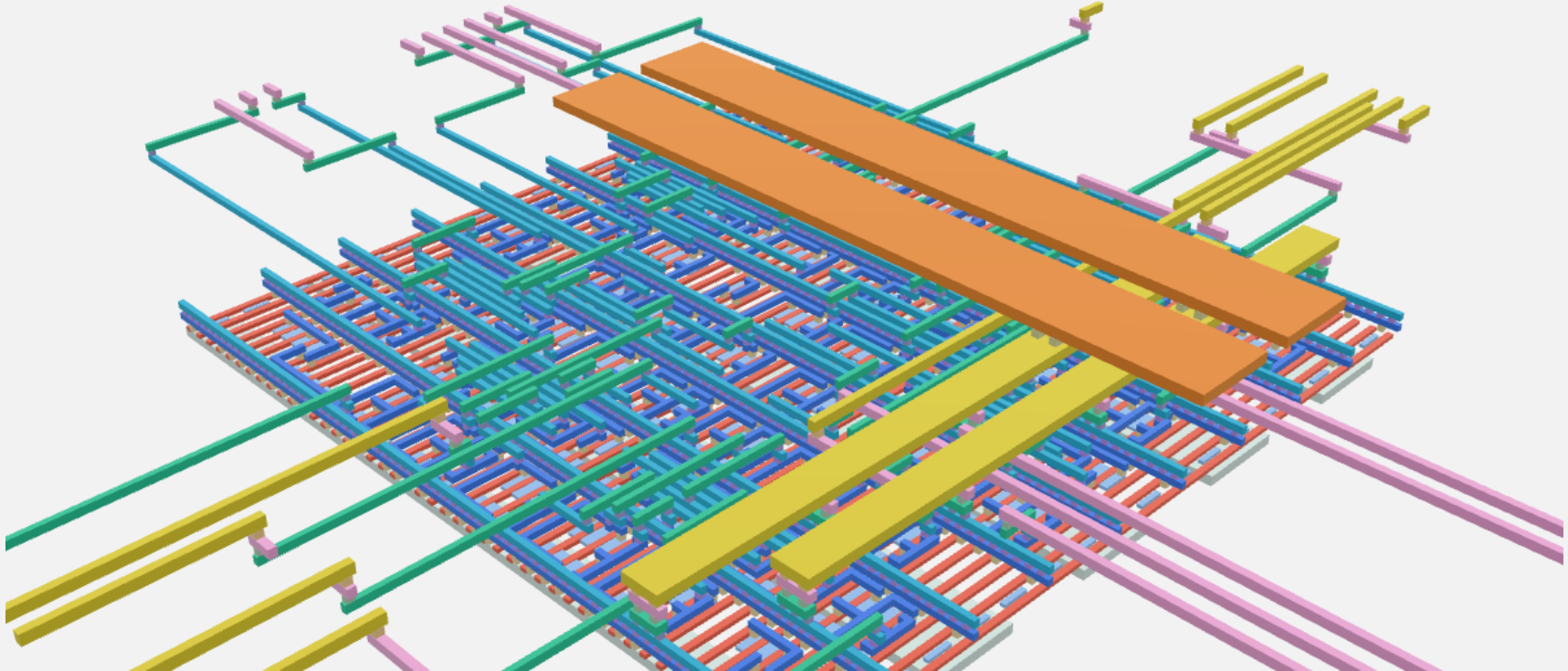
# 2.1
# CSE140 – Digital Design

2026 Summer Session 1

**Abarajithan G**



# Daily reminder:

- Open FeedbackFruits
  Canvas → CSE140 → Assignments → Slides 2.1

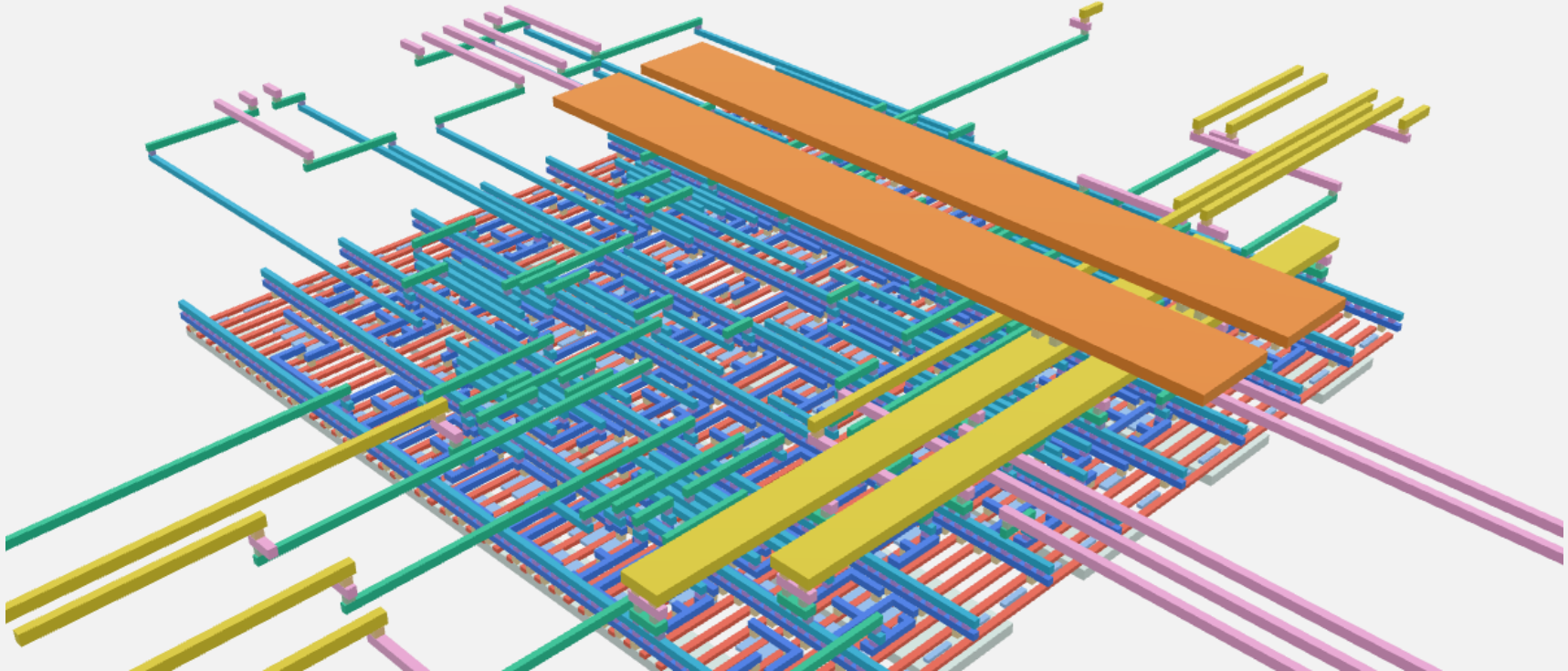
- Attendance is mandatory

- Paper exams follow assignments. Don’t use AI

# Last week’s feedback

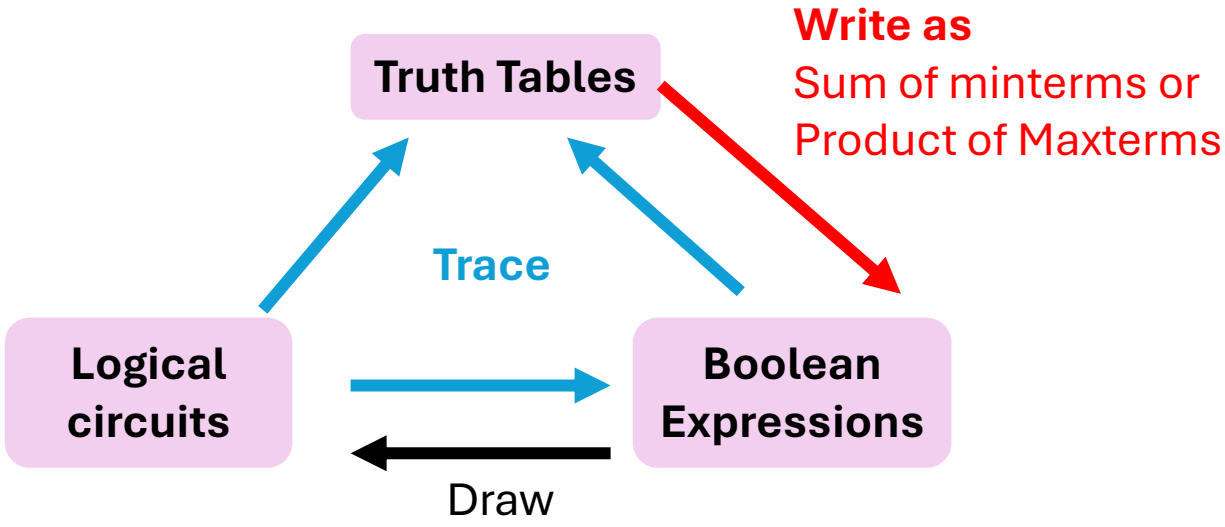
- Some people liked we went through Boolean algebra identities, without assuming those are known
- Some people said pacing can be improved.
- We will cover K-maps, numerical representation and some SystemVerilog today.

# Recap & K-maps



# Summary

- Boolean functions can be represented in 3 ways: Truth tables, circuits, expressions
- Canonical forms (unique representations)
  - Truth Table
  - Sum of Minterms
  - Product of Maxterms

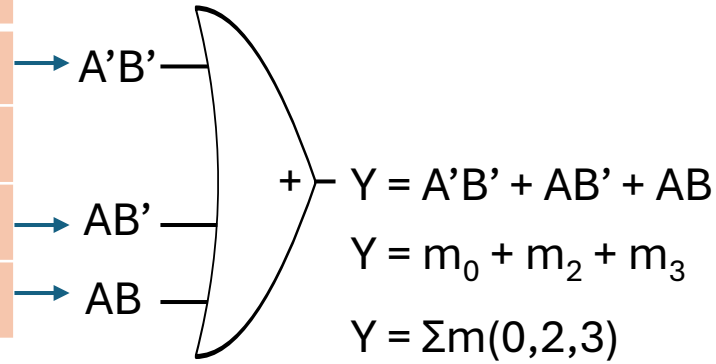



- **Finding the smallest circuit**
  - Simplify an expression using identities
  - K-maps
- But even these are limited to AND,OR,NOT

# Canonical Forms: Sum of Minterms

- Truth table → expression?
- Truth Table is a canonical representation – because it's unique
- Can we create a canonical algebraic expression?

| minterm | i | A | B | Y |
|---|---|---|---|---|
| A'B' | 0 | 0 | 0 | 1 |
| A'B | 1 | 0 | 1 | 0 |
| AB' | 2 | 1 | 0 | 1 |
| AB | 3 | 1 | 1 | 1 |




- **Minterm**
  - product that includes all input variables
  - N inputs → $2^N$ minterms.
  - $\mathbf{m_i}$ : minterm of the $\mathbf{i^{th}}$ row of truth table (i in binary).
  - If A=0, B=1, minterm is A'B. If A=1,B=1, minterm is AB
- Truth table → expression: Expression is OR of all minterms where output is 1 in truth table

# NAND: Universal Gate

- Fundamental gate that can be used to construct any Boolean function
- NAND(A,A) = (A.A)' = A' = NOT(A) – can be used to create not gate

- Steps to make any function using NAND gates:
  1. Write the truth table
  2. Write it as sum of minterms: $m_x + m_y + m_z$
  3. Invert the sum twice and distribute the complement to make it NAND of NANDs $(m_x + m_y + m_z)'' = (m_x' m_y' m_z')'$
  4. Implement the NOT gates inside each minterm using NAND

| i | A | B | C | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 1 | 1 |
| 4 | 1 | 0 | 0 | 1 |
| 5 | 1 | 0 | 1 | 0 |
| 6 | 1 | 1 | 0 | 1 |
| 7 | 1 | 1 | 1 | 0 |

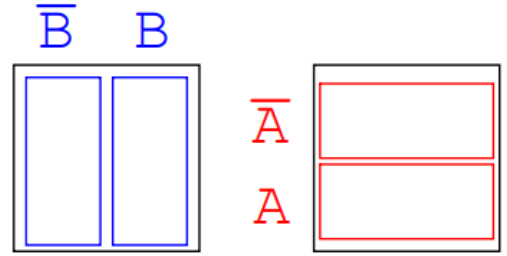
- Example: Implement using NAND gates

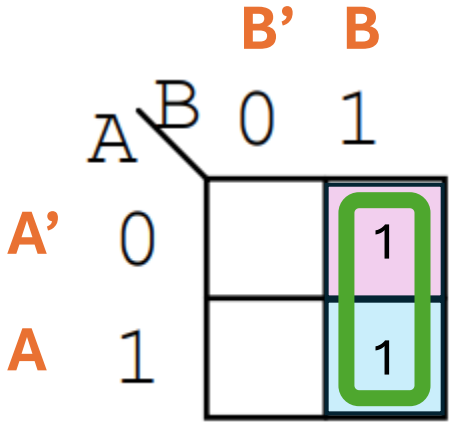
  Y = A'B'C' + A'BC + AB'C' + ABC'  [sum of minterms]

# Karnaugh maps

- A visual way to simplify circuits using AND,OR,NOT
- **Steps:**
  - Draw the map & Put 1s where output has ones
  - Make the biggest rectangular/square groups of 1s (must be power of 2)
  - Starting from the biggest, for each rectangle:
    - write the terms whose values don't change across the rectangle
    - sum the products, until all 1s are covered

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



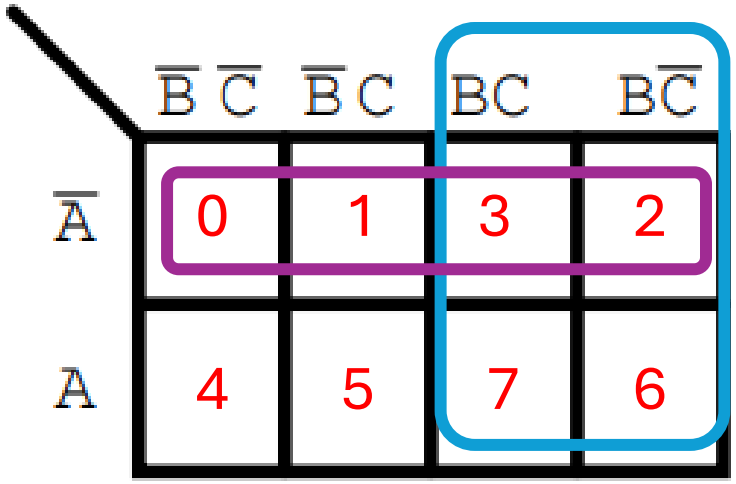




- B=1 across the rectangle of size 2
- Term is B
- F = B

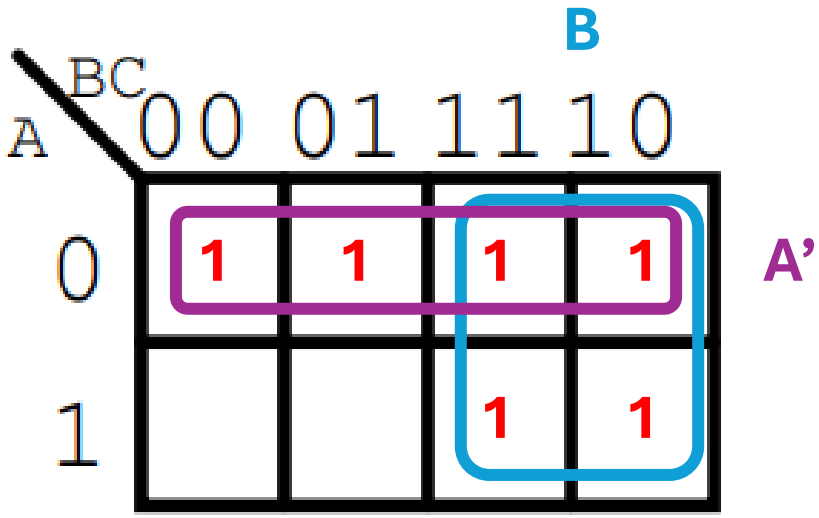
# 3-variable K-map

- **Important:** Columns are enumerated as gray code: 00, 01, 11, 10
- Rows & columns wrap around – need to draw rectangles accordingly

| i | A | B | C | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 0 | 1 | 1 | 1 |
| 4 | 1 | 0 | 0 | 0 |
| 5 | 1 | 0 | 1 | 0 |
| 6 | 1 | 1 | 0 | 1 |
| 7 | 1 | 1 | 1 | 1 |




Each cell corresponds to a minterm




1 means that minterm exists in the sum

**F = A' + B**

# **Activity 2 –** Derive the expression through a K-map

- **Important:** Edges of the K-map wraps around (check the minterms)

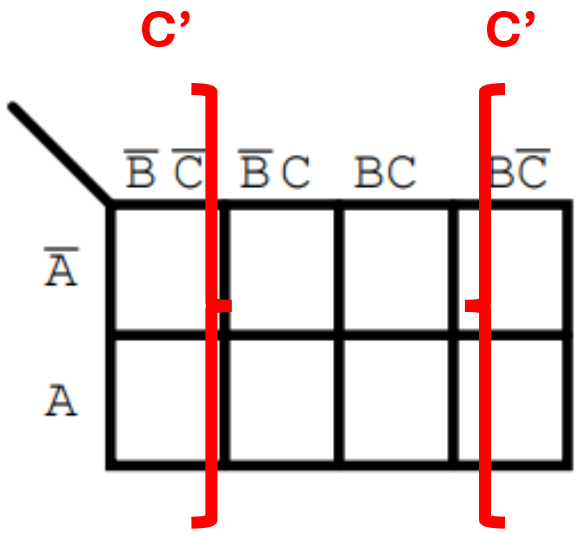
| i | A | B | C | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 0 | 1 | 1 | 1 |
| 4 | 1 | 0 | 0 | 1 |
| 5 | 1 | 0 | 1 | 0 |
| 6 | 1 | 1 | 0 | 1 |
| 7 | 1 | 1 | 1 | 0 |

**DONE?**

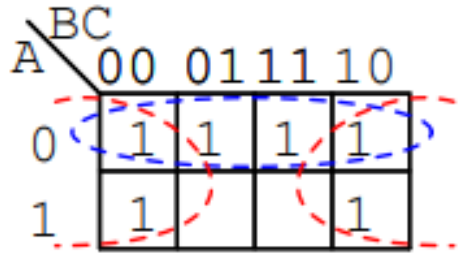
# **Activity 2 –** Derive the expression through a K-map

- **Important:** Edges of the K-map wraps around (check the minterms)

| i | A | B | C | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 0 | 1 | 1 | 1 |
| 4 | 1 | 0 | 0 | 1 |
| 5 | 1 | 0 | 1 | 0 |
| 6 | 1 | 1 | 0 | 1 |
| 7 | 1 | 1 | 1 | 0 |




$$Out = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + \overline{A}BC + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + AB\overline{C}$$

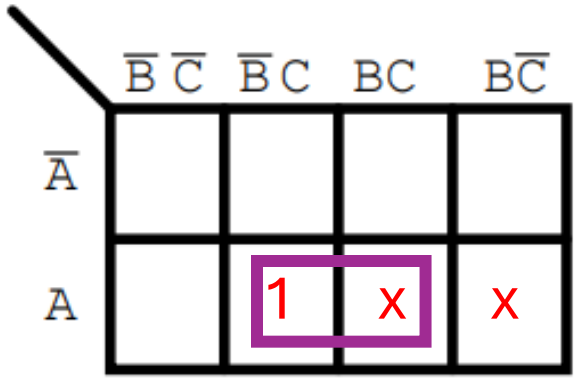



$$Out = \overline{A} + \overline{C}$$

# Don’t care Conditions

- A counter counts 0,1,2,3,4,5,0,1,2… indefinitely.
- We need a circuit to detect whenever it reaches 5.

| i | A | B | C | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 | 0 |
| 5 | 1 | 0 | 1 | 1 |
| 6 | 1 | 1 | 0 | x |
| 7 | 1 | 1 | 1 | x |




Y =AB’C

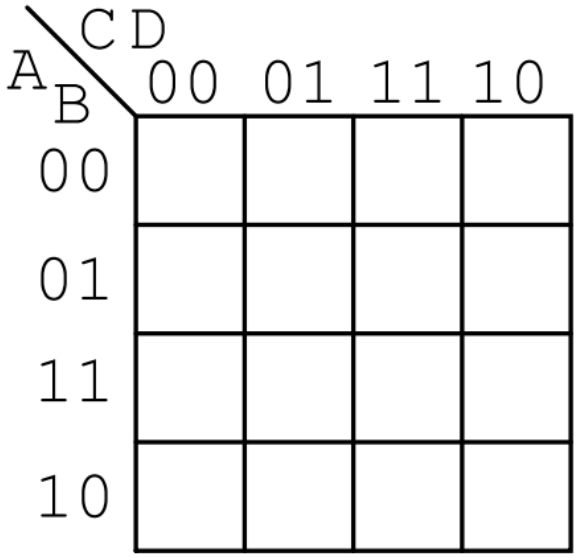
Utilizing don’t cares:

**Y =AC**

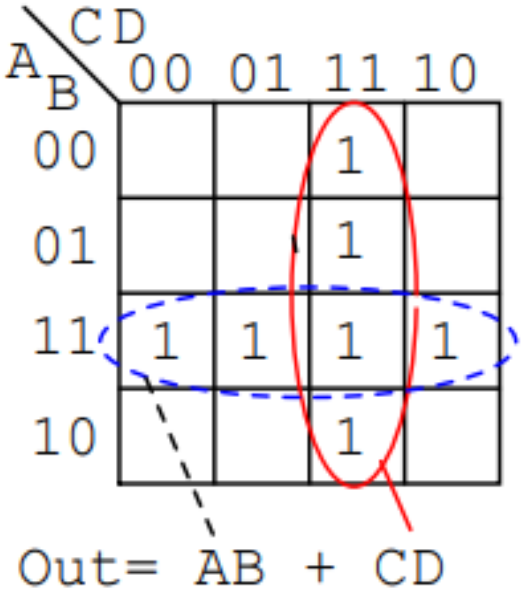
The counter will never be 6 or 7.
So, our circuit can ignore those input cases, if it makes the circuit simpler

# 4-variable K-map

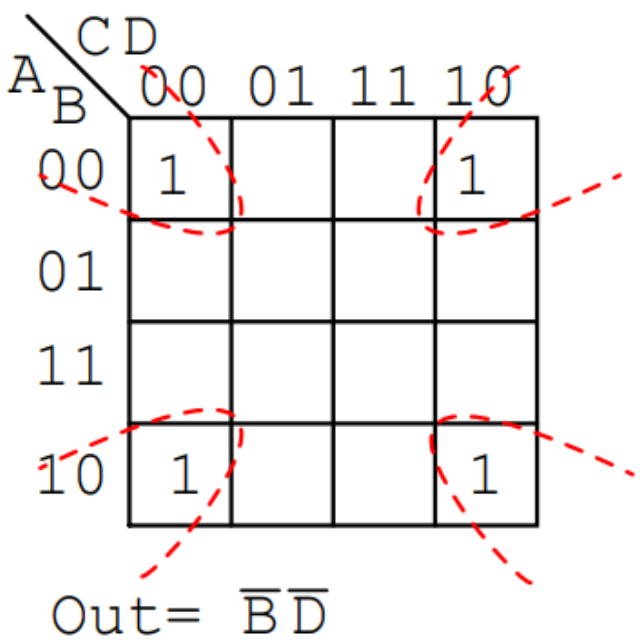
- Both rows and columns are enumerated as gray code: 00, 01, 11, 10, and they wrap

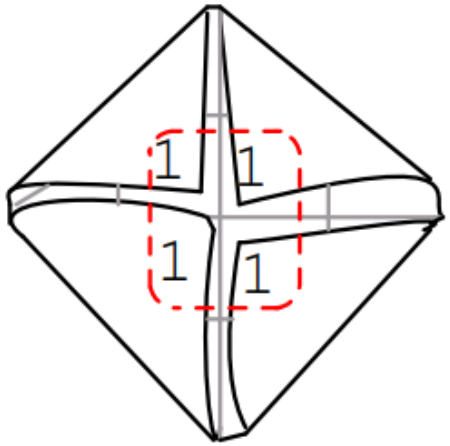


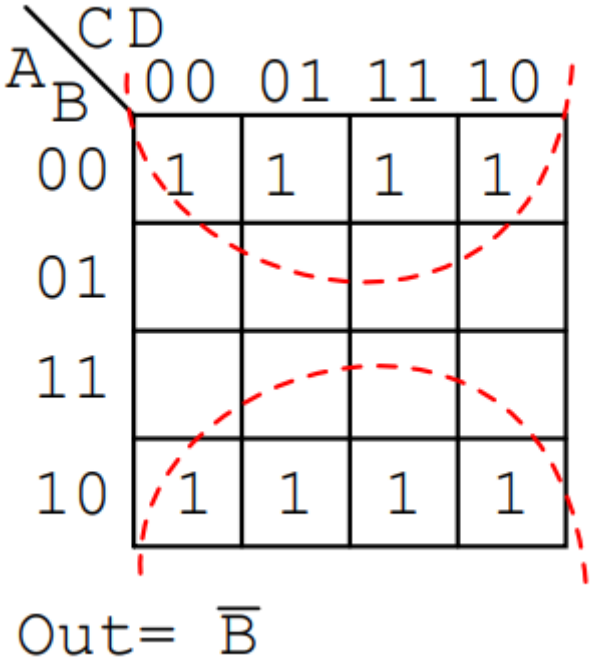

$$Out = \overline{A}\,\overline{B}CD + \overline{A}B\overline{C}D + ABCD + A\overline{B}CD + AB\overline{C}\,\overline{D} + AB\overline{C}D + ABC\overline{D}$$



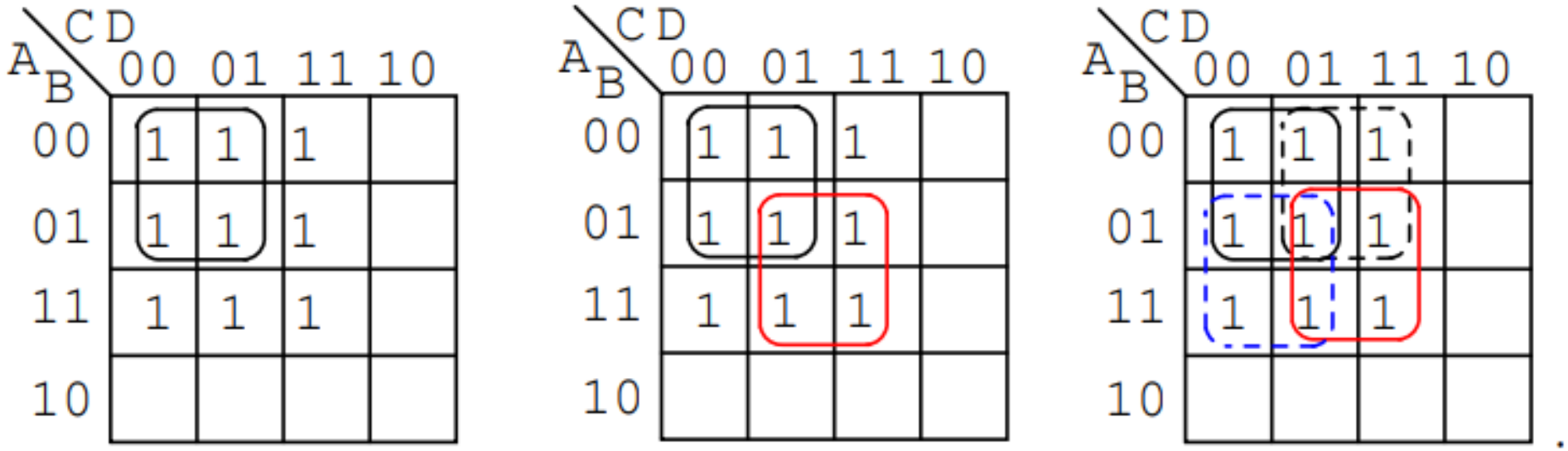
# Wrap arounds



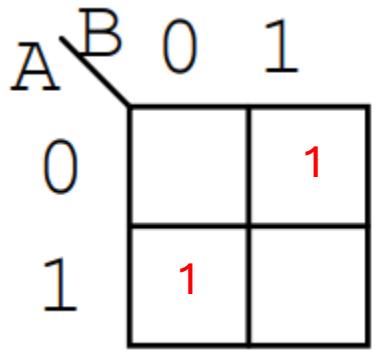
# Another Example




$$\text{Out} = \overline{A}\,\overline{C} + \overline{A}D + B\overline{C} + BD$$

# Limits of K-maps

| i | A | B | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 2 | 1 | 0 | 1 |
| 3 | 1 | 1 | 0 |




Y = AB’ + A’B

**Y = A xor B is simpler**

- K-maps only help to boil down a circuit to AND, OR, NOT gates
- But modern PDKs have standard cells (lego blocks) of compound gates
- Synthesis tools use advanced algorithms to perform minimization

# Half Adder

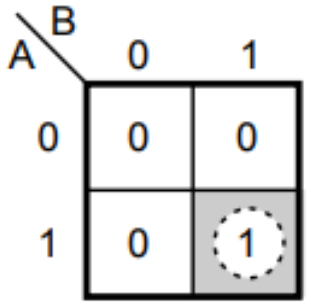
- Adding two 1-bit numbers
- Multiple outputs – treat each output separately
- Find the expression for each output

| A | B | Sum | COut |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

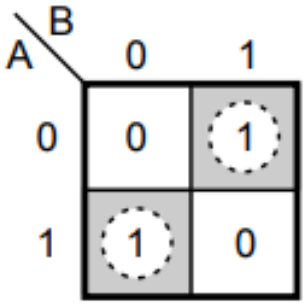
DONE?

# Half Adder

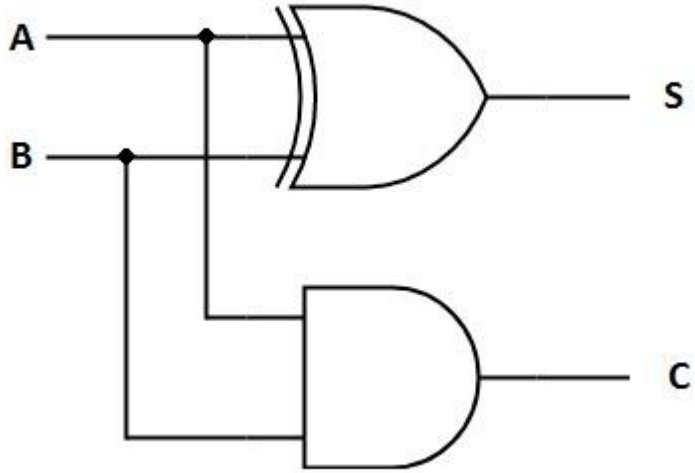
| A | B | Sum | COut |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| | | XOR | AND |




Carry = AB

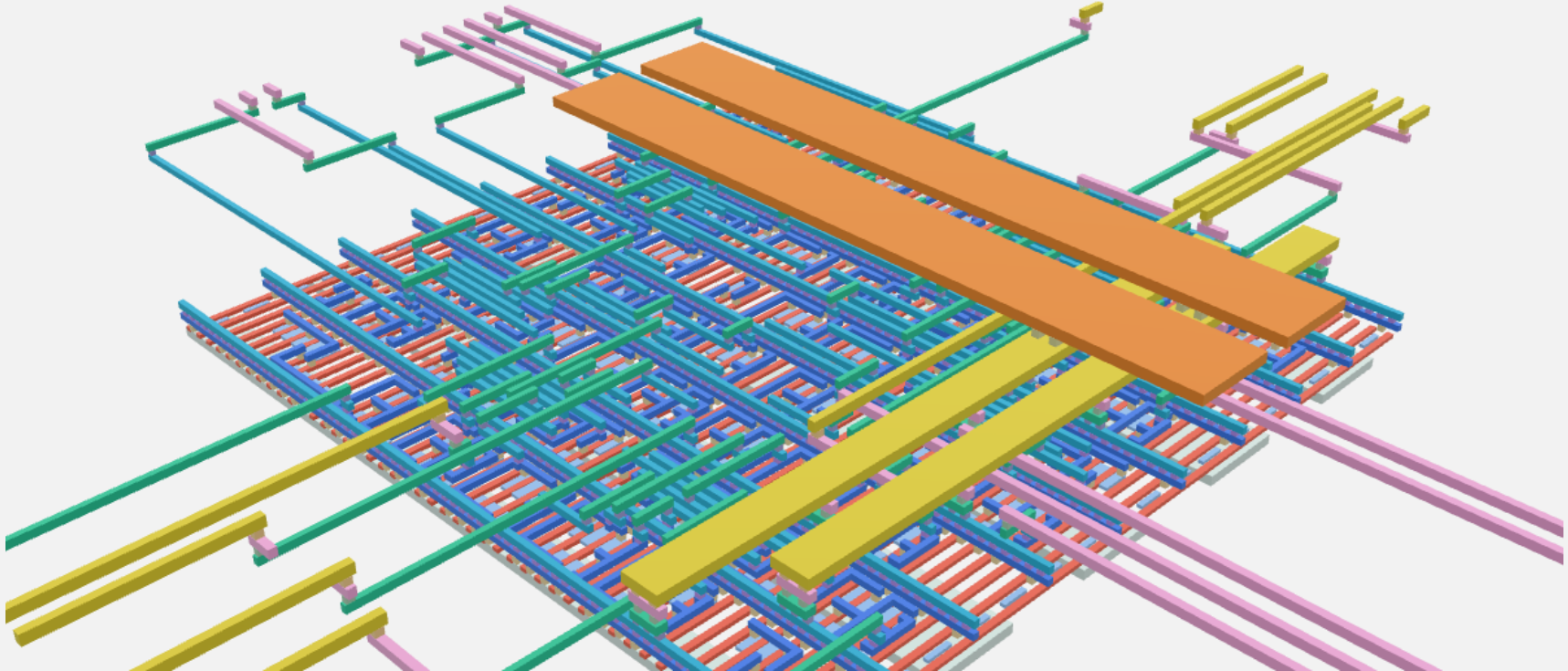



Sum = $A\overline{B} + \overline{A}B$




K-maps are limited to two-level (AND, OR) simplification

# Introduction to SystemVerilog



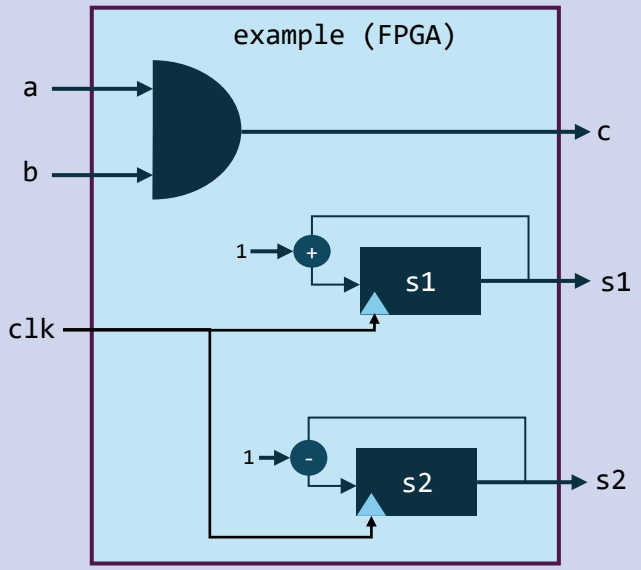
# HDL: Hardware Description Language

| Programming Language (control flow) | Hardware Description Language (HDL) (data flow) |
|---|---|

```
a = True
b = False
c = a and b
print(c)          >> False

a = True
b = True
print(c)          >> False

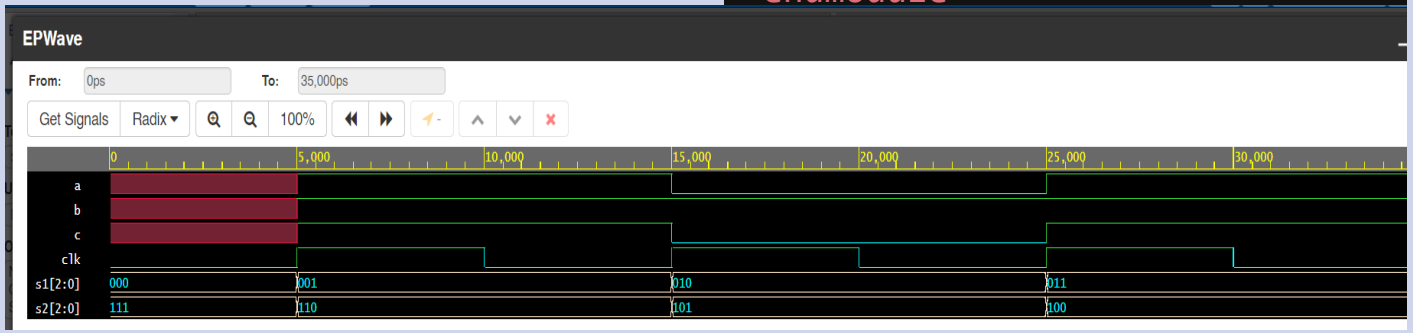
s1 = 0
for i in range(3):
    s1 = s1 + 1
    print(s1)     >> 1
                     2
                     3
s2 = 7
for i in range(2):
    s2 = s2 - 1
    print(s2)     >> 6
                     5
```




```
module example (
    input  logic clk,a,b,
    output logic c,
    output logic [3-1:0] s1=0,
    output logic [3-1:0] s2=7
  );

  always_comb c = a & b;

  always_ff @(posedge clk)
    s1 <= s1 + 1;
  always_ff @(posedge clk)
    s2 <= s2 - 1;
endmodule
```



# An HDL is not a Programming Language

| Programming Language | Hardware Description Language (HDL) |
| --- | --- |
| Step by step **instructs** an existing processor what to do | **Textual description of your circuit.**<br>We will describe our own processor with an HDL, and make it execute a programming language we created. |
| **Control flow:** Read and executed line by line | Tells the synthesis tool how to make the circuit.<br>**Data Flow:** After implementation, you can give signals and get outputs |
| **Compiling:**<br>High level code gets translated to the instructions for a particular processor family | **Compiling (Synthesis):**<br>Translating a high-level design (RTL) into a low-level, target-specific design (netlist) |

# One Language, Two Objectives

| Design | Verification (Simulation) |
| --- | --- |
| Describes a circuit<br>• RTL – high level<br>• Netlist – low level | • Feeds inputs to the designed circuit step by step<br>• Gets outputs from the circuit<br>• Compares with expected outputs to “verify” |
| **Synthesizable** into a digital circuit. | **Not synthesizable**, testbenches are only simulated.<br>Runs as a software. |
| Read by the synthesis compiler<br>• FPGA: Vivado/Quartus synthesis,<br>• ASIC: Synopsys Design Compiler, Cadence Genus | Read by the simulator<br>• FPGA: Qsim (Altera Quartus), ModelSim, Vivado<br>• ASIC: Synopsys VCS, Cadence Xcelium |

# Hello SystemVerilog!

```
// hello_tb.sv

module hello_tb;

  initial begin
    $display("Hello, SystemVerilog!");
    $finish;
  end

endmodule
```

```
verilator --binary hello_tb.sv
```

Compile using verilator

```
./obj_dir/Vhello_tb
```

Run compiled simulation

- This is software code
- SystemVerilog is used to describe hardware (Synthesizable RTL), and also to write software code (non synthesizable testbench) to test it
- Verilator – open source, industry-standard simulator

DONE?

# Simulating signals

```sv
// and_tb.sv
`timescale 1ns/1ps

module and_tb;
  bit A, B, C;
  always_comb C = A && B;


  initial begin
    $dumpfile("waveform.vcd"); $dumpvars(0, and_tb);

        A = 0; B = 0; #1; $display("A=%b B=%b C=%b", A, B, C);
    #9  A = 0; B = 1; #1; $display("A=%b B=%b C=%b", A, B, C);
    #9  A = 1; B = 0; #1; $display("A=%b B=%b C=%b", A, B, C);
    #9  A = 1; B = 1; #1; $display("A=%b B=%b C=%b", A, B, C);

    #9 $finish;
  end
endmodule
```

```
verilator --binary --timing --trace-vcd and_tb.sv
```

Compile using verilator

```
./obj_dir/Vand_tb
```

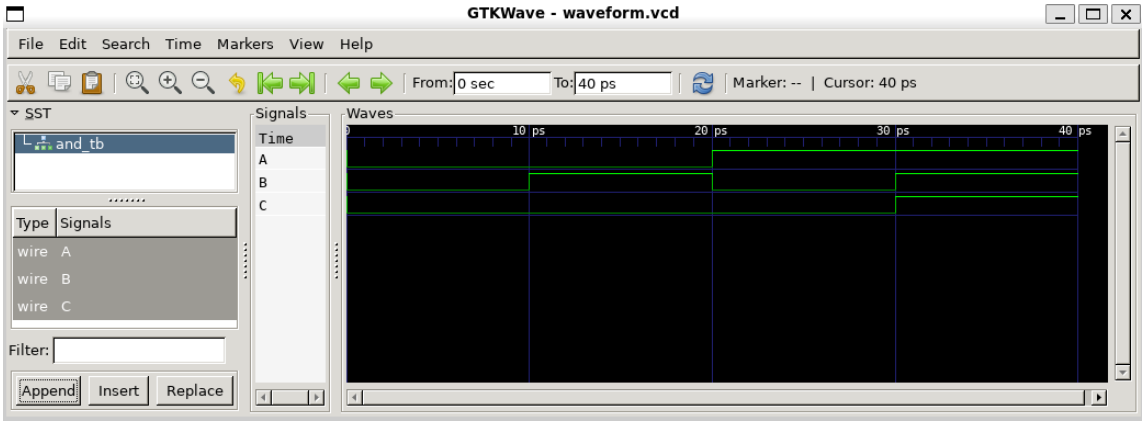
Run compiled simulation

```
gtkwave waveform.vcd
```

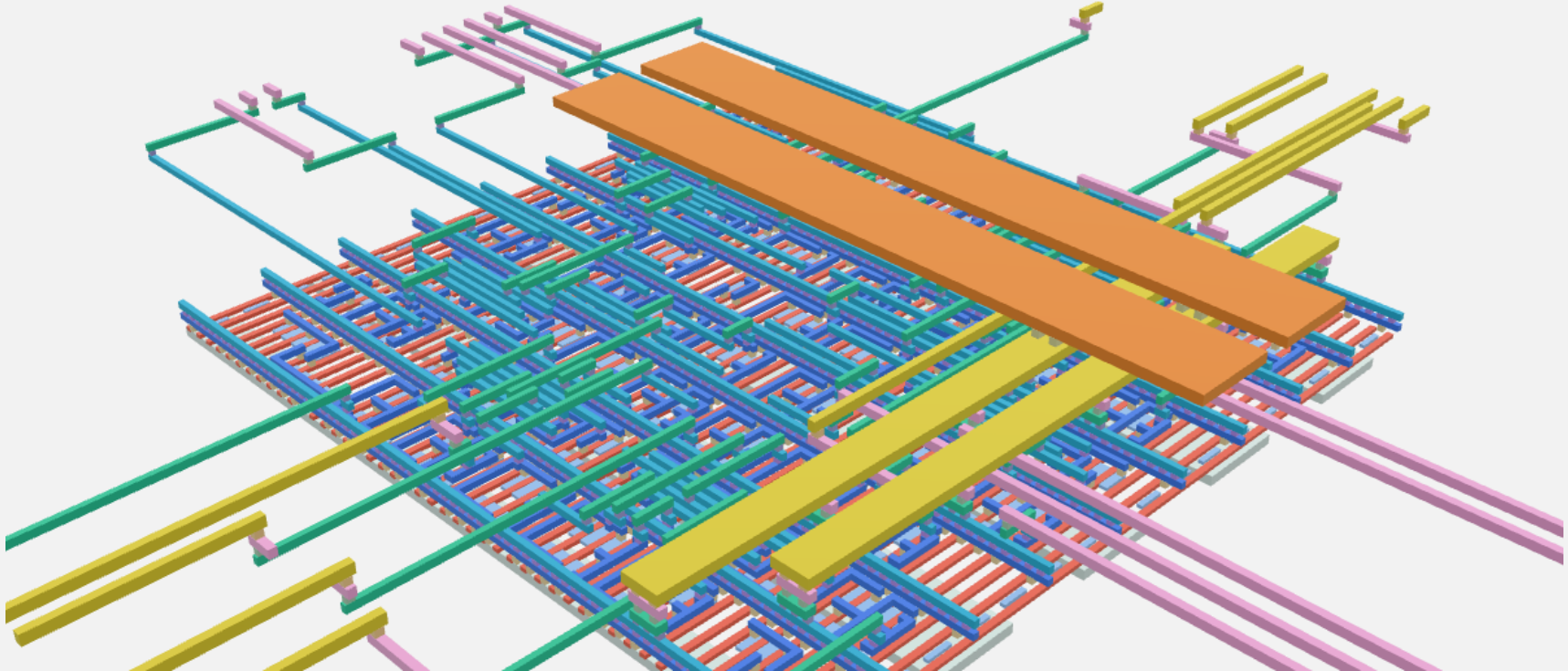
View waveform

**DONE?**

- Always_comb – create combinational (stateless) circuits
- Dumpfile, dumpvars – to dump waveform
- Display – print statement, like C
- #9 – wait for 9ns

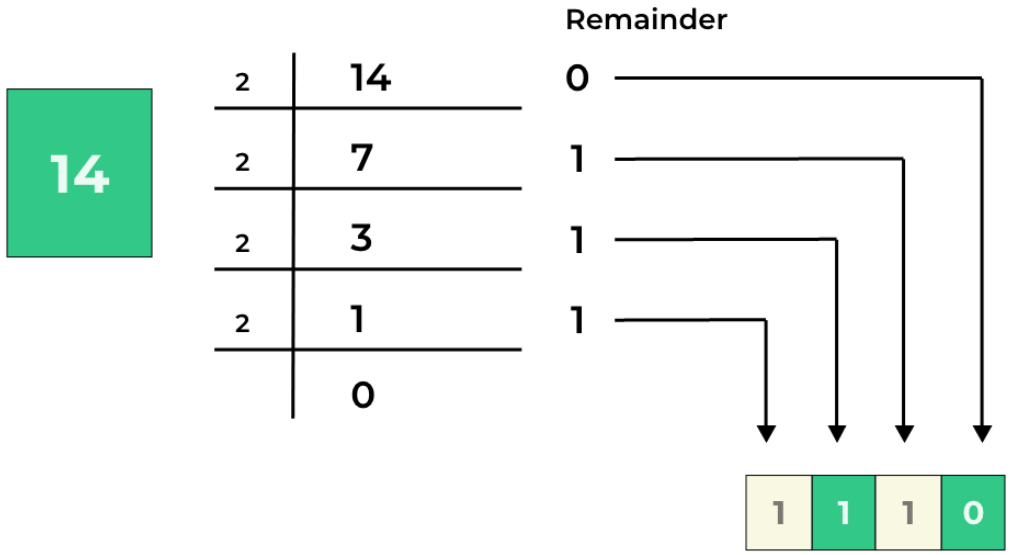


# Number Representation



# Decimal to binary

- Repeatedly divide a number by 2, keeping track of remainders
- Write the last quotient and all reminders




https://prepinsta.com/c-program/decimal-to-binary-conversion/

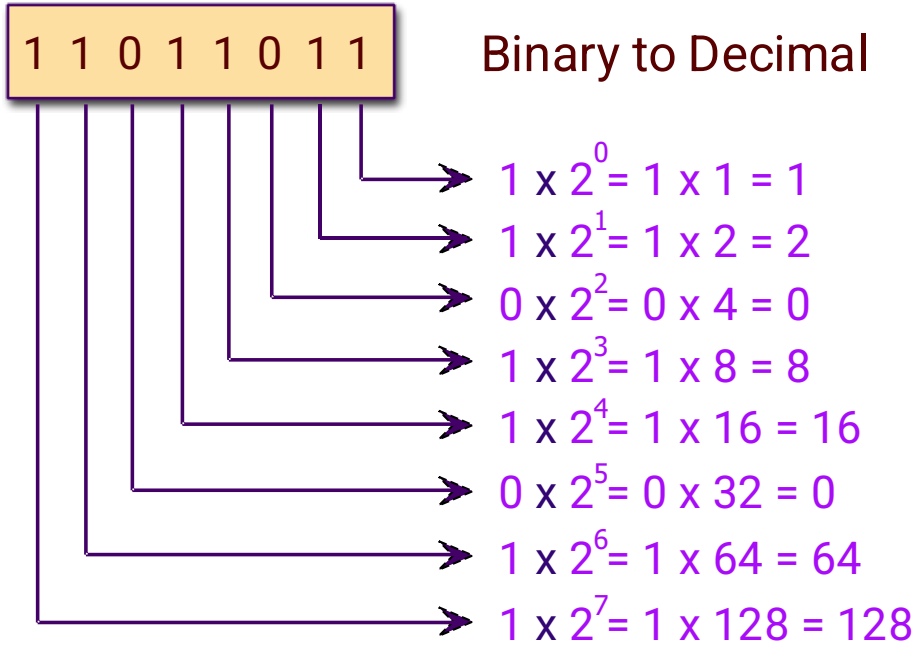
**172 = 10101100**

| | | |
|---|---|---|
| 2 | 172 | |
| 2 | 86 | 0 |
| 2 | 43 | 0 |
| 2 | 21 | 1 |
| 2 | 10 | 1 |
| 2 | 5 | 0 |
| 2 | 2 | 1 |
| | 1 | 0 |

https://byjus.com/gate/decimal-to-binary-conversion-notes/

# Binary to Decimal

- Multiply ith digit (from LSB – least significant bit) by 2^i and add




$1 + 2 + 8 + 16 + 64 + 128 = 219$

$(11011011)_2 = (219)_{10}$

© w3resource.com

# Unsigned Integer - Addition

**A** **B** **A+B**

$$10001 + 11101 = 101110:$$

$$\begin{array}{cccccc} & 1 & & & & 1 \\ & 1 & 0 & 0 & 0 & 1 \\ + & 1 & 1 & 1 & 0 & 1 \\ \hline 1 & 0 & 1 & 1 & 1 & 0 \end{array}$$

How many bits in each input?
How many bits in the output?

$10111 + 110101$ = **Answer?**

# Unsigned Integer - Addition

$$
\begin{array}{cccccccc}
10111 + 110101 = 1001100: & & & & & & & \\
 & 1 & 1 & & 1 & 1 & 1 & \\
 & & & 1 & 0 & 1 & 1 & 1 \\
+ & & 1 & 1 & 0 & 1 & 0 & 1 \\
\hline
 & 1 & 0 & 0 & 1 & 1 & 0 & 0
\end{array}
$$

# Unsigned Integer - Largest numbers

- Largest unsigned number you can represent in B bits = **$2^B-1$**
- Example with 3 bits:
  - 0,1,2,3,4,5,6,7 ; $2^3=8$; max = $2^3-1 = 7$
  - This happens when all bits are 1s. 7 = 111
- Conversely, bits needed to represent a number X = ceil(log2(X+1)) = **clog2(X+1)**
- Example: 14
  - log2(14+1) = 3.90…
  - clog2(15) = 4 bits needed
- Example: 8; clog2(8+1) = 4 bits needed

# Bit growth during addition

- If you add two 3-bit integers, you need 4 bits to represent the output correctly.
- Why? – consider the worst case. Add the largest numbers in 3 bits:
  - 7+7 = 14 = 1110 (4 bits)

- **How many bits  do you need to represent the sum of N, 3-bit numbers?**

# Bit growth during addition

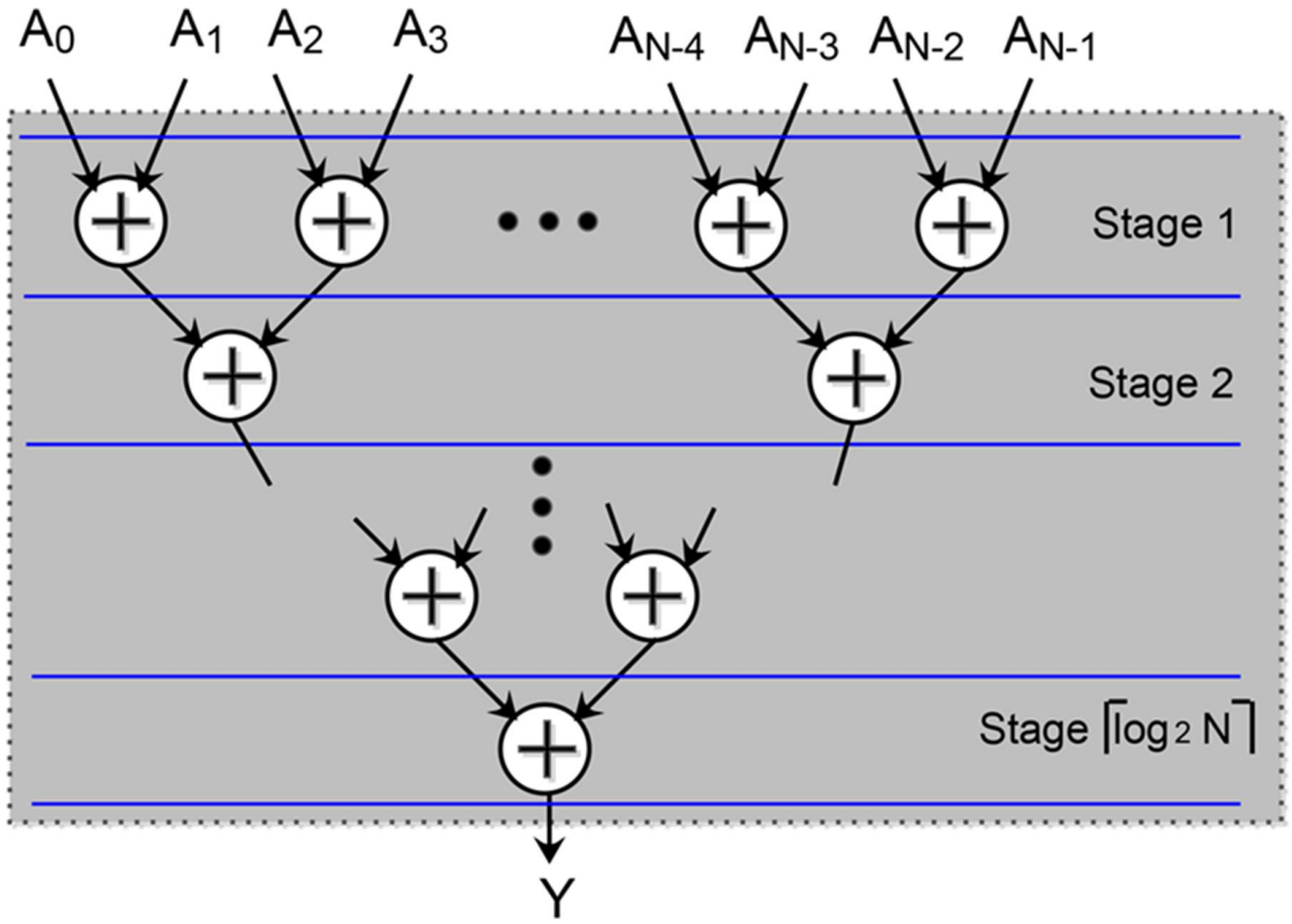
- How many bits  do you need to represent the sum of N, B-bit numbers?
- Consider the worst case: add N largest numbers ($2^B$-1) together.
- Largest output = ($2^B$-1).N
- Bits needed

  = clog2(($2^B$-1)*N + 1)

  = ceil($\log_2$($2^B$-1) + $\log_2$ (N))

  ≈ **B + clog2(N)**

# Adder Tree




B bits

B+1 bits

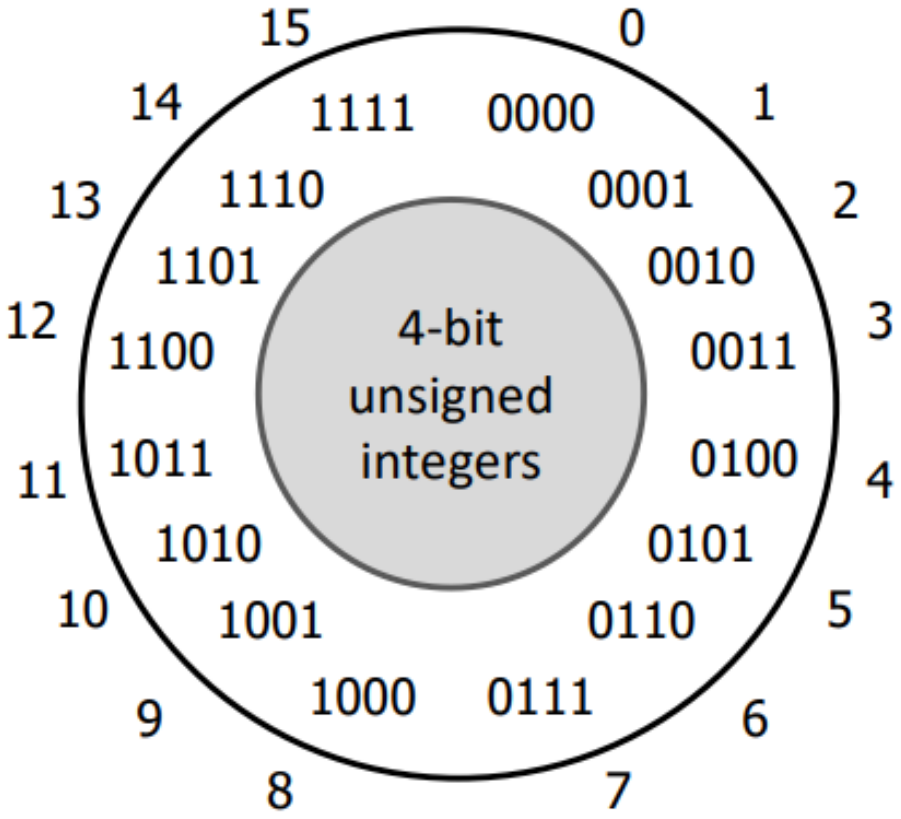
B+2 bits

B+3 bits

B+clog2(N) bits

# Unsigned Overflow

- Example with 3 bits:
  - 0,1,2,3,4,5,6,7 ; $2^3$=8; max = $2^3$-1 = 7
  - This happens when all bits are 1s. 7 = 111
  - Add 1 to the largest number, you get 0. Overflow!
  - Subtract 1 from zero, you get 7. Overflow!
- Unsigned integer of B bits:
  - Largest value: ($\mathbf{2^B-1}$)
  - Smallest value: (**0**)
- The number line becomes a circle for fixed bits.
- Example with 4 bits:



# Unsigned Integer – Multiplication

- Shift one input (A), multiply by corresponding bit in other input (B), add the results

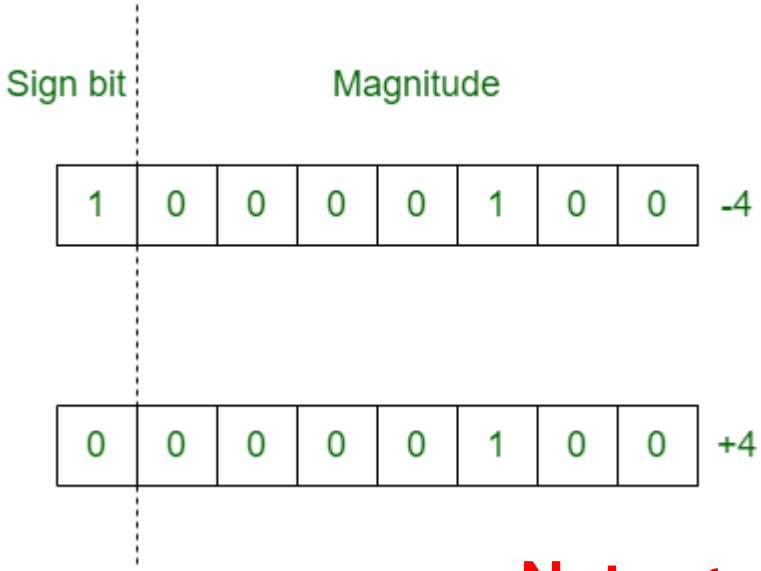
```
        1  0  1  0
      x    1  0  1
  ---------------------
        1  0  1  0
     0  0  0  0  *
+ 1  0  1  0  *  *
  ---------------------
  1  1  0  0  1  0
```

$(1010)_2 \times (101)_2 = (110010)_2$

geeksforgeeks.org/maths/binary-multiplication/

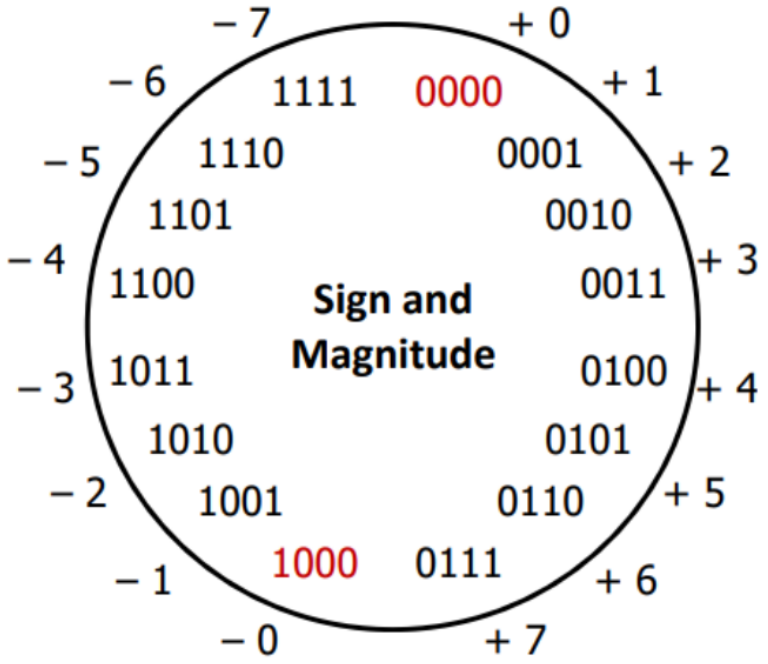
# Signed Integer (1): Sign-Magnitude

- How to represent a negative number?
- Easy way – an extra bit to indicate sign
  - 1 – negative
  - 0 - positive




**4.1 4-bit Sign and Magnitude** **Two problems**

- two representations of zero (`0b0000` and `0b1000`—negative zero??)
- arithmetic is cumbersome as negatives "increment" in the wrong direction (i.e., adding 1 to a negative number makes it more negative)



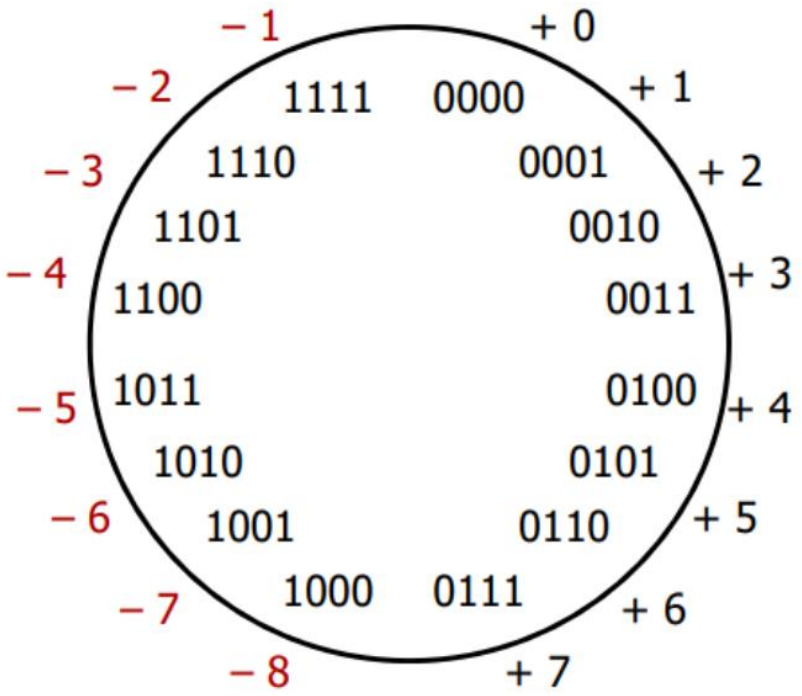

**Not a great idea – this is not the standard way**

# Signed Integer (2): Two's Complement

- Conversion:
  - Zero is zero, only one zero
  - Positive numbers – as they are
  - Negative numbers:
    - Invert the bits of a number, add one
    - -X = ~X+1
- Signed integer of B bits:
  - Largest value: ($\mathbf{2^{B-1}-1}$)
  - Smallest value: ($\mathbf{-2^{B-1}}$)
- Subtraction – negate and add

**4.2 4-bit Two's Complement**

- negative numbers "flipped" so that incrementing does what we want
- no more -0
- preserves nice property of sign and magnitude that positive encodings match unsigned encodings




Note that the most significant bit end up still indicating the sign!

# Clamping/Clipping

- The sum of N numbers of bitwidth B, needs bitwidth B+clog2(N)
- But we often use standardized bitwidths: int8, int16, int32, because hardware like CPU cannot have datapaths for varying bitwidths.
- In DSP hardware, we often accumulate into higher bitwidth to prevent error during accumulation, then “clamp” before outputting to smaller bitwidth

- **Clamping:**
  - If value is bigger than largest or smaller than smallest, pass those (saturating).
  - Otherwise pass the value

# Clamping: Activity

1. Clamp 134 from 16-bit signed integer to 8-bit signed integer
2. Clamp 134 from 16-bit unsigned integer to 8-bit unsigned integer
3. Clamp -53 from 8-bit signed integer to 6-bit signed integer

# Clamping: Activity

1. Clamp 134 from 16-bit signed integer to 8-bit signed integer
   - 8-bit signed max: 2^7-1 = 127; input outside range
   - Output = 127
2. Clamp 134 from 16-bit unsigned integer to 8-bit unsigned integer
   - 8-bit unsigned max: 2^8-1 = 255; input within range
   - Output = 134
3. Clamp -53 from 8-bit signed integer to 6-bit signed integer
   - 6-bit signed min: -2^5 = -32; input outside range
   - -32

# Daily Feedback

- One thing I liked about today’s class:
- One thing that could be improved:
- One place where I felt lost (if any):