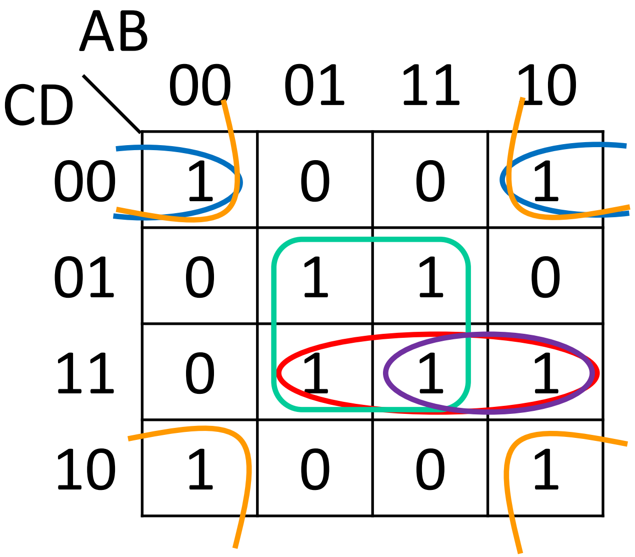
0	0	1	0
0	1	1	1

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

- $F = BD + \overline{B}\overline{D} + ACD$

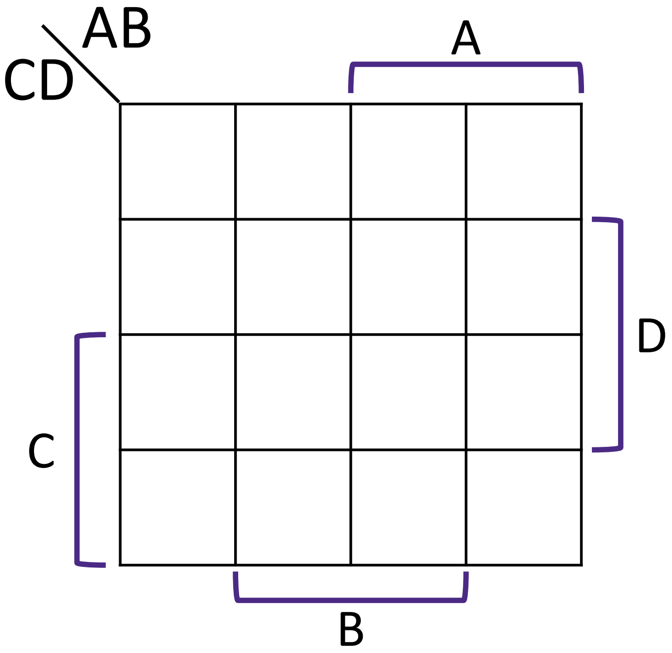


CD \ AB	00	01	11	10
00	1	0	0	1
01	0	1	1	0
11	0	1	1	1
10	1	0	0	1

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

- $F = \overline{A}D + BD + \overline{B}C + A\overline{B}D$



I DON'T LIKE
BOOLEAN EQUATIONS
USE K-MAPS
JOBEEART

Miso Moment

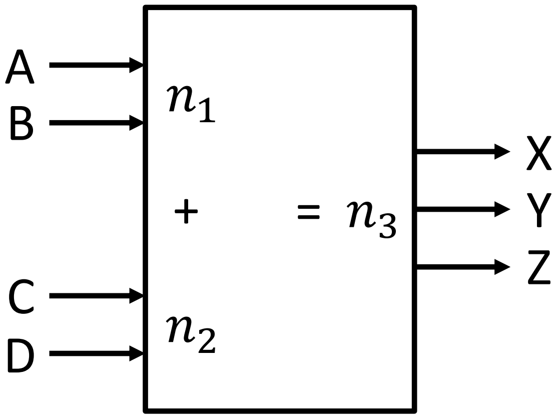
Lecture Outline

- Hazards
- Karnaugh Maps (K-maps)
- **Design Examples**

Design Example: 2-bit Adder

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

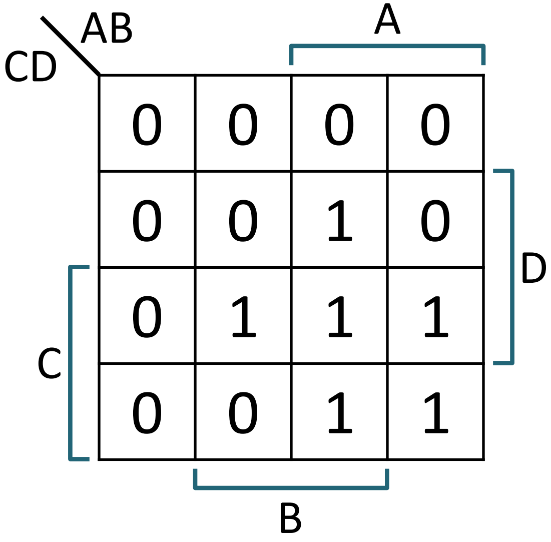
- Block Diagram and Truth Table:



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

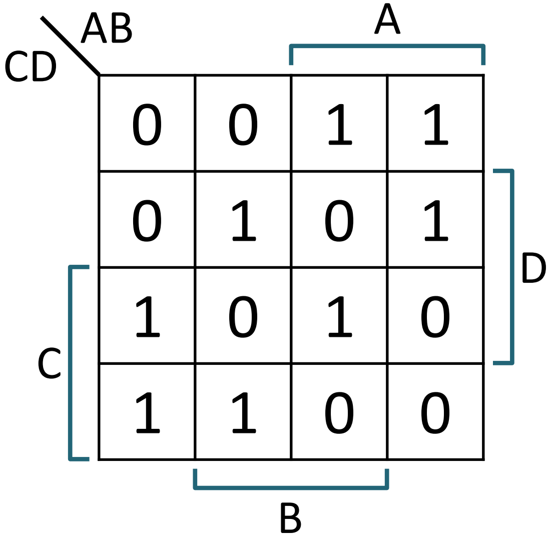
Design Example: 2-bit Adder

K-map for X



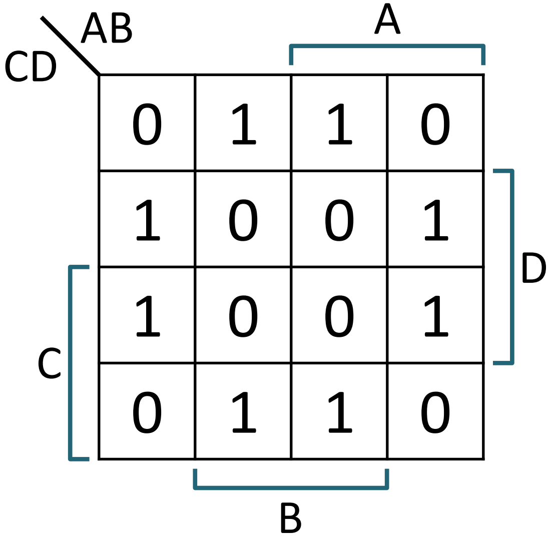
0	0	0	0
0	0	1	0
0	1	1	1
0	0	1	1

K-map for Y



0	0	1	1
0	1	0	1
1	0	1	0
1	1	0	0

K-map for Z



0	1	1	0
1	0	0	1
1	0	0	1
0	1	1	0

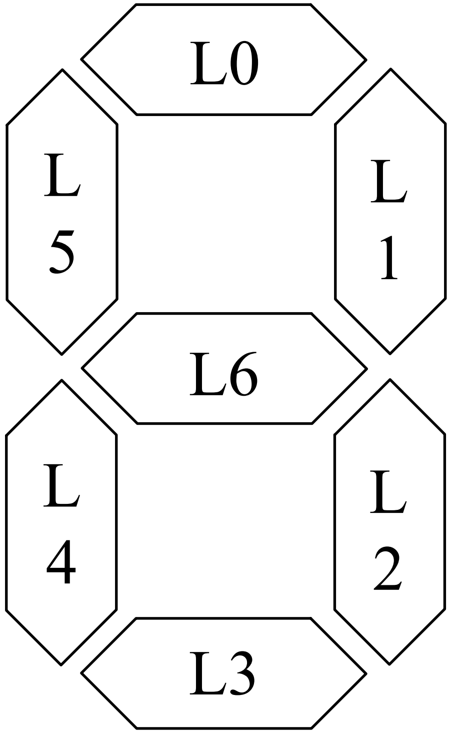
$X =$

$Y =$

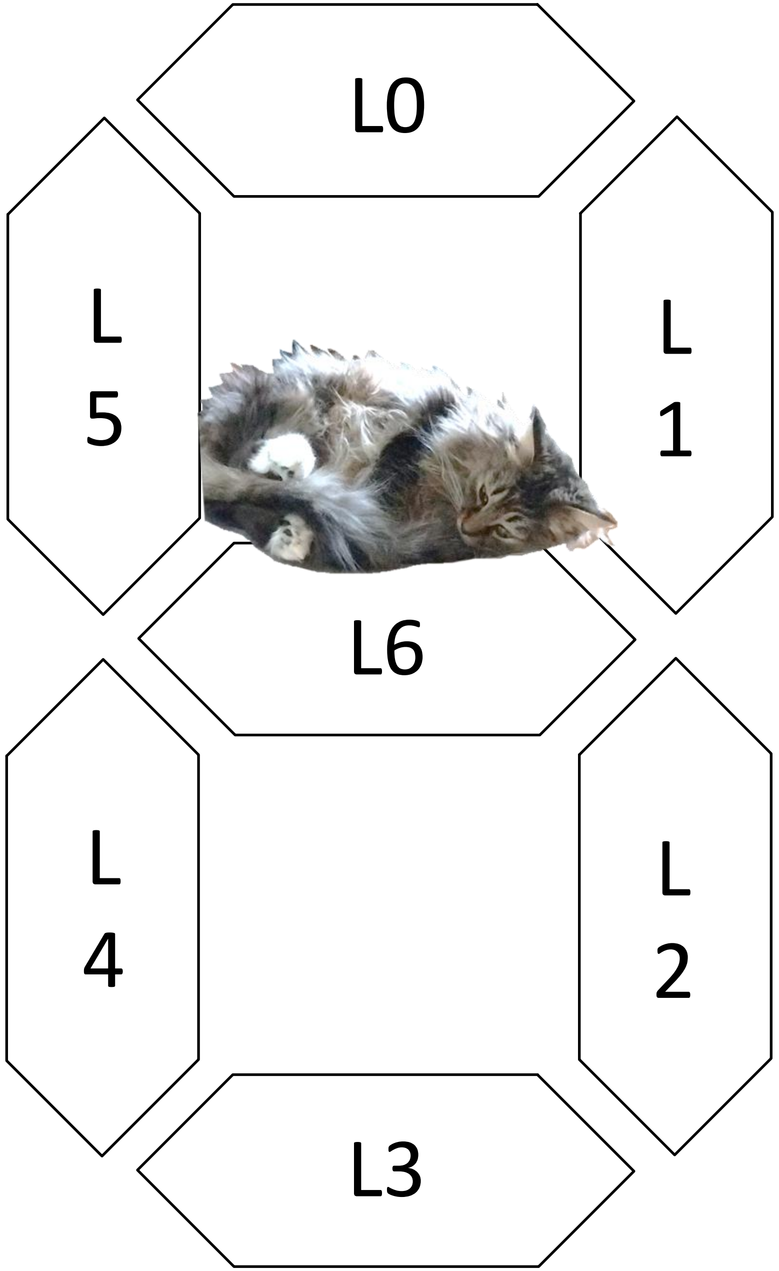
$Z =$

Case Study: Seven-Segment Display

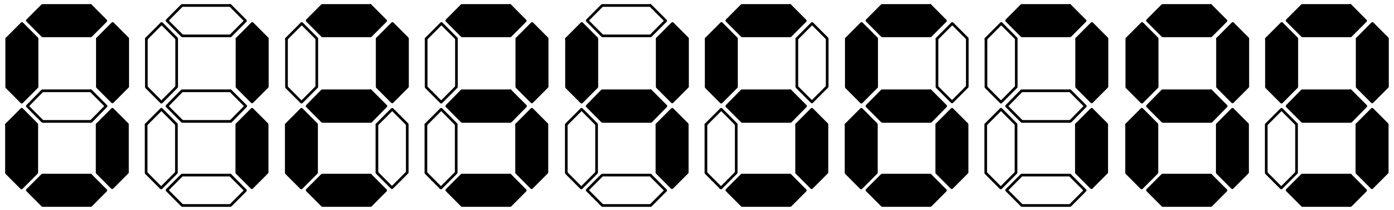
- Chip to drive digital display



B3	B2	B1	B0
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



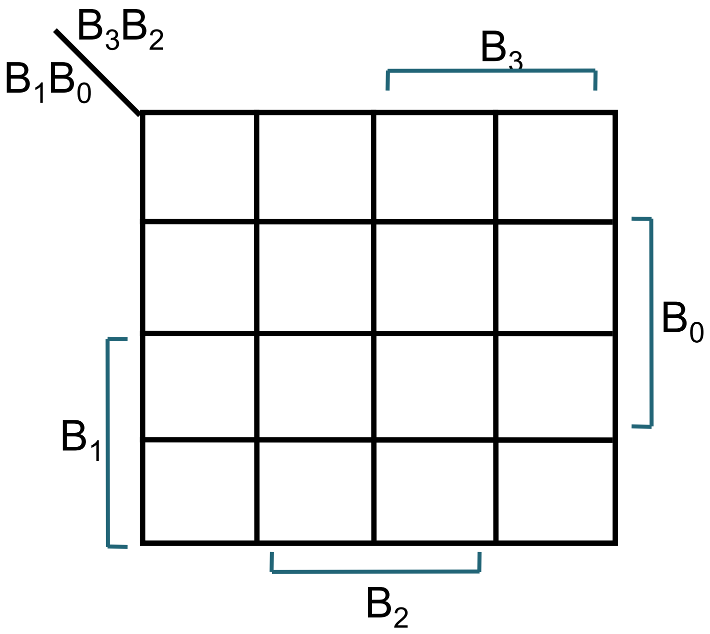
B3	B2	B1	B0	Val	L0	L1	L2	L3	L4	L5	L6
0	0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	1	0	1	1	0	0	0	0
0	0	1	0	2	1	1	0	1	1	0	1
0	0	1	1	3	1	1	1	1	0	0	1
0	1	0	0	4	0	1	1	0	0	1	1
0	1	0	1	5	1	0	1	1	0	1	1
0	1	1	0	6	1	0	1	1	1	1	1
0	1	1	1	7	1	1	1	0	0	0	0
1	0	0	0	8	1	1	1	1	1	1	1
1	0	0	1	9	1	1	1	1	0	1	1

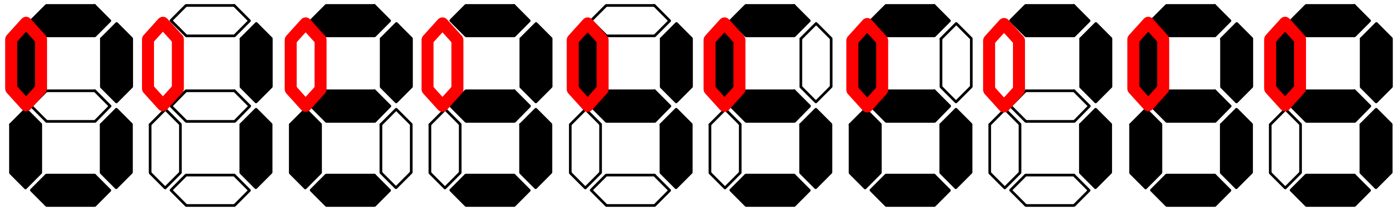


Case Study: Seven-Segment Display

- 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





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
         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;
         default: leds = 7'bX;
      endcase
endmodule
```

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###…`
  - `n` is bit width, `b` is radix/base, `#`s are the actual digits
  - *e.g.*, `4'd12`, `4'b1100`, `4'hC`

- Concatenation: `{sig, …, sig}`
  - Ordering matters; result will have combined widths of all signals

- Replication operator: `{n{m}}`
  - repeats value `m`, `n` times

Verilog “control flow”

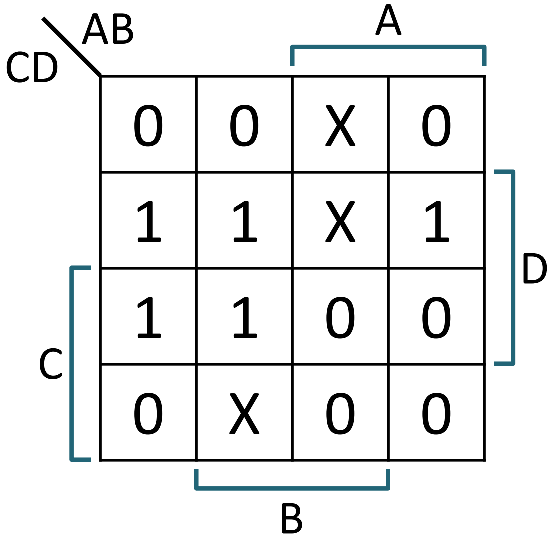
- Verilog has synthesizable **if** and **case** statements
  - In hardware, each “clause” exists in parallel and is always computing its output whether the condition is true or not
  - A **multiplexer (“mux”)** is placed physically after these blocks to electrically connect one of them to the output 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!

Don’t Cares

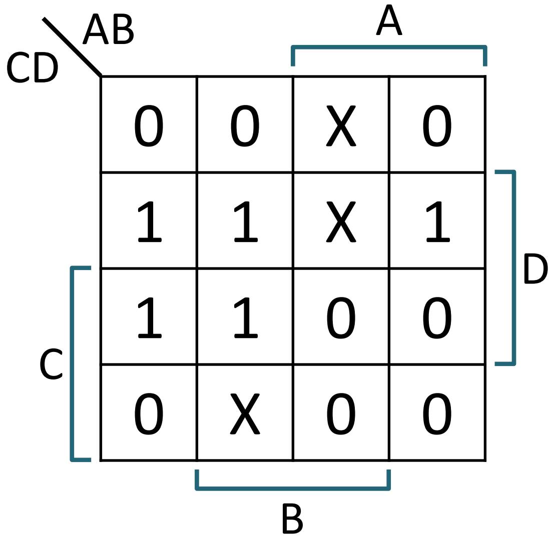
- Use symbol ‘X’ to mean it can be either a 0 or 1
    - Make choice to simplify final expression



0	0	X	0
1	1	X	1
1	1	0	0
0	X	0	0

Let all $X = 0$:

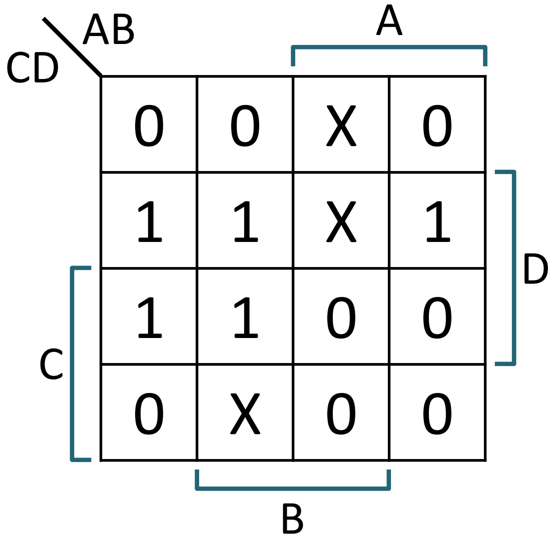
$F =$



0	0	X	0
1	1	X	1
1	1	0	0
0	X	0	0

Let all $X = 1$:

$F =$



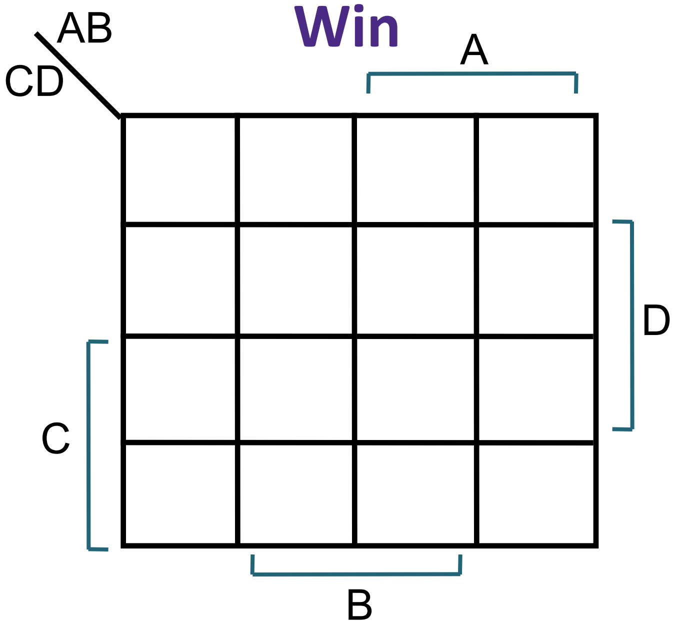
0	0	X	0
1	1	X	1
1	1	0	0
0	X	0	0

Choose wisely:

$F =$

Exercise for the reader: Rock-Paper-Scissors

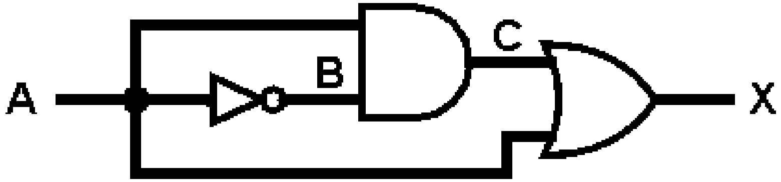
- Rock (00), Paper (01), Scissors (10) for two players P0 and P1
- **Output:** Win = Winner's ID (0/1)
  Tie = 1 if Tie, 0 else

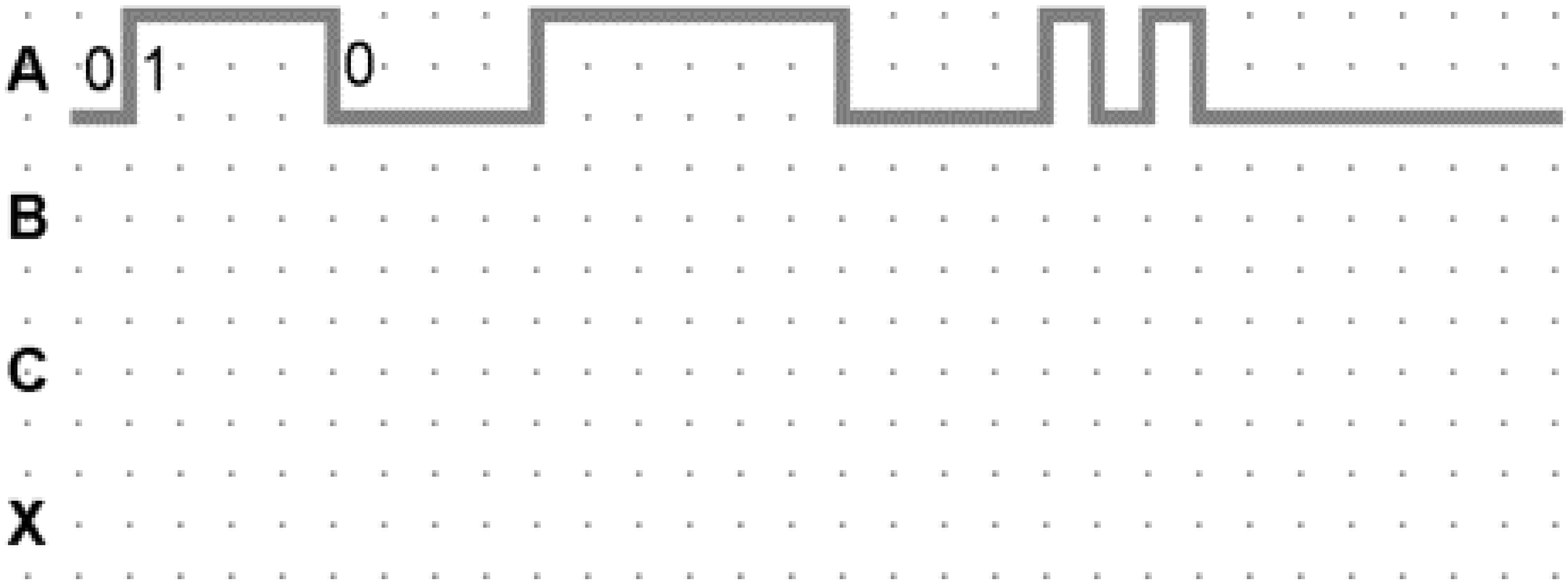


P1		P0			
A	B	C	D	Win	Tie
0	0	0	0		
0	0	0	1		
0	0	1	0		
0	0	1	1		
0	1	0	0		
0	1	0	1		
0	1	1	0		
0	1	1	1		
1	0	0	0		
1	0	0	1		
1	0	1	0		
1	0	1	1		
1	1	0	0		
1	1	0	1		
1	1	1	0		
1	1	1	1		

Exercise for the reader: Circuit hazards

- Let the CL delays be 1 tick (NOT) and 3 ticks (AND, OR). How many ticks is the signal X high?





Exercise for the reader: Verilog Signals

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

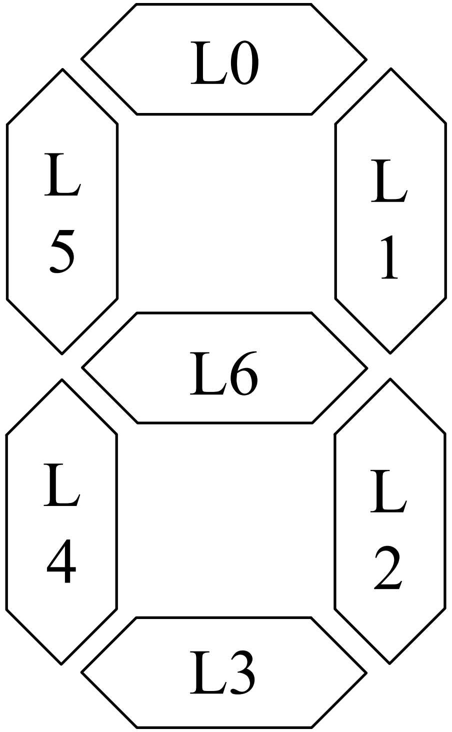
- What's the value of `pear`?

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

```
    assign orange =
```

Exercise for the reader: 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
```