EECS 270 W25
Discussion 4

January 30th, 2025
By: Mick Gordinier

Please fill out the Google form if you haven't already! —>
(Link Here As Well)

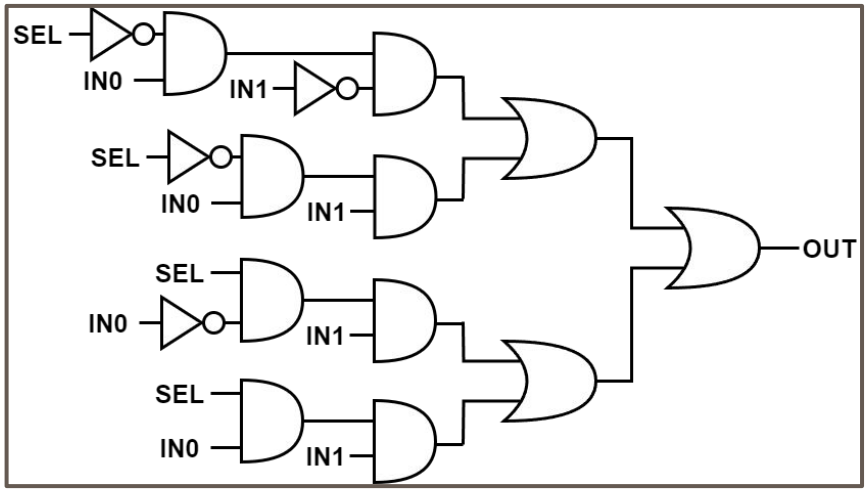
1. Discussion 3 Recap
2. Radix Conversion Practice Problems
3. Behavioral Verilog Introduction
4. Project 3A “Robbie” Discussion
5. Project 3B “Renee” Discussion
6. (Bonus) Additional Behavioral Verilog Stuff and Practice
7. (Bonus) Radix Conversion from ANY Base (B_1) → ANY Base (B_2)

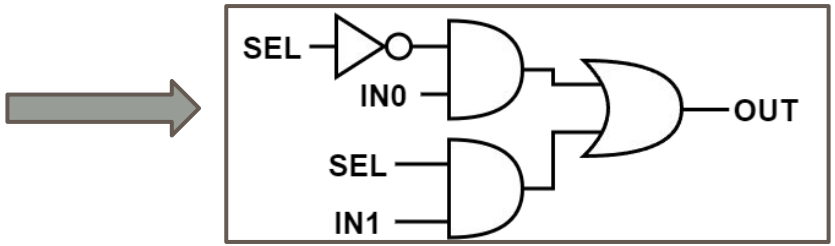
Discussion 3 Recap

Why Should We Care About Boolean Algebra

- Extremely helpful in finding how we can optimize our logic / circuit!

- **Building and optimizing circuits by doing math**





	A	Name	B
T1	$x \cdot 1 = x$	Identities	$x + 0 = x$
T2	$x \cdot 0 = 0$	Null Elements	$x + 1 = 1$
T3	$x \cdot x = x$	Idempotency	$x + x = x$
T4		Involution $(x')' = x$	
T5	$x \cdot x' = 0$	Complements	$x + x' = 1$
T6	$x \cdot y = y \cdot x$	Commutativity	$x + y = y + x$
T7	$x \cdot (x + y) = x$	Absorption	$x + (x \cdot y) = x$
T8	$x \cdot (x' + y) = x \cdot y$	No Name	$x + (x' \cdot y) = x + y$
T9	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	Associativity	$(x + y) + z = x + (y + z)$
T10	$x \cdot (y + z) = x \cdot y + x \cdot z$	Distributivity	$x + (y \cdot z) = (x + y) \cdot (x + z)$
T11	$x \cdot y + x' \cdot z + y \cdot z = x \cdot y + x' \cdot z$	Consensus	$(x + y) \cdot (x' + z) \cdot (y + z) = (x + y) \cdot (x' + z)$
T12	$f(x_1, \dots, x_n, 0, 1, \cdot, +)' = f(x_1', \dots, x_n', 1, 0, +, \cdot)$	De Morgan's	

Radix Conversion Practice Problems

Converting from ANY Base (B) → Decimal

Unsigned N-digit value in base B = $x_{(N-1)}x_{(N-2)}..x_1x_0$

→ Decimal value = $(B^{(N-1)} * x_{(N-1)}) + (B^{(N-2)} * x_{(N-2)}) + ... + (B^{(1)} * x_{(1)}) + (B^{(0)} * x_{(0)})$

Ex. $2E3D_{16} = [16^3 * (2)] + [16^2 * (E == 14)] + [16^1 * (3)] + [16^0 * (D == 13)]$

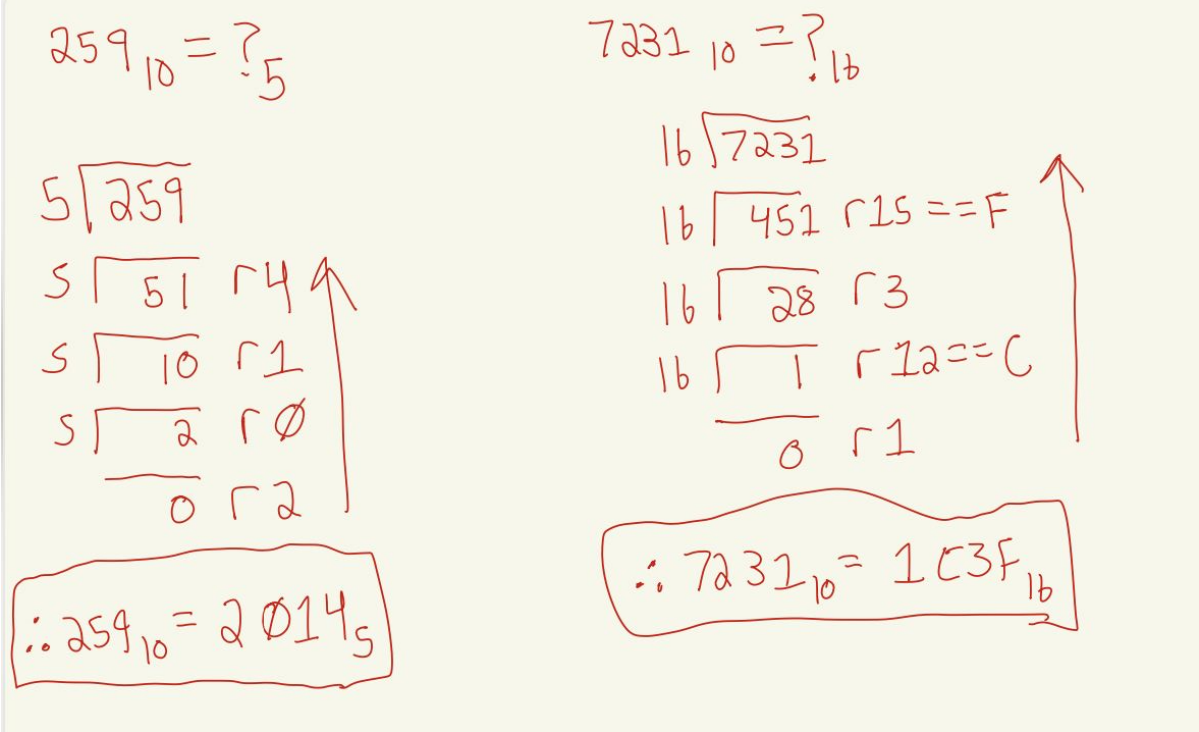
$= (4096 * 2) + (256 * 14) + (16 * 3) + (1 * 13)$

$= (8192) + (3584) + (48) + (13)$

$= \mathbf{11{,}837_{10}}$

Converting from Decimal → ANY Base (B)

- Same steps as binary, but with a different divisor



Practice!

1) 0 0 1 0 _____

2) 1 1 1 0 _____

3) 1 0 1 1 _____

4) 0 0 1 0 0 0 0 1 _____

5) 0 0 0 1 1 0 1 0 _____

6) 1 0 1 0 1 0 1 0 _____

Practice!

1) 8

2) 17

3) 11

4) 36

5) 70

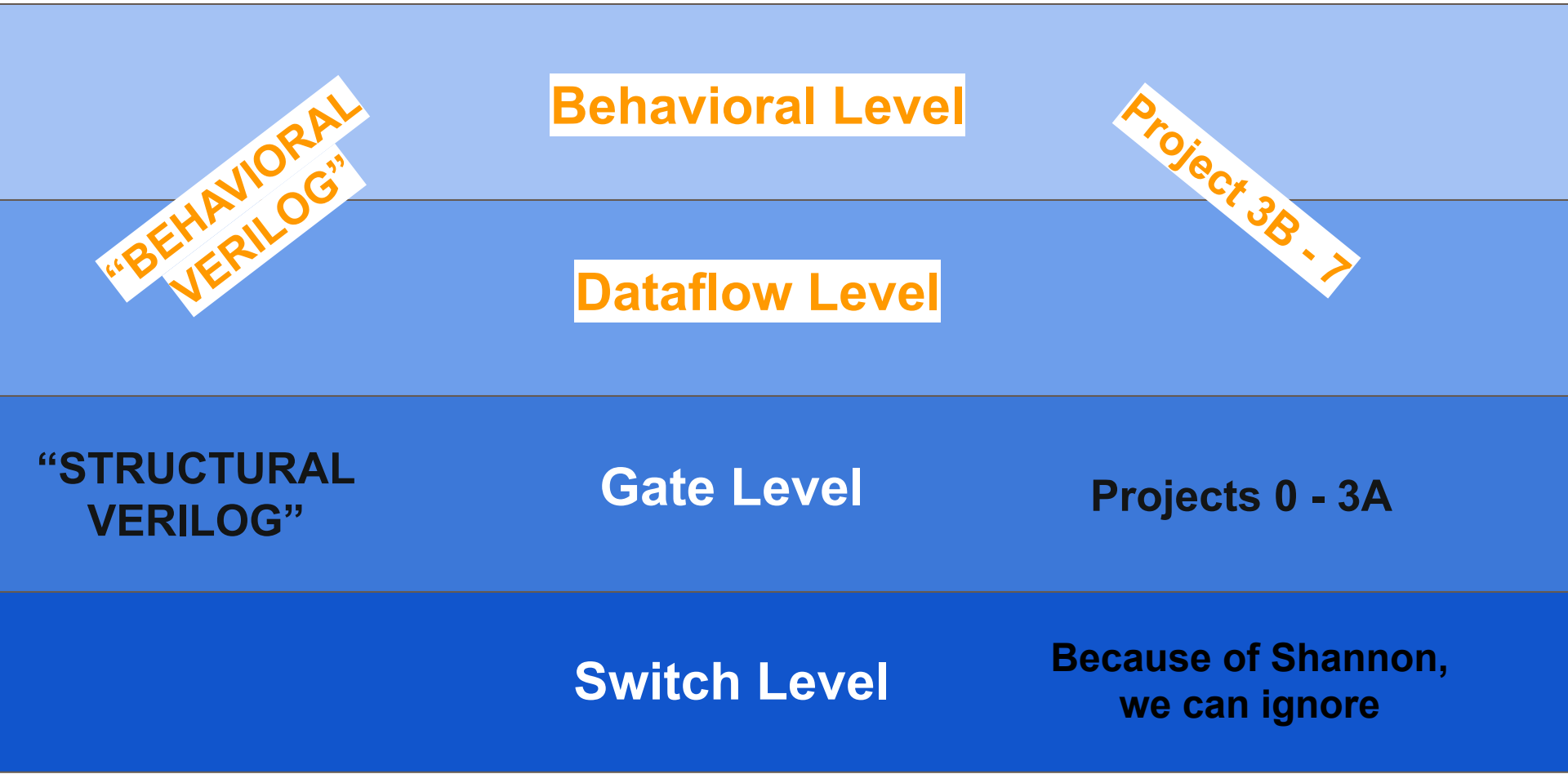
6) 132

Practice Problems!

1. Convert $123D_{16}$ to base 10
2. Convert $1000\_1001\_1010_{2}$ to base 10
3. Convert $B23D_{16}$ to Binary
4. Convert BDF_{16} to Octal
5. Convert 381_{10} to base 5
6. Convert 3210_{5} to base 3

Behavioral Verilog Introduction

Verilog's Many Levels of Abstraction



Behavioral Verilog Overview

- Gate-Level “Structural Verilog” Modelling
  - We can instantiate every primitive gate individually and connect them together
  - Helpful in building our knowledge of digital logic design
  - PROBLEM: Can get very difficult and tedious will more complex logic designs

- **Dataflow and Behavioral Level “Behavioral Verilog” Modelling**
  - Higher level abstraction of Verilog
  - **Allows us to focus more on the algorithmic approach than the individual gates**

- Logical Synthesis compiles our design to low-level Verilog to be understood by the board

Declaring and Assigning Wires (Behavioral)

```
// Declaring 1-bit wires
wire out_and, out_or_3, one_bit_mux;

// Continious assignment to wire 'out_and'
// Performing BIT-WISE and of i1 and i2
// NOTE: Left side MUST be a wire (or some net)
assign out_and = i1 & i2;

// Able to perform multiple operations in a single assign
assign out_or_3 = i1 | i2 | i3;

// Use of parenthesis to explicitly define ordering
assign one_bit_mux = (sel & i1) | (~sel & i2);
```

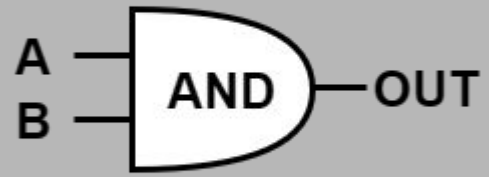
PLEASE USE PARENTHESES

- **To ensure the exact ordering of operations to occur, use parenthesis!!**

```
// The ordering is very hard to determine
// Verilog has an internal ordering
// DO NOT ASSUME
assign out = i1 ^ i2 & i3 | i4 & i5 ~^ i6;

// Explicitly define order with parenthesis :)
assign safe_out = ((i1 ^ i2) & i3) | (i4 & (i5 ~^ i6));
```

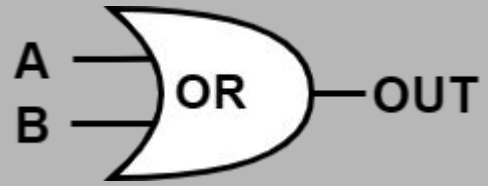
The Bitwise Operators - AND, OR, NOT

AND



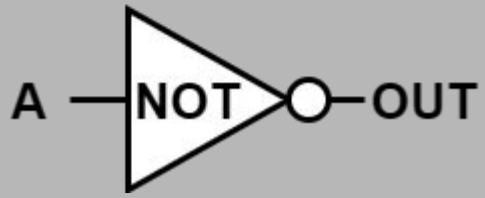
A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

OR



A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

NOT



A	OUT
0	1
1	0

The Bitwise Operators - NAND, NOR, XOR

- Will cover and practice more in-depth in later discussions

NAND

A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0

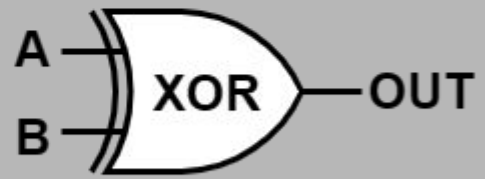
Inverted AND

NOR

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0

Inverted OR

XOR



A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	0

Inequality Operator

The Bitwise Operators - XNOR, BUF

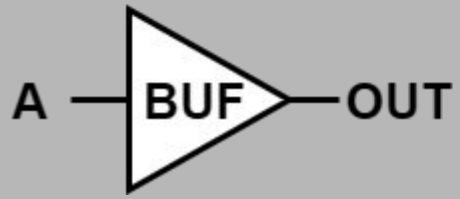
- Will cover and practice more in-depth in later discussions

XNOR

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	1

Equality Operator

BUF



A	OUT
0	0
1	1

Pass-Through Operator

Operator	Boolean Algebra	Verilog
AND	● (Looks like multiplication)	& (Bitwise), && (Logical)
OR	+ (Looks like addition)	| (Bitwise), || (Logical)
NOT	‘ , Ā (Either representation works)	~ (Bitwise), ! (Logical)
NAND	(A ● B)’	~(A & B)
NOR	(A + B)’	~(A | B)
XOR	A □ B	^
XNOR	(A ⊙ B), (A □ B)’	~^, ^~, ~(A ^ B) (Any representation works)
BUF	No operator	(Direct assignment)

Example Moving from Structural → Behavioral Verilog

Capture, Create, Implement Process (Recap)

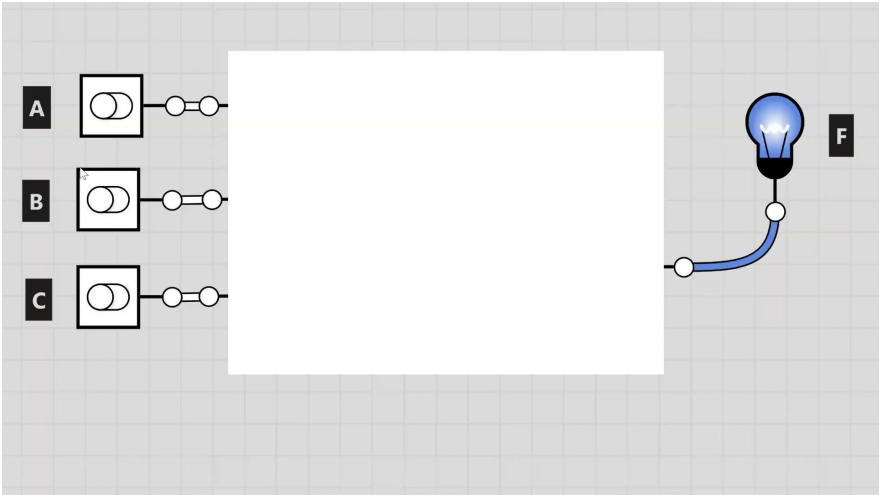
1. **Capture** the behavior of the function
   a. Either through a truth table or equation (Whichever is easier)

2. **Create** equations (If you haven't already)
   a. Canonical Sums-of-Products

3. **Implement** a gate-based circuit for <u>each</u> output
   a. Can then translate that gate directly into Verilog

Minority Function Implementation

- **Minority Function**
  - Inputs: 3 1-bit inputs (A, B, C)
  - Outputs: 1-bit output (F)
  - **F will go high if there are one or fewer logical high across the 3 inputs**



Capture the Function Behavior (Truth Table)

- **Minority Function**
  - Inputs: 3 1-bit inputs (A, B, C)
  - Outputs: 1-bit output (F)
  - **F will go high if there are one or fewer logical high across the 3 inputs**

A	B	C	F

Capture the Function Behavior (Truth Table)

- **Minority Function**
  - Inputs: 3 1-bit inputs (A, B, C)
  - Outputs: 1-bit output (F)
  - **F will go high if there are one or fewer logical high across the 3 inputs**

A	B	C	F

Capture the Function Behavior (Truth Table)

- **Minority Function**
  - Inputs: 3 1-bit inputs (A, B, C)
  - Outputs: 1-bit output (F)
  - **F will go high if there are one or fewer logical high across the 3 inputs**

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

Create Your Equation(s)

The 3-step process produces a Canonical SOP!!

1. Determine which rows resulted in ‘F’ being True (1)

2. Generate an minterm for Each Combination
   a. Only that combination should result in output being true

For combination (0, 0, 0) → A’ ● B’ ● C’

The only time this expression is True is when (0, 0, 0) is passed

Fot combination (0, 0, 1) → A’ ● B’ ● C

The only time this expression is True is when (0, 0, 1) is passed

3. OR each of the minterms

F = ?

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

Create Your Equation(s)

The 3-step process produces a Canonical SOP!!

1. Determine which rows resulted in ‘F’ being True (1)

2. Generate an minterm for Each Combination
   a. Only that combination should result in output being true

For combination (0, 0, 0) → A’ ● B’ ● C’

The only time this expression is True is when (0, 0, 0) is passed

Fot combination (0, 0, 1) → A’ ● B’ ● C

The only time this expression is True is when (0, 0, 1) is passed

3. OR each of the minterms

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

F = (A’ ● B’ ● C’) + (A’ ● B’ ● C) + (A’ ● B ● C’) + (A ● B’ ● C’)

Implement Your Gate-Based Circuit

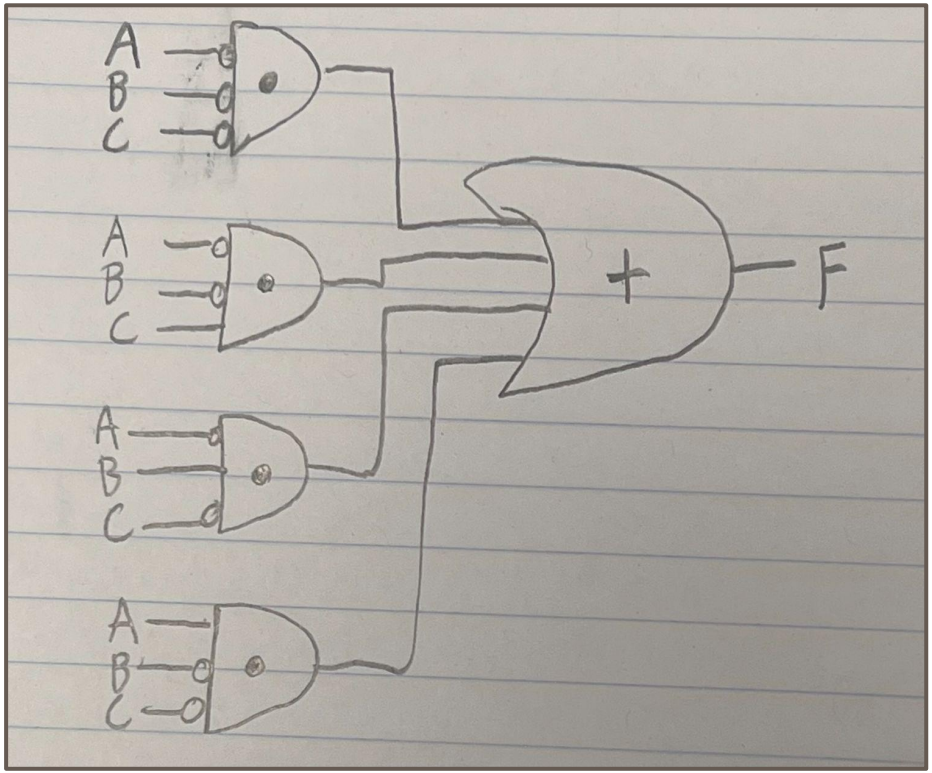
F =

(A’ ● B’ ● C’) +

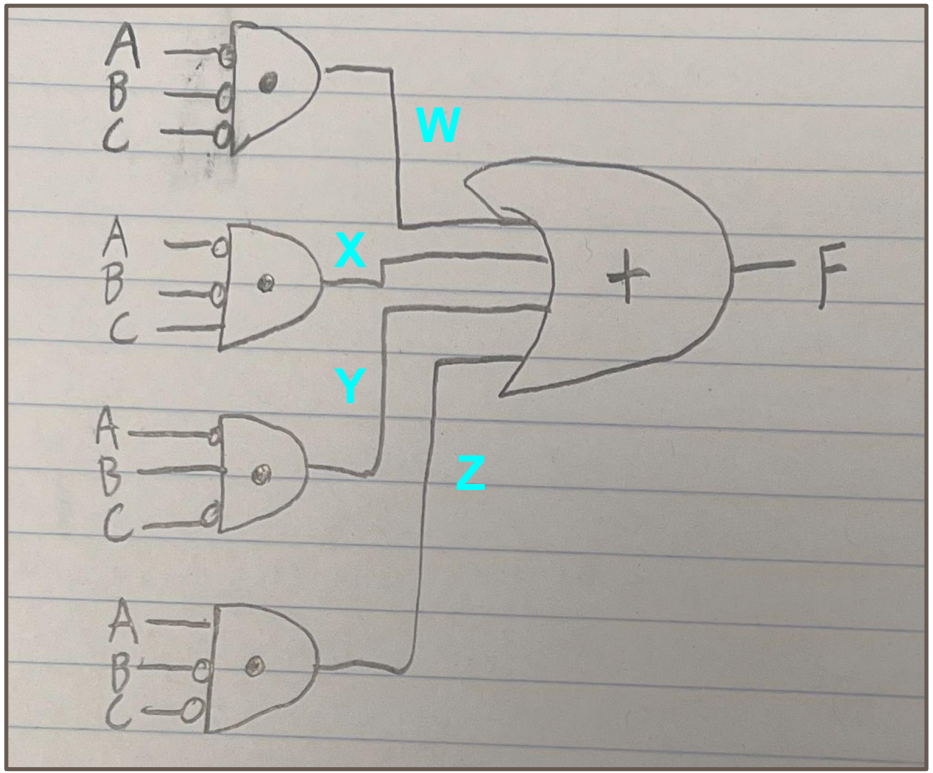
(A’ ● B’ ● C) +

(A’ ● B ● C’) +

(A ● B’ ● C’)



Implementation in **STRUCTURAL** Verilog




```
module Minority(
  input a, b, c,
  output out
);

  // Step 1. Declare your wires
  wire na, nb, nc;
  wire w, x, y, z;

  // Step 2. Drive some not a, b, c signals
  not n1(na, a);  // a'
  not n2(nb, b);  // b'
  not n3(nc, c);  // c'

  // Step 3. Drive w, x, y, z
  and a1(w, na, nb, nc);  // a' & b' & c'
  and a2(x, na, nb, c);   // a' & b' & c
  and a3(y, na, b, nc);   // a' & b & c'
  and a4(z, a, nb, nc);   // a & b' & c'

  // Step 4. Drive our output
  or o1(out, w, x, y, z);

endmodule
```

Implementation in **BEHAVIORAL** Verilog

Can just use your boolean equation

F = (A’ • B’ • C’) +
(A’ • B’ • C) +
(A’ • B • C’) +
(A • B’ • C’)

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & ~b & ~c) |
               (~a & ~b & c) |
               (~a & b & ~c) |
               (a & ~b & ~c);

endmodule
```

Simplifying Boolean Equation (w/ Behavioral Verilog)

OLD

F = (A' • B' • C') +
(A' • B' • C) +
(A' • B • C') +
(A • B' • C')

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & ~b & ~c) |
               (~a & ~b & c) |
               (~a & b & ~c) |
               (a & ~b & ~c);

endmodule
```

NEW

F = ?

Simplifying Boolean Equation (w/ Behavioral Verilog)

OLD

F = (A' • B' • C') +
(A' • B' • C) +
(A' • B • C') +
(A • B' • C')

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & ~b & ~c) |
               (~a & ~b & c) |
               (~a & b & ~c) |
               (a & ~b & ~c);

endmodule
```

NEW (Distribution)

F = [(A' • B') • (C' + C)] +
(A' • B • C') +
(A • B' • C')

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & ~b & (~c | c)) |
               (~a & b & ~c) |
               (a & ~b & ~c);

endmodule
```

Simplifying Boolean Equation (w/ Behavioral Verilog)

OLD

F = [(A' • B') • (C' + C)] +
(A' • B • C') +
(A • B' • C')

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & ~b & (1)) |
               (~a & b & ~c) |
               (a & ~b & ~c);

endmodule
```

NEW (Complements & Null)

F = (A' • B') +
(A' • B • C') +
(A • B' • C')

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & ~b) |
               (~a & b & ~c) |
               (a & ~b & ~c);

endmodule
```

Simplifying Boolean Equation (w/ Behavioral Verilog)

OLD

F = (A' • B') +
(A' • B • C') +
(A • B' • C')

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & ~b & (1)) |
               (~a & b & ~c) |
               (a & ~b & ~c);

endmodule
```

NEW (Distribution)

F = [(A' • (B' + (B • C'))] +
(A • B' • C')

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & (~b | (b & ~c))) |
               (a & ~b & ~c);

endmodule
```

Simplifying Boolean Equation (w/ Behavioral Verilog)

OLD

$F = [(A' \bullet (B' + (B \bullet C'))] + (A \bullet B' \bullet C')$

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & (~b | (b & ~c))) |
               (a & ~b & ~c);

endmodule
```

NEW ("No Name" & Distribution)

$F = (A' \bullet B') + (A' \bullet C') + (A \bullet B' \bullet C')$

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & ~b) |
               (~a & ~c) |
               (a & ~b & ~c);

endmodule
```

Simplifying Boolean Equation (w/ Behavioral Verilog)

OLD

F = (A' • B') +
(A' • C') +
(A • B' • C')

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & ~b) |
               (~a & ~c) |
               (a & ~b & ~c);

endmodule
```

NEW (Distribution)

F = (A' • B') +
[(C' • (A' + (A • B'))]

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & ~b) |
               (~c & (~a | (a & ~b)));

endmodule
```

Simplifying Boolean Equation (w/ Behavioral Verilog)

OLD

F = (A' • B') +
[(C' • (A' + (A • B'))]

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & ~b) |
               (~c & (~a | (a & ~b)));

endmodule
```

NEW (Distribution)

F = (A' • B') +
(A' • C') +
(B' • C')

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & ~b) |
               (~a & ~c) |
               (~b & ~c);

endmodule
```

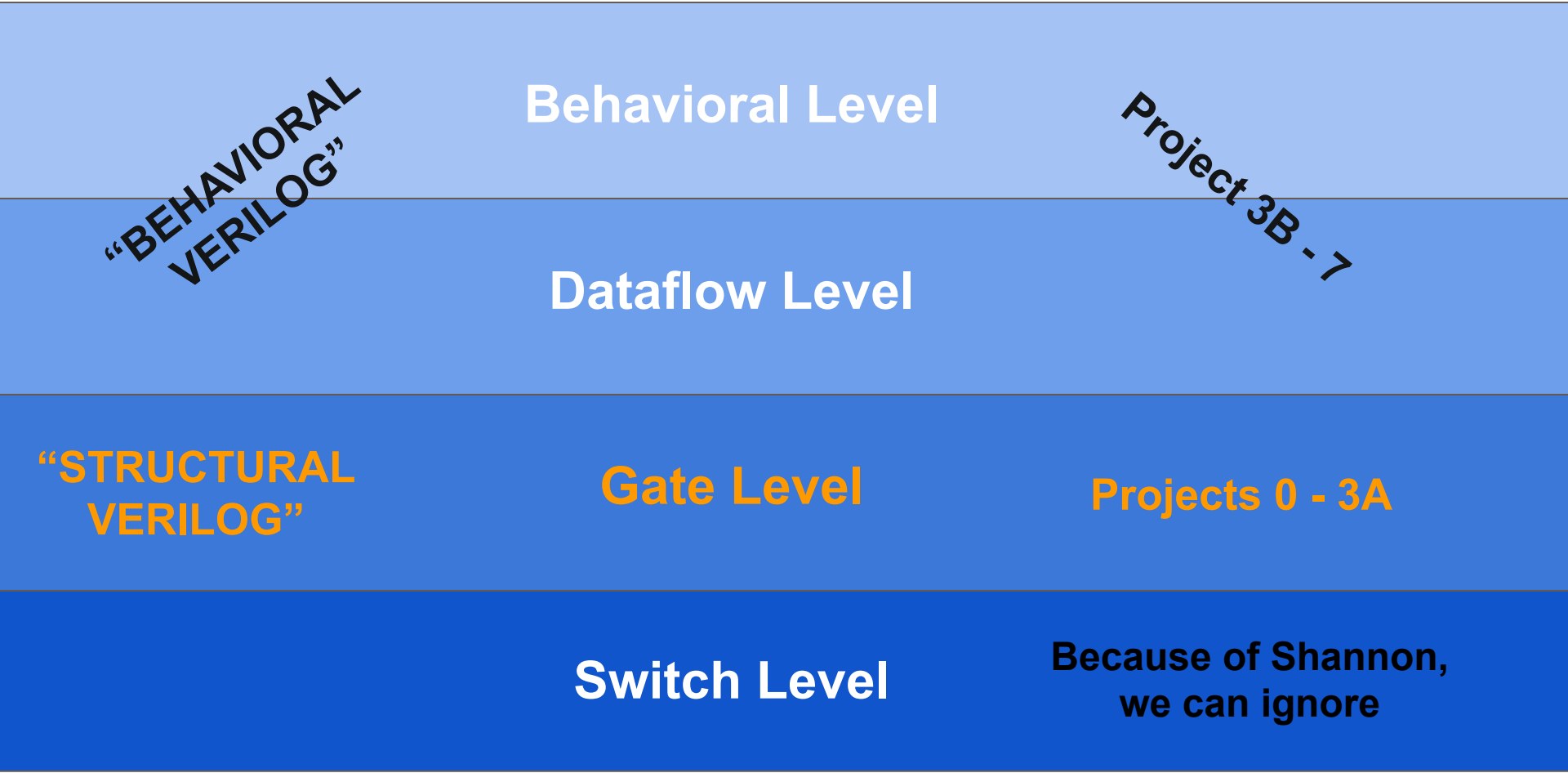
FINAL Implementation in **BEHAVIORAL** Verilog

F = (A' • B') + (A' • C') + (B' • C')

```
module Minority(
  input a, b, c,
  output out
);
  assign out = (~a & ~b) | (~a & ~c) | (~b & ~c);

endmodule
```

Project 3A “Robbie” Overview

Verilog's Many Levels of Abstraction



USE STRUCTURAL VERILOG FOR Project 3A!!!

- **You SHOULD NOT be using**
  - **Continuous ‘assign’ statements**
  - **Procedural statements ‘always’ or ‘initial’**

- These abstractural statements are used in “Behavioral” Verilog
  - Will be using in Projects 3B - 7

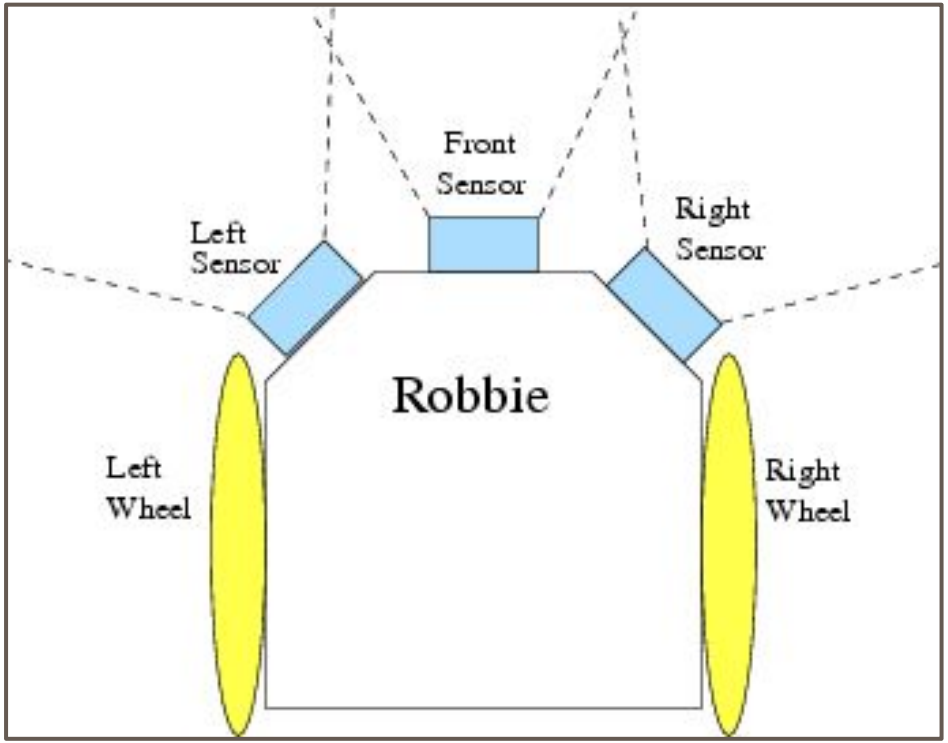
- **You SHOULD ONLY be using**
  - **Primitive gate instantiations as described in previous discussion**
  - **Module instantiations as described in previous discussion**

Exception - Allowing Verilog Parameters

- Way to define constant values in Verilog
- Will be helpful to use for RobbieAction module for displaying

parameter A = 7’b001010;

Robbie Interface



Inputs (Switches)

1. Left Sensor (ls)
2. Front Sensor (fs)
3. Right Sensor (rs)

Input Values

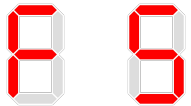
‘1’ - Beacon Detected

‘0’ - No Beacon Detected

Outputs (HEX Displays)

1. Left Wheel
2. Right Wheel

Output Values



Importance of Modularization

- **Goal with modules: Split Work, Simplify, and Abstractify**

- **Split Work**
  - Technically, everything can be done in one module
  - Very hard to test everything at once instead of one functionality at a time
- **Simplify**
  - Make each module as simple as possible (A lot easier to test)
  - If one module is too complex, split it down further to simpler modules
- **Abstractify**
  - Goal is to hide internal complexities of the modules
  - High-level modules should focus on what the module does, rather than how it does it

Project 3B “Renee” Overview

USE **BEHAVIORAL** VERILOG FOR Project 3B!!!

- **You are allowed to use**
  - Continuous ‘assign’ statements
  - Ternary operators / Concatenation

- **Behavioral Operations NOT allowed for this project**
  - Addition (+), subtraction/integer negation (-)
  - Greater than (>), less than (<), or equality (==)
  - If statements

- **My Advice -** Stay away from procedural blocks in non-testbench code
  - You don’t need ‘always’ or ‘initial’ statements for this project

Renee Interface - A LOT More Requirements

Inputs (Switches)

1. 3-bit Left Sensor (ls)
2. 3-bit Right Sensor (rs)
3. 4 Surrounding Bumpers (fb, rb, bb, lb)

Outputs (HEX Displays)

1. Left Wheel
2. Right Wheel

Renee should satisfy these requirements:

Renee should head in the direction of the sensor that indicates it has the strongest signal. If both signals are zero, Renee should stop. If the two signals are the same but not zero, Renee should go forward. However, the beacon senors are ignored if Renee has hit something: in that case her first priority is to move away from the thing she hit. If any of her bumpers detect a collision, she should follow these rules:

1. If her front bumper sensor is the only one detecting a collision, she should move in reverse.
2. If her back bumper sensor is the only one detecting a collision, she should move forward.
3. If her left bumper sensor is the only one detecting a collision, she should move right back.
4. If her right bumper sensor is the only one detecting a collision, she should move left back.
5. If her left and front bumper sensors are the only ones detecting a collision, she should move right back.
6. If her right and front bumper sensors are the only ones detecting a collision, she should move left back.
7. If her left and back bumper sensors are the only ones detecting a collision, she should move right.
8. If her right and back bumper sensors are the only ones detecting a collision, she should move left.
9. If any two bumper sensors on opposite sides are detecting a collision (front and back or left and right), she should stop.

Verilog Ternary Operator (Behavioral)

- Similar to a MUX operation (Built in project 1)
  - If-Else Statement
- Can assign multiple bits at a time!

```
// Conditional Ternary Operator
// If   sel = 1'b1 --> out = 4'b1001
// Else sel = 1'b0 --> out = 4'b0011
wire [3:0] out;
assign out = sel ? (4'b1001) : (4'b0011);
```

THANK YOU

Please fill out form if you haven’t already done so!
(Link Here As Well)

(Bonus) Additional Behavioral Verilog Stuff and Practice

Logical vs. Bitwise Operations (Behavioral)

Logical Operations

- Always evaluate to a 1-bit value
  - 0, 1, or X
- Logical And (&&), Or (||), Not (!)

```
// out1 = (true) || (true) = (true) = 1'b1
assign out1 = 4'b1001 || 4'b0011;

// out2 = (true) || (false) = (true) = 1'b1
assign out2 = 4'b1001 || 4'b0000;

// out3 = !(true) = (false) = 1'b0
assign out3 = !(4'b1001)

// out4 = (true) && (false) = (false) = 1'b0
assign out4 = 4'b1001 && 4'b0000;
```

Bitwise Operations

- Perform bit-by-bit operations
  - Will zero-extend the less-sized number
- Bitwise And (&), Or (|), Not (~)
- Bitwise Xor (^), Xnor (~^)

```
// out1 = (4'b1001) | (4'b0011) = 4'b1011
assign out1 = 4'b1001 | 4'b0011;

// out2 = (4'b1001) | (4'b0000) = 4'b1001
assign out2 = 4'b1001 | 4'b0000;

// out3 = ~(4'b1001) = 4'b0110
assign out3 = ~(4'b1001)

// out4 = (4'b1001) & (4'b0011) = 4'b0001
assign out4 = 4'b1001 & 4'b0011;
```

Reduction Operators (Behavioral)

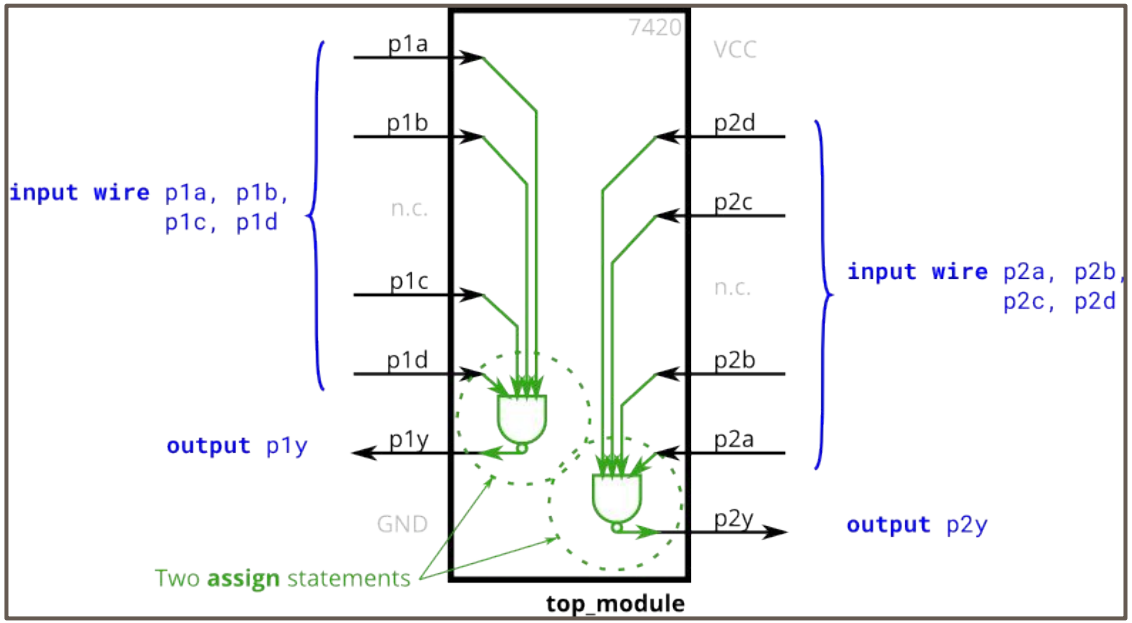
- Applying operation on every single bit in a bitvector

```
// Assume x = 4'b1010
assign out1 = &X //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
assign out2 = |X //Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
assign out3 = ^X //Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
```

Verilog Concatenation (Behavioral)

```
// Declaring 4-bit wires
wire [3:0] i1, i2;

assign i1 = 4'b1010;
assign i2 = 4'b0011;

// Declaring 8-bit wire
wire [7:0] o1;

// Using concatenation for assignment
// Appending the inputs into one bitvector
// o1 = {i2, i1} = {4'b0011, 4'b1010} = 8'b0011_1010
assign o1 = {i2, i1};
```

Practice Problem 1 - 7420 Dual 4-input NAND Gate




```
module chip_7420(
  input p1a, p1b, p1c, p1d,
  output p1y
  input p2a, p2b, p2c, p2d,
  output p2y
);

// Hint: Only need 2 assign statements

endmodule
```

Practice Problem 1 Code

```
module chip_7420(
  input p1a, p1b, p1c, p1d,
  output p1y,
  input p2a, p2b, p2c, p2d,
  output p2y
);

// Hint: Only need 2 assign
statements

endmodule
```

```
`timescale 1ns/1ns

module testbench;
  reg p1a, p1b, p1c, p1d;
  reg p2a, p2b, p2c, p2d;
  wire p1y, p2y;

  chip_7420 c(.p1a(p1a), .p1b(p1b), .p1c(p1c), .p1d(p1d),
              .p1y(p1y),
              .p2a(p2a), .p2b(p2b), .p2c(p2c), .p2d(p2d),
              .p2y(p2y));

  initial begin

    // Initial statements
    // Setting all the bits to 0
    // p1y = 1, p2y = 1
    {p1a, p1b, p1c, p1d} = 4'b0000;
    {p2a, p2b, p2c, p2d} = 4'b0000;
    #5;

    // p1y = ~(p1a & p1b & p1c & p1d)
    //     = ~(1 & 0 & 1 & 0) = ~(0) = 1
    {p1a, p1b, p1c, p1d} = 4'b1010;
    #5;

    // p2y = ~(p2a & p2b & p2c & p2d)
    //     = ~(1 & 1 & 1 & 1) = ~(1) = 0
    {p2a, p2b, p2c, p2d} = 4'b1111;

  end
endmodule
```

Practice Problem 2 - Population Counter

Specification: Build a "population count" circuit counts the number of '1's in an input vector. Build a population count circuit for a 3-bit input vector. The output is a 2-bit unsigned value of the number of people in the input vector.

in[2:0] = 3'b111 → out[1:0] = 2'b11

in[2:0] = 3'b101 → out[1:0] = 2'b10

in[2:0] = 3'b001 → out[1:0] = 2'b01

HINT: Make a truth table and equation for EACH BIT of out!

Practice Problem 2 Code

```
module population_count(
  input [2:0] in,
  output [1:0] out);

  // Have an equation for each bit of
out
  // Make a truth table (capture) for
each output bit
  // There should be 2 assign
statements, one for each bit

endmodule
```

```
`timescale 1ns/1ns

module testbench;
  reg [2:0] in;
  wire [1:0] out;

  population_count c(.in(in), .out(out));

  // Setting in[2], in[1], in[0] = 0 at simulation time = 0
  initial in = 3'b000;

  // always statements - always running
  // Will delay statement every 5 time units --> 5ns
  // Will increment in every 5 ns!
  // Want to test all combinations of in[2], in[1], in[0]
  // Run simulation for (8 combos) * (5 ns) = 40 ns
  always #5 in = in + 1;

endmodule
```

Practice Problem 3 - Truth Table → Verilog

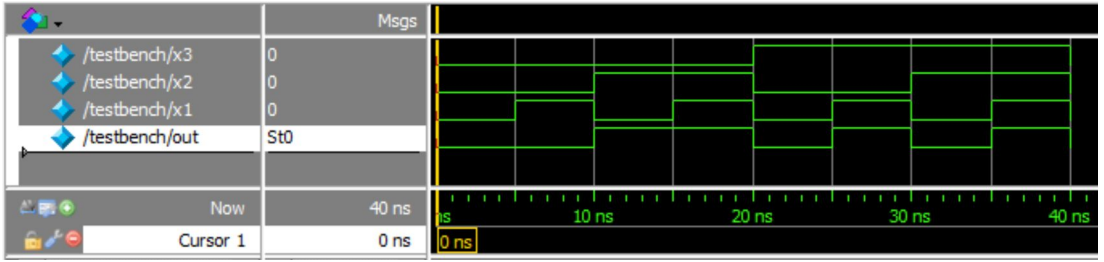
x3	x2	x1	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

```
module truth_table(
  input x3, x2, x1,
  output out);

  // Can be reduced to 4 primitive gates

  // BONUS: Try to make it into a ternary operation!

endmodule
```

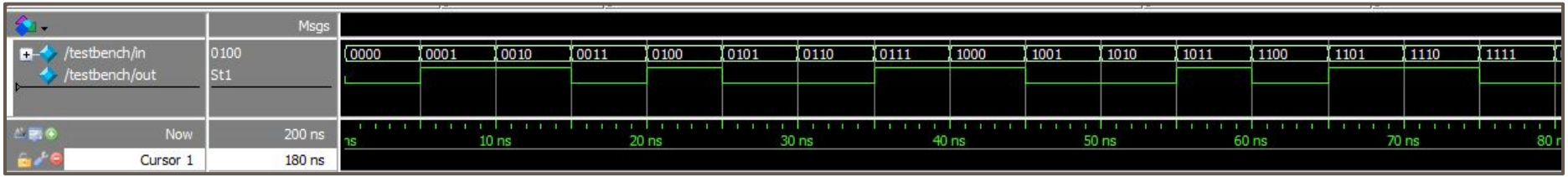
Practice Problem 3 Code

```
module truth_table(
  input x3, x2, x1,
  output out);

  // Can be reduced to 4 primitive gates

  // BONUS: Try to make it into a ternary operation!

endmodule
```

```
`timescale 1ns/1ns

module testbench;
  reg x3, x2, x1;
  wire out;

  truth_table c(.x3(x3), .x2(x2), .x1(x1),
                .out(out));

  // Setting x3, x2, x1 = 0 at simulation time = 0
  initial {x3, x2, x1} = 3'b000;

  // always statements - always running
  // Will delay statement every 5 time units --> 5ns
  // Will increment {x3, x2, x1} every 5 ns!
  // Want to test all combinations of x3, x2, x1
  // Run simulation for (8 combos) * (5 ns) = 40 ns
  always #5 {x3, x2, x1} = {x3, x2, x1} + 1;

endmodule
```



HARD Practice Problem 4 - Simulation Wave → Verilog



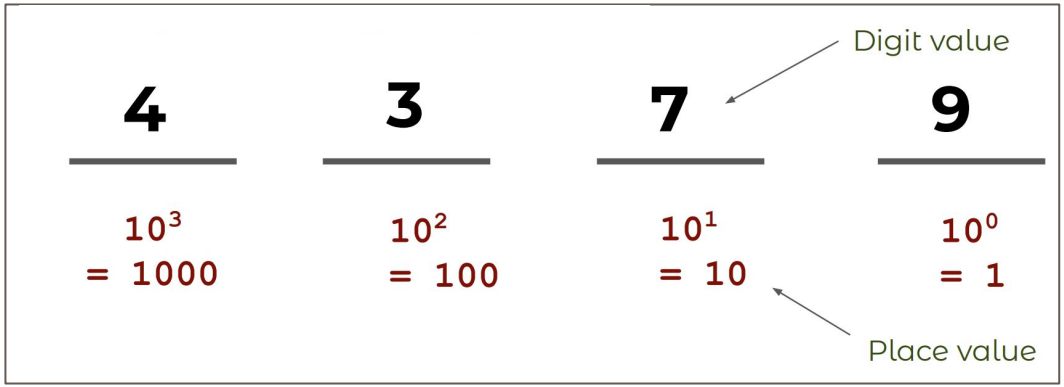

```
`timescale 1ns/1ns

module hard_problem(
  input [3:0] in,
  output out);

endmodule

module testbench;
  reg [3:0] in;
  wire out;

  hard_problem c(.in(in), .out(out));

  initial in = 4'b0000;

  always #5 in = in + 1;

endmodule
```

HINT: You can create the circuit with just 3 gates

(Bonus) Conversion from ANY Base (B_1) → ANY Base (B_2)

Decimal Notation System (Base **10**)

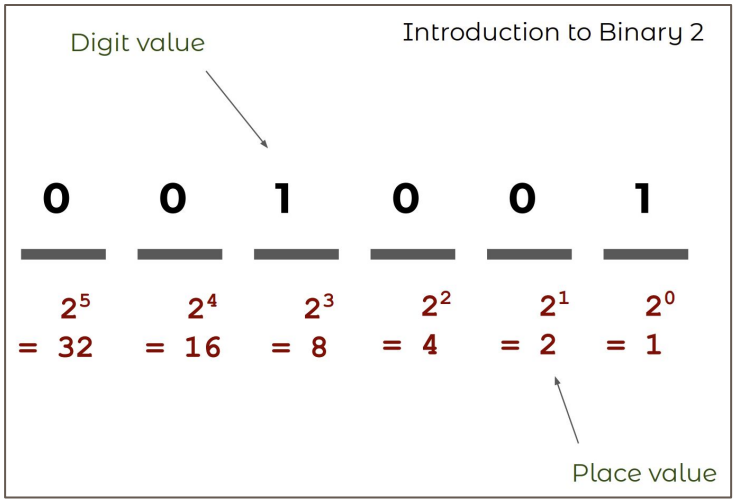
- Use **10** digits to represent all of the numbers
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Each digit is a power of **10**



$$
\begin{aligned}
&(10^3 * 4)\\
&+ (10^2 * 3)\\
&+ (10^1 * 7)\\
&+ (10^0 * 9)\\
&\text{----------}\\
&(1000 * 4)\\
&+ (100 * 3)\\
&+ (10 * 7)\\
&+ (1 * 9)\\
&\text{----------}\\
&\mathbf{4379}
\end{aligned}
$$

Binary Notation System (Base **2**)

- Use **2** digits to represent all of the numbers
  - 0, 1
- Each digit is a power of **2**



$$
\begin{aligned}
&(2^5 * 0)\\
&+ (2^4 * 0)\\
&+ (2^3 * 1)\\
&+ (2^2 * 0)\\
&+ (2^1 * 0)\\
&+ (2^0 * 1)\\
&\text{----------}\\
&(32 * 0)\\
&+ (16 * 0)\\
&+ (8 * 1)\\
&+ (4 * 0)\\
&+ (2 * 0)\\
&+ (1 * 1)\\
&\text{----------}\\
&\mathbf{9}
\end{aligned}
$$

Common Bases / Radices

Common Name	Base/Radix	Single Digit Range
Binary	2	{0, 1}
Decimal	10	{0, 1, 2, …, 8, 9}
Hexadecimal	16	{0, 1, …, 9, A, B, …, F}
Octal	8	{0, 1, …, 7}

Converting from Unsigned Binary → Decimal

Unsigned N-bit Binary value = $x_{(N-1)}x_{(N-2)}..x_1x_0$

→ Decimal value = $(2^{(N-1)} * x_{(N-1)}) + (2^{(N-2)} * x_{(N-2)}) + ... + (2^{(1)} * x_{(1)}) + (2^{(0)} * x_{(0)})$

Ex. $10101_2 = [2^4 * (1)] + [2^3 * (0)] + [2^2 * (1)] + [2^1 * (0)] + [2^0 * (1)]$

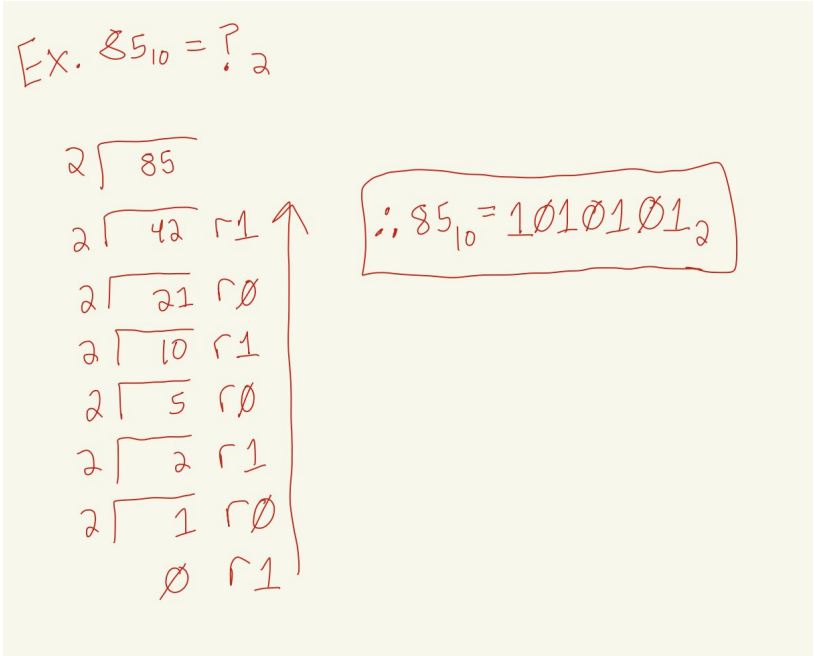
$= (2^4) + (2^2) + (2^0)$

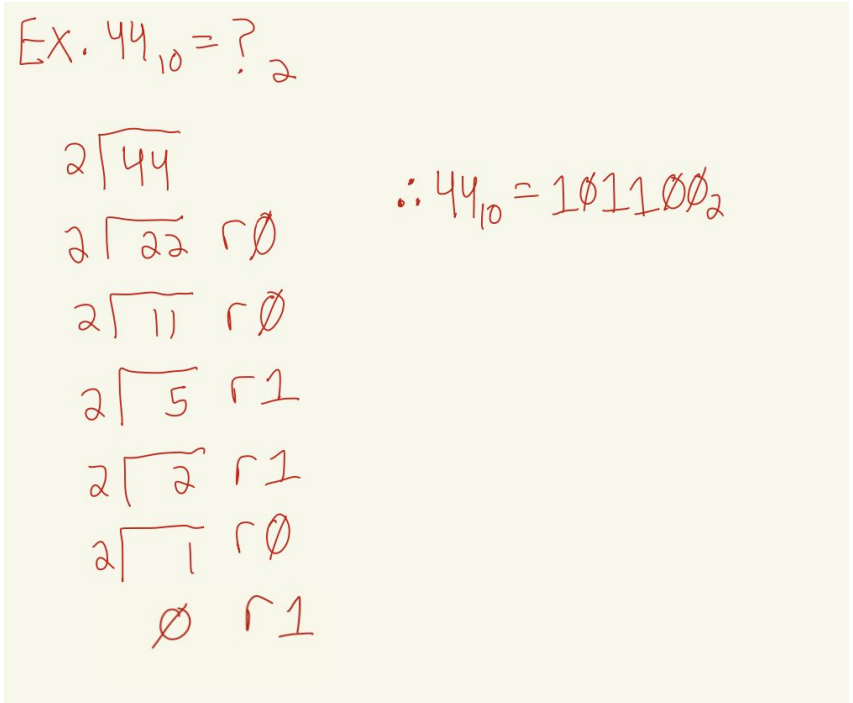
$= (16) + (4) + (1)$

$= \mathbf{21_{10}}$

Converting from Decimal → Binary

- Will require special long division





Practice!

1) 0 0 1 0 _____

2) 1 1 1 0 _____

3) 1 0 1 1 _____

4) 0 0 1 0 0 0 0 1 _____

5) 0 0 0 1 1 0 1 0 _____

6) 1 0 1 0 1 0 1 0 _____

Practice!

1)	0 0 1 0	2
2)	1 1 1 0	14
3)	1 0 1 1	11
4)	0 0 1 0 0 0 0 1	33
5)	0 0 0 1 1 0 1 0	26
6)	1 0 1 0 1 0 1 0	170

Practice!

1) 8

2) 17

3) 11

4) 36

5) 70

6) 132

Practice!

1) 8

2) 17

3) 11

4) 36

5) 70

6) 132

Converting from ANY Base (B) → Decimal

Unsigned N-digit value in base B = $x_{(N-1)}x_{(N-2)}..x_1x_0$

→ Decimal value = $(B^{(N-1)} * x_{(N-1)}) + (B^{(N-2)} * x_{(N-2)}) + ... + (B^{(1)} * x_{(1)}) + (B^{(0)} * x_{(0)})$

Ex. $2E3D_{16} = [16^3 * (2)] + [16^2 * (E == 14)] + [16^1 * (3)] + [16^0 * (D == 13)]$

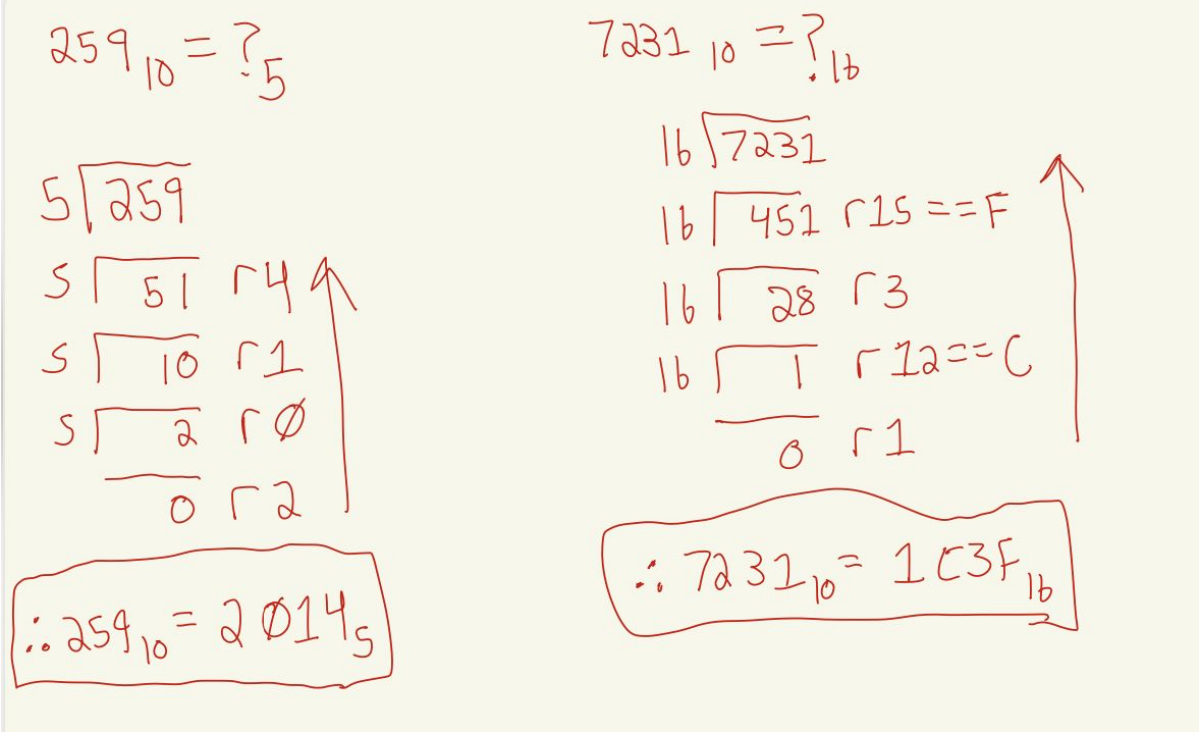
$= (4096 * 2) + (256 * 14) + (16 * 3) + (1 * 13)$

$= (8192) + (3584) + (48) + (13)$

$= \mathbf{11,837_{10}}$

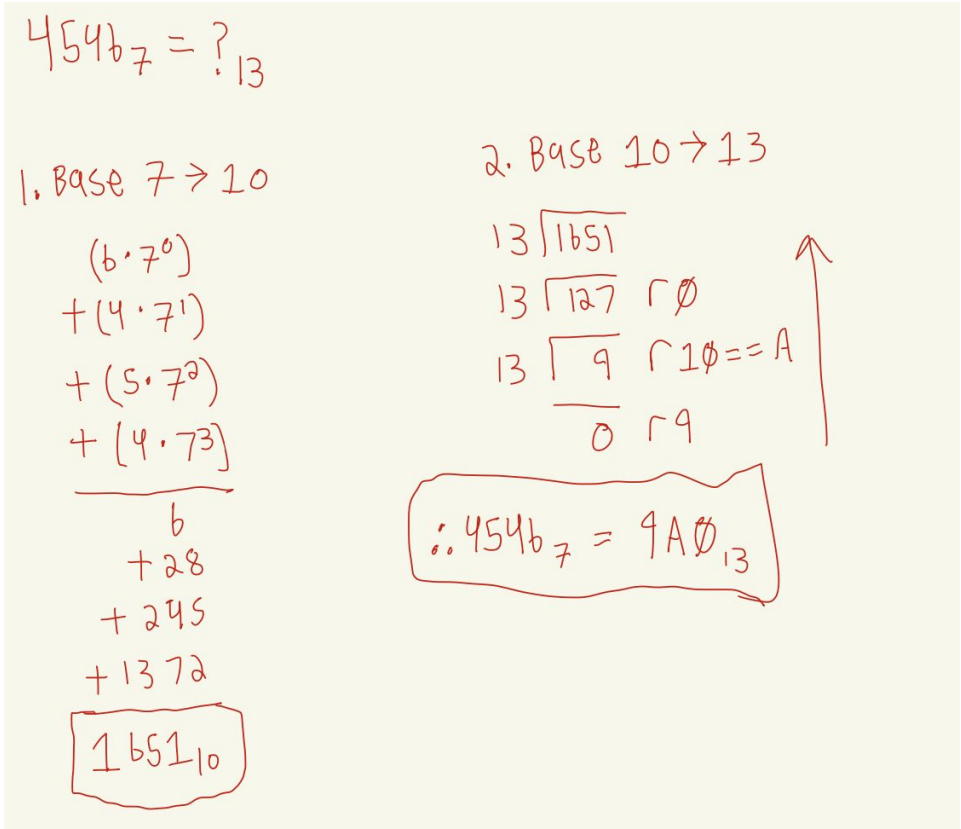
Converting from Decimal → ANY Base (B)

- Same steps as binary, but with a different divisor



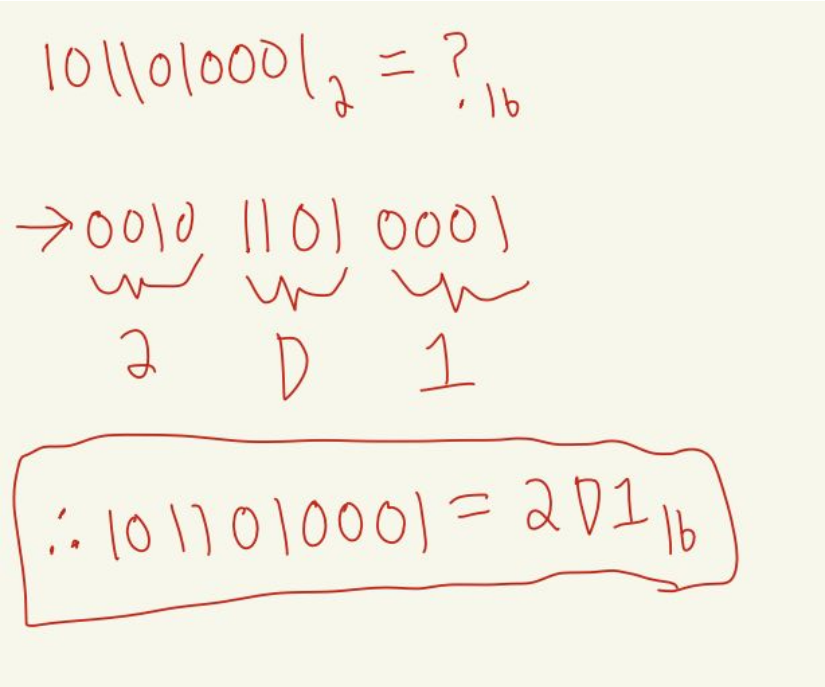
Steps to Convert from ANY Base (B_1) → ANY Base (B_2)

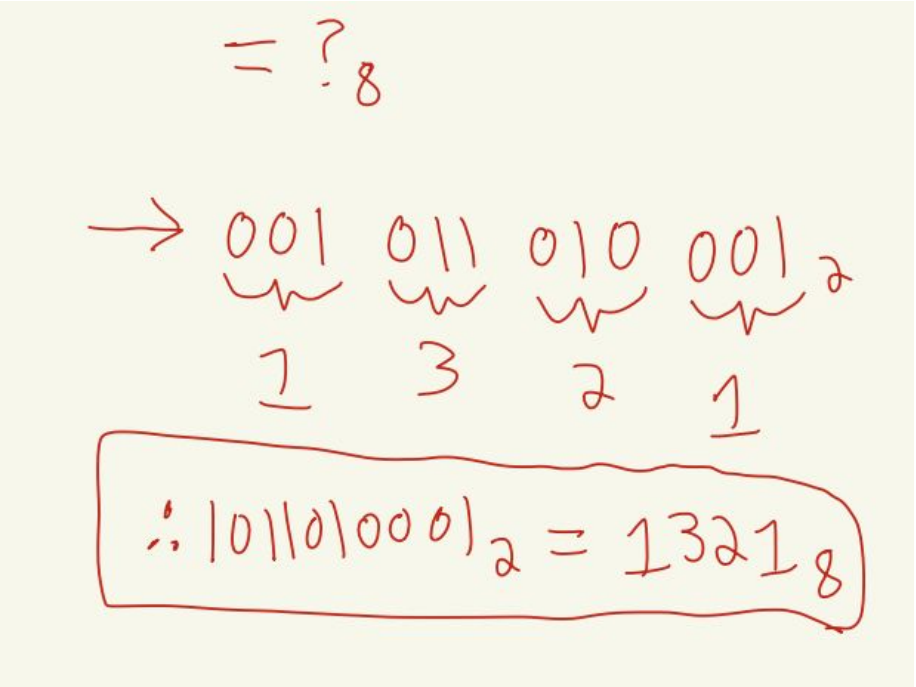
1. Convert B_1 → Decimal
2. Convert from Decimal → B_2



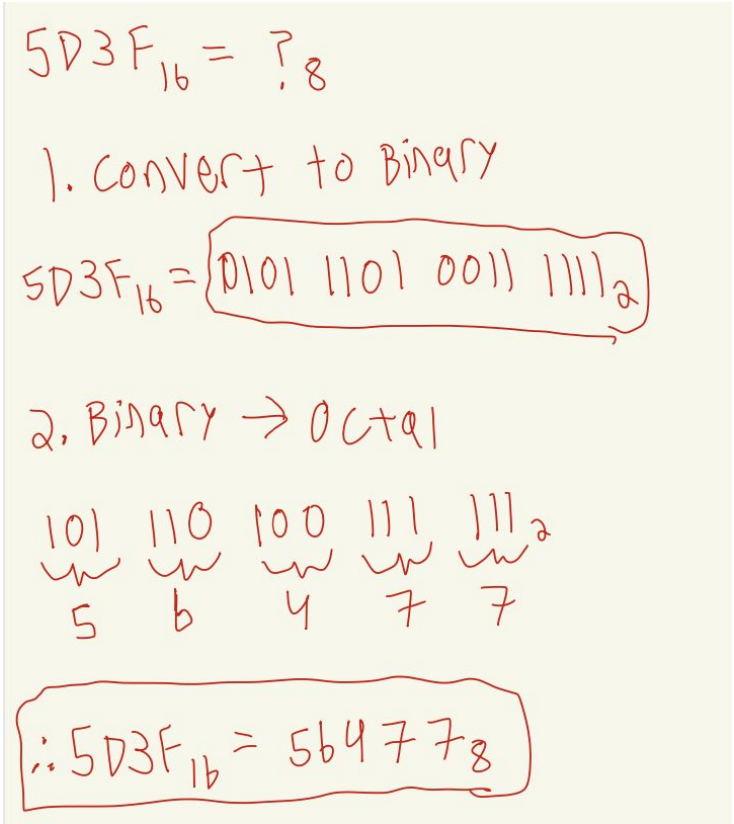
Trick with Octal ↔ Binary ↔ Hexadecimal Conversions

- Very easy to convert between these bases





Trick with Octal ↔ Binary ↔ Hexadecimal Conversions



Practice Problems!

1. Convert $123D_{16}$ to base 10
2. Convert $1000\_1001\_1010_{2}$ to base 10
3. Convert $B23D_{16}$ to Binary
4. Convert BDF_{16} to Octal
5. Convert 381_{10} to base 5
6. Convert 3210_{5} to base 3

Practice Problems!

1. Convert $123D_{16}$ to base 10 (4669_{10})
2. Convert $1000\_1001\_1010_2$ to base 10 (2202_{10})
3. Convert $B23D_{16}$ to Binary ($1011\_0010\_0011\_1101_2$)
4. Convert BDF_{16} to Octal (5737_8)
5. Convert 381_{10} to base 5 (3011_5)
6. Convert 3210_5 to base 3 (120221_3)