**1A:** No calculator on test

Base 10 – Decimal (0-9)

* Base 10 to other bases
  + Also successive division (and multiplication on right side)
* 612 in base ten to binary
  + 512, 64, 32, 4 ,
  + 1001100100
    - Convert in our heads using the powers of two (make sure to pick largest numbers)
* N digits – 3 digits can count to 999 (1000 combos)
  + # of combinations = baseN
  + 2N > 1000 = 10 digits

Base 2- Binary (0,1)

* Successive division from decimal to binary
* Remainder on right side…( then go bottom to top to print out values)
* 1,2,4,8,16,32,64,128, 256, ( 512, 1024, 2048, 4096, 8192… )
* With decimal – right side we multiply it (then go top to bottom to print out values)
  + successive multiplication
* Subtraction
  + Need to use the borrow if top one is zero
  + 10010-010111=
    - 001111
* Multiplication
  + Simple because its zeros and ones
    - Multiply by zero or one
    - Not hard to move from addition to multiplication with circuits
* Division
  + Similar method
* The last bit is always the ones' place, and if a number is odd, it must have a 111 in that ones' place. There's no way to create an odd number in the binary system without that ones' place

Convert from any base to base 10

* X0 \* A0 + X1 \* A1 + ….
* 245 in base 6 is
  + 5\*1 + 4\*6 + 2\*62 = 101 (base 10)
* 370 (base 8) – becomes 3 \* 8^2 + 7 \* 8^1 + 0 \* 8^0
  + And then you just add them together
    - 8^0 is a one

Base 8 -Octal (0-7)

* Shortcut: groupings of 3 binary digits
* 11011101.1101101 = 335.664 (binary to octal)
* Start from decimal point and move left and right

Base 16- Hexadecimal (0-F)

* 0-9, A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
* F3.C
  + 15\*16 + 3 + 12/16
  + 243.75
* 16N ( or 2^4) > 1000 (power of 3)
* Usually to convert between bases, we go through base 10, but there’s a shortcut if we go through base2-base4… or other corresponding bases
  + B – would be 4 digits in binary (11- decimal) would be 1011
* 11011101.1101101 = DD.DA

**1B Unknown bases**

* Study quadratic formula
* Multiplication in different bases
  + Consider converting to base 10 and then convert to the other base
* 1Z0110 (base 2) = 36 (base b)
  + Z can be zero or a one
    - So we should split it up to two possible answers
* Largest number represented in base 5 with 3 digits
  + 5^3 = 125 combos – 0-124
    - 444 (in base 5)
* How many digits represent 520 in base 8
  + 4 digits – 8 to the power of 4
    - More than enough digits
* In what bases is the number 19 (base 10) represented using 2 digits
  + Base 20 or more can be represented in 1 digit
  + **Base 5 with an extra digit remaining**
  + From 5 to 19, we can use 2 digits

**2A Signed value representation**

* + Negative binary numbers
  + We can represent 2^n digits in binary
  + With 4 combos, we can do 15 different combs (0000… 0001…. 1111)
  + Negative numbers force us to sacrifice some of our real estate
    - We would dedicate the first bit to positive or negative (signed bit)
    - 0 = positive
    - 1 = negative
    - Rest of digits represent the magnitude (3 bits – 0-7 combos (7 to -7) with redundancy with +0 and -0)
      * Double representation for that number
      * Not 100% efficient here
  + 1\_0 \_0 \_0 \_
  + 7 bits to represent the number 32
  + 15 would be 001111
  + 0100000 (32)
  + 1001111 (-15)
  + Not easy to subtract this way

1s complement

* Solves this negative number subtraction issue
* Backward of binary number
* 1101 (-2) becomes 0010 (2)
* Amount of bits matter
* Subtraction:
  + If the first digit would be 1+1 = it becomes a zero and then we go back to the end and increment that number (if it was a zero) – can’t fit in the front
    - If it was a one, then we use the carry method

2s complement

* Same as 1s complement but solves problem of extra zero
* We do 1s complement and then add a +1 to the last bit (digit)
* We gain a tiny bit of real estate with this (7 to -8 can be represented)
* -6 = 0110 > 1001 > then add a +1 to last column
  + So it becomes 1010 in 2s compl
* Addition/subtraction is a bit simpler
  + In 2s complement we don’t carry about anything carried past the number (no carry back)
  + We ignore final carry (leave it as is)
  + 32-15 = 0010001 in 2s compl
* Binary – 2s complement
  + Same both ways with switching and then add 1 to the end
* We can only represent negative numbers - The smallest number we can represent is −2N−1

Overflow

* # exceeds boundary
  + Goes beyond numbers it can represent
  + Regular binary lets us see if we have overflow (straightforward)
  + 1101 + 1100 (13+12) but only have 4 possible digits (0-15 combos)– we know if we have to carry above the amount of columns we have
  + Final Carry = overflow in regular binary
* Adding negative and positive together doesn’t usually exceed boundaries / aka no overflow
* Two negatives or positives
  + If sign changes, it signifies an overflow

ACSII

* Way to represent characters and other interesting things using binary code
* 128 chars
* Capitol, lowercase, numbers, and some symbols

BCD (8,4,2,1)

* Represent numbers individually (each represents 4 bits)
  + But adds on extra bits as a side effect
  + Illegal combos (6 items)
    - 1010
    - 1011
    - 1100
    - 1101
    - 1110
    - 1111
* Whenever we get an illegal combo, we need to add 6 (0110) before we go on to next column
* Less efficient but more intuitive

**3A - ECC**

Gray Code

* Will use going forward – with k-maps
* Each successive number only changes by one bit (digit)
* Symmetry line – above we put zeros, below is ones
* Always will be powers of 2
* And reflect what we did
  + 00
  + 01
  + 11
  + 10
* XOR ⊕
  + Either or – only A is true or only B is true… else we will have zeros
  + Only one of the two variables has value of 1
* Convert from Binary to gray
  + 11010010
  + Start on left hand side
  + First digit is always the same (imagine adding a zero – and take Xor between first two digits)
    - Then we compare the next to (1 and 1)
  + Code becomes 10111011
* Convert from gray to binary
  + A little more confusing
  + First digit is always the same (imagine adding a zero – and take Xor between first two digits)
    - Then we compare from gray to the binary (conversion)
    - First is 1 so we compare the gray 1 to the binary 1
      * Then we move to the grey 0 and the binary one , and so on
  + So we compare between old and new number

Parity bit (error correction)

* How many ones do we have
* If we have an even number of ones
  + Parity bit is a zero
  + Odd number of ones, will have a parity bit of 1 at the end

Code distance

* Dictionary
  + All possible combinations 2^8 – 256
    - By not using all combos, we leave room for Error Correction
  + Dictionary defines which combos are legal
* Define code distance between possible words
* Minimum distance between two words in the dictionary where distance is how many bits are different between the two words (or more)
  + 100 and 111 have distance of 2
  + And then we are interested (usually) with the **minimum distance** between two or more words
    - Two words that care close to each other
  + Sometimes we can add another point in without adversely effecting the code distance
    - (001,100,111,010) – all distance of two
  + Distance = **2C + D + 1**
    - D = # of errors detected
    - C = # of errors corrected
    - D + C = total # of errors can be detected
    - For example: 2(0) + 1 + 1
      * With parity bit, we can detect an error but not correct anything
    - 3 = (two possible answers)
      * 2(1) + 0 + 1
        + Assume no more than 1 error (distance of 3)
        + Can detect and correct **same** error (not including error that I’m correcting)
      * 2(0) + 2 + 1
        + Better for noisy channels
        + Assume 2 errors (maybe)

But less options for correcting them

Hamming code (4,7)

* 4 regular bits, adding on 3 parity bits (each relate to subset of bits)
* Based on parity bits, we know exactly where error is
* Goal is distance of 3
  + Correct an error
    - How many parity bits do we need to add to a string in order to get a distance of 3
* 2k (– 1) > m
  + N – Number of info bits
  + K – added parity bits
  + M = n + k = total number of bits
  + We would need 4 parity bits to get distance of 3
* Insert parity bit at every power of 2
  + #1,#2,#8,#16
    - Longer string is = more efficient hamming code (but higher probability of getting errors)
* Hamming (4,7)
  + Count from 1 and no (not zero)
* P1,P2, I1,P3, I2,I3,I4 – transmission string
  + P1 – (P1, I1, I2, I4)
  + P2 – (P2, I1,I3,I4)
  + P3 – (P3,,I2,I3,I4)
    - Each parity bit is determined by the Is in that sub-section
    - If number in each subsection is even # of ones, parity is going to be zero
  + Person gets 7-bit transmission on other end
  + W1, W2,W3,W4,W5,W6,W7
    - And calculates
    - A3 – (1,3,5,7)
    - A2 – (2,3,6,7)
    - A1 – (4,5,6,7)
      * Whatever comes out (each gives us a zero or 1)
        + We get a 3-digit number and then convert it using binary to get exact position of the error
    - Only if he gets 0,0,0, he will know there is no mistake
      * Otherwise, there’s an error and he converts the binary to decimal, then goes into that position in code and corrects the error (flips the bit)

Timeline

Description automatically generated

3B – Hamming, Gray, BCD

**4A** Input and output (0 or 1)

Logic gates

* Inverter/Not
  + From A’ to A (Not A)
  + or A to A’
* And
  + 2 (or more inputs) – one output
    - Only if both are true
  + A \* B = AB
* Or
  + 2 (or more inputs) – one output
    - At least one input needs to be on
  + A + B
* Nand
  + (A \* B)’
    - Reverse of AND
    - Only zero when both inputs are 1 (else it’s a 1)
* Nor
  + (A + B)’
    - Reverse of OR
    - Only equals to one when both inputs are 0
* X-Or
  + A ⊕ B = AB’ + A’B
  + Used in gray code conversions
    - Equal to one only when one input is equal to 1
* X-Nor
  + A ⊙ B = AB + A’B’
  + Opposite of either or.. only equal to 1 when either both are off or both are on
* Venn diagrams come in handy here

Truth tables

* Inputs to outputs in simple table

Boolean algebra

* Duality doesn’t switch the NOTs ‘ … rather just the signs from addition to multiplication and vis versa
* We use Axioms to modify and change these algorithms
  + #1 Logical values
    - If a is not zero, then it must be a 1 (and vis versa)
    - A = 0, if A != 1
  + #2 logical negation
    - Using the Not operator – taking complement (reverse)
    - A = 0, if A’ = 1
  + #3 logical product
    - Multiplication by zero gets you zero
    - A\*B = 1, if A=B=1
    - Otherwise: A\*B = 0
  + #4 logical sum
    - Addition
    - A+B=1 if A=1 or B=1
    - Otherwise
      * A+B=0
  + #5 Logical precedence
    - NOT precedes AND, AND precedes OR
    - F = (((A’) ∙ B) + C)
* Theorems
  + **De Morgan’s Law of duality**
    - A\*0=0 is equivalent to A+1=0
      * By changing signs and switching NOTs
    - algebraic equality will remain true if all 0’s and 1’s are interchanged and all AND and OR operations are interchanged
  + Identity
    - A+0 = A and dual would be A·1 = A
    - An identity operation is one that when performed on a variable will yield itself regardless of the variable’s value.
  + Null Element
    - A null element operation is one that, when performed on a constant value, will yield that same constant value regardless of the values of any variables within the same operation
    - A+1 = 1 and dual would be A\*0=0
  + Idempotent
    - An idempotent operation is one that has no effect on the input, regardless of the number of times the operation is applied.
    - A+A=A and dual would be A\*A=A
  + Complements
    - A+A’ = 1 and dual would be A\*A’ = 0
    - This theorem describes an operation of a variable with the variable’s own complement.
  + Involution
    - An involution operation describes the result of double negation.
    - (A’)’ = A
      * Or A’’ = A
  + Commutative Property
    - Changing the order of variables in an OR operation does not change the end result.
    - A+B = B+A and dual would be A·B = B·A
  + Associative Property
    - The term associative is used to describe an operation in which the grouping of the quantities or variables in the operation have no impact on the result.
    - (A+B)+C = A+(B+C) and dual would be (A·B)·C = A·(B·C)
  + Distributive Property
    - The term distributive describes how an operation on a parenthesized group of operations (or higher precedence operations) can be distributed through each term
    - A·(B+C) = A·B + A·C dual would be A+(B·C) = (A+B)·(A+C)
  + **Absorption**
    - The term absorption refers to when multiple logic terms within an expression produce the same results. This allows one of the terms to be eliminated from the expression, thus reducing the number of logic operations. The remaining terms essentially absorb the functionality of the eliminated term. This theorem is also called covering because the remaining term essentially covers the functionality of both itself and the eliminated term.
    - A+A·B = A would be A·(A+B) = A
  + Uniting
    - The uniting theorem, also called combining or minimization, provides a way to remove variables from an expression when they have no impact on the outcome.
    - A·B + A·B’ = A would be (A+B)·(A+B’) = A
  + **De Morgan’s Theorem**
    - A’ + B’ = (A·B)’ would be A’ · B’ = (A + B)’
    - Inverse of each component and addition/multiplication will be switched

**Table

Description automatically generated with medium confidence**

SOP vs POS

* Each row has it’s own min/max terms
* Sum of products
  + Minterms using ∑
  + Using multiplication (and)
  + Take 1s out of truth table
  + F = B\*C + A\*B + A\*C’
* Product of Sums
  + Maxterms using π Π
  + Using addition (or)
  + Take the 0s from the truth table
  + F= (B+C’)\*(A’+B)
* Not most optimized (canonical)

Same truth table = identical as far as we are concerned

Complete sets

* And , Inverter, OR gate
  + Can derive any function from that
* OR , AND gates
* OR gate and OR gate
* AND, OR and NOT (a Full Set)
* AND and NOT (a Complete Set)
* OR and NOT (a Complete Set)
* NAND (a Minimal Set)
* NOR (a Minimal Set)
* A set of Boolean operators is said to be functionally complete when the set can implement all possible logic functions. The set of operators {AND, OR, NOT} is functionally complete because every other operation can be implemented using these three operators (i.e., NAND, NOR, BUF, XOR, XNOR). The De Morgan’s Theorem showed us that all AND and OR operations can be replaced with NAND and NOR operators. This means that NAND and NOR operations could be by themselves functionally complete if they could perform a NOT operation. A NAND gate can be configured to perform a NOT operation. This configuration allows a NAND gate to be considered functionally complete because all other operations can be implemented.
* This approach can also be used on a NOR gate to implement an inverter. A
* NOR gate can be configured to perform a NOT operation, thus also making it functionally complete.

Need to brush up on converting between unknown/ different bases, complete sets, duality,

**5A**

Karnaugh maps (not necessary for test but helpful)

* Use gray code to skip lines when necessary
* EPI – We cant make a function without it
* PI prime indicators – any combos – EPIS are also PIs
* We use the horizontal method
* We want a variable that doesn’t change, and then write B – if it’s zero both times, then it’s B’ but 1 both times would be B
  + If it changes it’s independent and we don’t need the function