Haricharan

Digital Systems

2nd May

What are the advantages of digital systems?

- Resilience to noise → Noise forms a Gaussian distribution around the actual value. Filtering analog signals is hard, especially with noise.
- Ease of storage → Flash/CD/DVD
    - Previously stored in analog form in magnetic tapes
- More secure (encrypted)
- Flexible signal processing (e.g. noise removal)

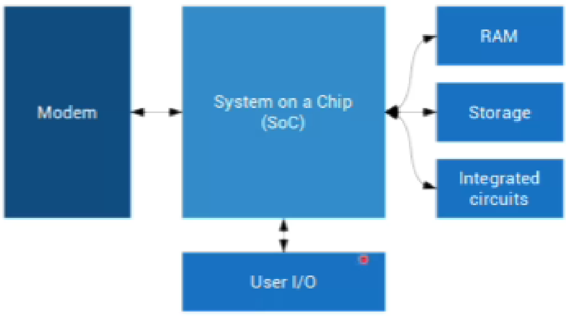
What are some examples and applications of digital systems?

- Digital thermometers
- automated driving
- digital computer stuff
- augmented reality, etc.
- The AI

What is the anatomy of a digital system?

- Top down methodology: we talk about big things and break it into smaller pieces
- Implement
    - Sequential logic
    - Combinational logic

Blocks in a Mobile Phone!

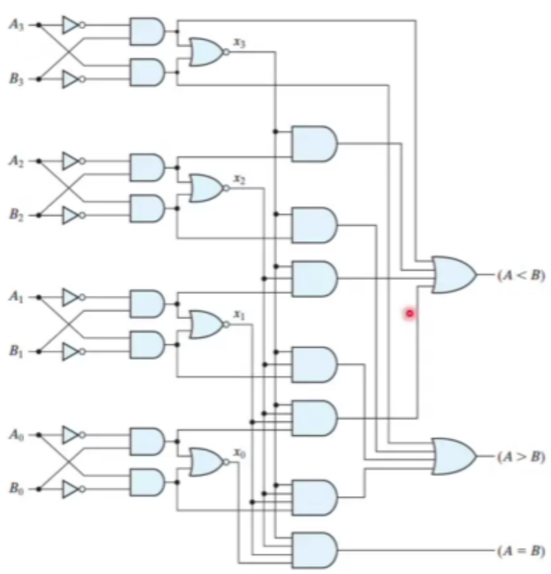


What is a system on a chip?

Microprocesser	Microcontroller
It's used in computer systems like desktops, etc.	It's used in embedded systems like washing machine, etc.

Anatomy of a smartphone- What is the iPhone SOC?
Huge number of logic blocks
CLB→ configurable logic block
Logic blocks→ consists of logic gates like NOR gates, AND gates, etc. and clocks

Combinational logic gate that does comparison



Sequential logic: uses clock cycles and does stuff, and synchronizes stuff!

3rd May

Any base to decimal conversion→ just multiply it out

$7392 = 7 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$

$$a_n a_{n-1} \dots a_0 . a_{-1} \dots a_{-m} = \sum_{i=-m}^{n} a_i \text{base}^i$$

Decimal to binary:

Eg. 41

41	Dividing by 2
20	1

41	Dividing by 2
10	0
5	0
2	1
1	0
0	1

Then write it from bottom up
Answer is *100101*.

Decimal to Octal
Eg. 153

153	Dividing by 8
19	1
2	3
0	2

Answer *231*

0.6875 in binary:
$0.6875 \times 2 = 1, 0.3750$
$0.3750 \times 2 = 0, 0.7500$
$0.7500 \times 2 = 1, 0.5$
$0.5000 \times 2 = 1, 0$

The number is written top-down.
0.1011

Binary, Hex, Octal conversion is lite, use 2, 4, and 3 digits respectively

8 bits makes up one byte (little children know this)

Summary

Start \ End	Binary	Decimal	Hexa
Binary	×	$\times 2^{n-1}$ for n^{th} digit	4 binary → Hexa
Decimal	÷2 (left) ×2 (right)	×	÷16 (left) ×16 (right)
Hexa	Each hexa → 4 binary	$\times 16^{n-1}$ for n^{th} digit	×

5th May

Complements

Given two n-digit numbers in base 'r'

Nice video: https://www.youtube.com/watch?v=4qH4unVtJkE
Negative numbers:

- Sign-magnitude
    - First bit for sign, remaining digits as usual binary
    - There's a problem: 5 + -5 in this representation is NOT zero.
- Ones' complement
    - Positive numbers as usual, negative numbers are just 1s complement (which is toggling every bit)
    - 5 → 0101
    - -3 → 1100
    - 5 - 3 → 10001 → 0001 → 1
    - So, it is offset by one and we have to do some juggling around while carrying out subtraction
- 2s complement
    - Take twos complement of every number to get its negative
    - you get one extra number
    - there are *no* signed zeroes
    - The numbers are the same as unsigned numbers mod (2^N)
    - First digit represent -2^N digit really
    - Number + its negative is 0

Addition of numbers: lite

Subtraction of numbers: M - N, where M and N are unsigned numbers

- M + (r^n - N)
- if M >= N:

    - Produces end carry, and leftmost carry digit corresponds to r^n, which needs to be discarded (n carry is formed when the first digit is +ve)
    - M - N
- if M < N:
    - Doesn't produce n carry, and result is r^n - (N - M) → We have to take r's complement and place negative sign at front

Eg. 6 - 13

- +6 → 00000110
- -13 →
    - Binary representation: (+13) → 00001101
    - Toggle: 1s complement: (-13) → 11110010
    - Add 1 to ones complement → Twos complement → 11110011
- Adding them both: 11111001 (first bit, i.e. sign bit is 1)
- end carry not there, so now do twos complement: 00000111
- This is **-7**!

Eg. 15 - 7:

- 15 → 00001111
- -7 → 11111001
- Add up:
  100001000 → **8**!

May 6

Boolean Algebra

Postulates of Boolean Algebra:
For a set B (not necessarily {0,1} btw):

1. Closure
2. Identity
    1. x + 0 = x
    2. x . 1 = x
3. Commutative
4. Distributive
    1. x + yz = (x + y)(x + z)
    2. x (y + z) = xy + xz
5. Complement
   5. x + x' = 1
   6. x x' = 0
6. There are at least two elements x and y such that $x \neq y$

To solve question, we need identity, distributive, complement

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

x	x'
0	1
1	0

Theorems of Boolean Algebra

1. **x + x = x, x. x = x**
   = x (1 + 1) → Distributive Law
   = x . 1 → OR definition
   = x → Identity element property

2. **x + 1 = 1, x . 0 = 0**
   = x + 1
   = x + x + x' → Complement
   = x + x' → Theorem 1
   = 1 → Complement definition

3. **Involution: (x')' = x**
   (x')' = (x')' + 0
   = (x')' + x x' → Complement definition
   = (x + (x')')(x' + (x')') → Distributive Law
   = x + (x')' → Complement
   = (x + (x')')(x + x') → Complement
   = x + (x') x → Property again!
   = x (1 + x') → Distributive
   = x

4. Associativity
xy = yx (Commutativity)
x = (a + b) + c
y = a + (b + c)
xy = ((a + b) + c)(a + (b + c)) = y
By symmetry, yx = x
So, x = y

5. De morgan!
Lemma:

Given:

- a + b = a + a' = 1
- ab = aa' = 0
  To prove:
- a = b'

Multiply first one by b
ab + b = ab + a'b
0 + b = a'a + a'b
b = a'(a + b) = a' 1 = a'

Now
x'y' + (x + y) = 1 (Using lemma)
(x + y)' + (x + y) = 1

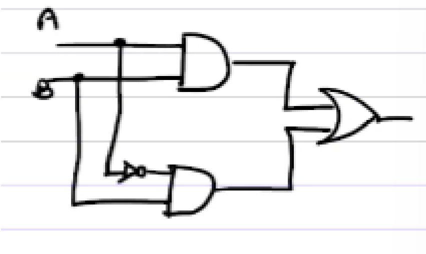
Therefore:

- (x + y)' = x'y'

6. Absorption
x + xy
= x (1 + y) → Distributive
= x

Usefulness of Boolean algebra: simplifying Boolean expression
The electrical idea is to keep number of gates minimum

a) AB + A' B = (A + A') B = B



Canonical Form of Boolean Expressions
Boolean expression consists of variables in either normal form or its complement

n variables with AND (lowercase)

- We can have 2^n "minterms" (minterm is just every possibility basically)
- minterms are ordered as if they were binary numbers (the image below makes it clear)
- $m_0 m_1 m_2 \ldots m_{2^n-1}$

- We consider "1" here, as the output 1 is unique
  n variables with OR (uppercase):
- We have 2^n "maxterms"
- $M_0 M_1 M_2 ... M_{2^n-1}$
- We consider "0" here, as the output 0 is unique

x	y	z	Minterms	Maxterms
0	0	0	$x'y'z'$ (m_0)	$x+y+z$ (M_0)
0	0	1	$x'y'z$ (m_1)	$x+y+z'$ (M_1)
0	1	0	$x'yz'$ (m_2)	$x+y'+z$ (M_2)
0	1	1	$x'yz$ (m_3)	$x+y'+z'$ (M_3)
1	0	0	$xy'z'$ (m_4)	$x'+y+z$ (M_4)
1	0	1	$xy'z$ (m_5)	$x'+y+z'$ (M_5)
1	1	0	xyz' (m_6)	$x'+y'+z$ (M_6)
1	1	1	xyz (m_7)	$x'+y'+z'$ (M_7)

> ⓘ **Boolean**
>
> Any boolean function can be expressed as
>
> - sum of minterms
> - products of maxterms!

Consider the function
f1(x, y, z) = xyz' + xy'z + xy'z'
This function is 1 when:
x	y	z
1	1	0
1	0	1
1	0	0
And 0 for anything else

f1(x, y, z) → $\Sigma\ 4, 5, 6$
f1 → $\Pi\ 0, 1, 2, 3, 7$

f2(x, y) → xy + xy' + x'y'
→ $\Sigma\ 0, 2, 3$
→ $\Pi\ 1$ = (x' + y)
This is intuitive enough!

f = Π Some maxterms = Σ of other minterms

- It's easy enough to see that $m_i' = M_i$
  - Say m_2 = x + y' + z
  - $m_2' = x'yz' = M_2$
- $(\sum_{\text{i runs over whatever}} m_i)' = \sum_{\text{i runs over everything else}} m_i$
  - Proof:
    - $f = \sum m_i$
    - f is 1 for certain values of i
    - f' is 0 for those values of i
    - f' is 1 for all other values of i
    - $f' = \sum_{\text{j runs over everything else}} m_j$
- $m_i = \Pi_{\text{j = everything else}} M_{\text{j}}$
- $f = \Sigma_{\text{i runs over whatever}} m_i$
- $f' = \sum_{\text{i runs over everything else}} m_i$
- $f'' = f = \prod_{\text{i runs over everything else}} m_i' = \prod_{\text{i runs over everything else}} M_i$

Dual of a function:

- Replacing all AND by OR and vice versa
- Replacing all 1s by 0s

Taking complement of a function is the same as taking dual of the function and replacing each literal by it's complement (this follows from de Morgan's theorem).

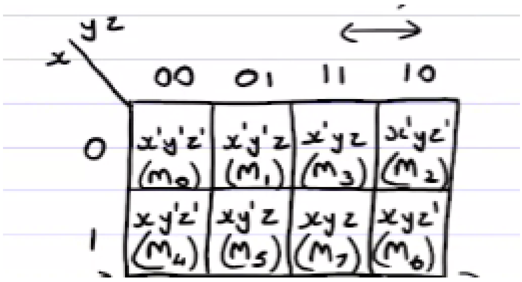
To go from positive logic to negative logic, we just have to take the dual!

May 9

Gate level minimization: K-maps

- The objective is find the optimal gate level implementation of the Boolean function
  - Complexity of digital logic circuit is related to the complexity of the algebraic expression
  - Truth tables are unique
  - Pictorial representation is Karnaugh map

- Consider f(x,y,z)



Only one bit changes between adjacent cells (in both axes)

Adjacencies	Literals
1	3
2	2
4	1
8	0

f2 = Sum(3,5,6,7)
x \ yz	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Adjacent cells in K-map represent whatever didn't change!

f2 doesn't change once in x direction, once in y direction, once in z direction
So, $f2 = xy + yz + zx$

f1 = Sum(1,4,7)
x \ yz	00	01	11	10
0	0	1	0	0
1	1	0	1	0
No adjacency, so it's simplified already

f3 = sum(3,4,6,7)
x \ yz	00	01	11	10
0	0	0	*1*	0
1	*1*	0	*1*	*1*
f3 = yz + xy + xz' = yz + xz'

- Reason: xy terms is redundant!

So the aim is to cover all minterms.

f4 = sum (0,2,3,4,5)
x \ yz	00	01	11	10
0	1	0	0	1
0	1	1	0	1

- Four cell adjacency!

f4 = z' + x y'

f5 = Sum(1,2,5,6,7) = Prod(0, 3, 4)
x \ yz	00	01	11	10
0	0	1	0	1
1	0	1	1	1
f5 = y'z + yz'+ xy

f5 = (y + z) . (x + y' + z')

f6 = A'C + A'B + A B' C + B C

A \ BC	00	01	11	10
0	0	1	1	1
1	0	1	1	0

f6 = C + A'B

Four variable K-maps
wx \ yz	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

Adjacencies	Literals
1	4
2	3
4	2
8	1
16	0

f1 = Sum(0,1,2,4,5,6,8,9,12,13,14)

wx \ yz	00	01	11	10
00	1	1	0	1
01	1	1	0	1
11	1	1	0	1
10	1	1	0	0

f1 = y' + w'z' + x z' = y' + z' (x + w')

f1' = yz + w x' y
f1 = (y' + z') (w' + x +y') = w'y' + w'z' + x y' + x z' + y'z' + y'
= y' + z' (x + w')

Prime Implicant: Total number of adjacencies in the K-map (including redundancies)
Essential Prime Implicant: Prime implicants that cover a minterm no other implicant covers
Redundant Implicant: Implicant for which each minterm is covered by some other implicant

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

Eg.

- m5 is covered only by one prime implicant, BD is essential prime implicant
- m0 is covered only by B' D'

Simplification of Kmap

K-Map Simplification Process:

Step 1: Represent all the minterms/maxterms

Step 2: Identify the essential prime implicants

Step 3: Cover all other minterms w/ max adjacency (prime implicants)

Step 4: Ensure no redundant terms

F

1.

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

2. Essential implicants:

- B'D' (for 0010 and 1010)
- B'C'
- A'C D

3. All minterms are covered!
4. F = B'D' + B'C' + A' C' D

May 12

Don't care conditions:

Eg. Binary coded decimal (BCD)
Representation of decimals in binary numbers
396 → Each value is represented by corresponding value
396 → 0011 1001 0110

BCD doesn't care about remaining inputs like 1111 (corresponding to 15). So, *we don't care* about that part.
Don't cares are represented by X in K-map

F = $\Sigma(1, 3, 7, 11, 15)$
d = $\Sigma(0, 2, 5)$

wx \ yz	00	01	11	10
00	X	1	1	X
01	0	X	1	0
11	0	0	1	0
10	0	0	1	0

SOP
F = yz + w'x' or yz + w'z
POS
F = z (w' + y)

May 13

> **NAND and NOR gates are easier to realize**

AND, NOT, OR gates → NAND and NOR gates
NAND and NOR gate → universal gate → we can create any other gates using these gates

- NOT gate → NAND(x, x)
- AND gate → NOT(NAND(x, x))
- OR gate → NAND(NOT(x), NOT(y))

> **Boolean function implementation**
>
> 1. Represent function in K-map
> 2. Obtain simplified function in terms of Boolean operators
> 3. Convert function to NAND/NOR logic
> 4. Implement using NAND/NOR gates

Eg.
F = AB + CD
NAND logic

- (A' + B')' + (C' + D')' = NOR(NOR(NOR(NOR(A), NOR (B)), NOR(NOR(C), NOR(D))))

Eg. F = $\Sigma(1, 2, 3, 4, 5, 7)$

1.

A \ BC	00	01	11	10
0	0	1	1	1
1	1	1	1	0

F = A B' + A' B + C
2. Done
3. NAND((NAND(A, NAND(B)), (NAND(NAND(A), B)), NAND(C))
NOR(NOR(C, NOR(NOR(A), B), NOR(A, NOR(B))))

Eg. F = ((A + B)(C + D)E)'
NOR(NOR(NOR(A, B)), NOR(C, D), NOR(E)))

In general, look at complement:

- AND-NOR ↔ NAND-AND → SOP form combining 1s
- OR-NAND ↔ NOR-OR → POS form combining 0s

x \ yz	00	01	11	10
0	1	0	0	0
1	0	0	0	1

SOP form
From zeroes, we get F'
F' = z + xy' + x'y
F = (z + xy' + x'y)' → AND-NOR realization
F = ((x'y)' (x'y)' z') → NAND-AND realization

POS form
F' = (x + y + z)(x' + y + z')
F = ((x + y + z) (x' + y + z'))' → OR-NAND realization
F = (x + y + z)' + (x' + y + z')' → NOR-OR realization

XOR gate

- Parity checking
- Binary adders
- Error detection and correction

$x \oplus y = xy' + x'y = (x + y')(x' + y)$

Identities:

- $x \oplus 0 = x$
- $x \oplus 1 = x'$

- $x \oplus x = 0$
- $x \oplus x' = 1$
- $x \oplus y' = x' \oplus y = (x \oplus y)'$
- Using NAND gate: NAND(NAND(x, y'), NAND(x', y))

Eg. $A \oplus B$

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

$A \oplus B \oplus C$

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

$A \oplus B \oplus C \oplus D$

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

17th May

Combinatorial Logic Circuits

Interconnection of logic gates to accomplish a logic operation

- Binary addition
- Binary subtraction
- Binary multiplication
- Comparison between binary numbers
- Encoding, decoding, etc.

Design Procedure

1. Determine number of inputs and outputs

2. Derive truth table
3. Obtain simplified Boolean expressions
4. Implement logic circuit

One-bit adders

Half-adder

Carries out binary addition of two binary inputs

x	y		c	s
0	0		0	0
0	1		0	1
1	0		0	1
1	1		1	0

$s = x \oplus y$
$c = xy$

Full Adder

x	y	z		c	s
0	0	0		0	0
0	0	1		0	1
0	1	0		0	1
0	1	1		1	0
1	0	0		0	1
1	0	1		1	0
1	1	0		1	0
1	1	1		1	1

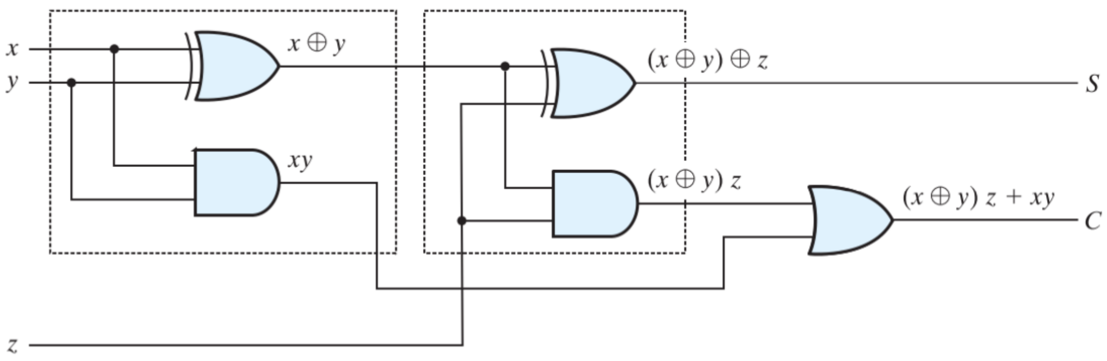
- **K-map for s**
x/yz	00	01	11	10
0|0|1|0|1
1|1|0|1|0

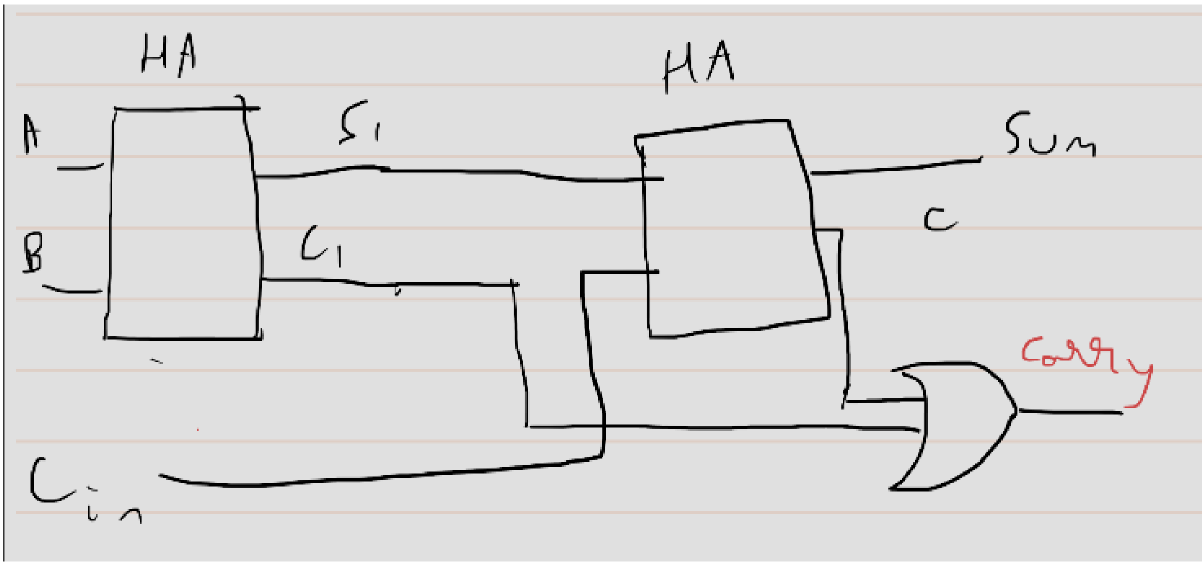
$s = xy'z' + x'yz' + x'y'z + xyz = z'(x \oplus y) + z((x \oplus y)') = (x \oplus y) \oplus z$

- **K-map for c**
x/yz	00	01	11	10
0|0|0|1|0
1|0|1|1|1

$c = xy + x'yz + xy'z = xy + z(x \oplus y)$

> We can just use the XOR used in s for c





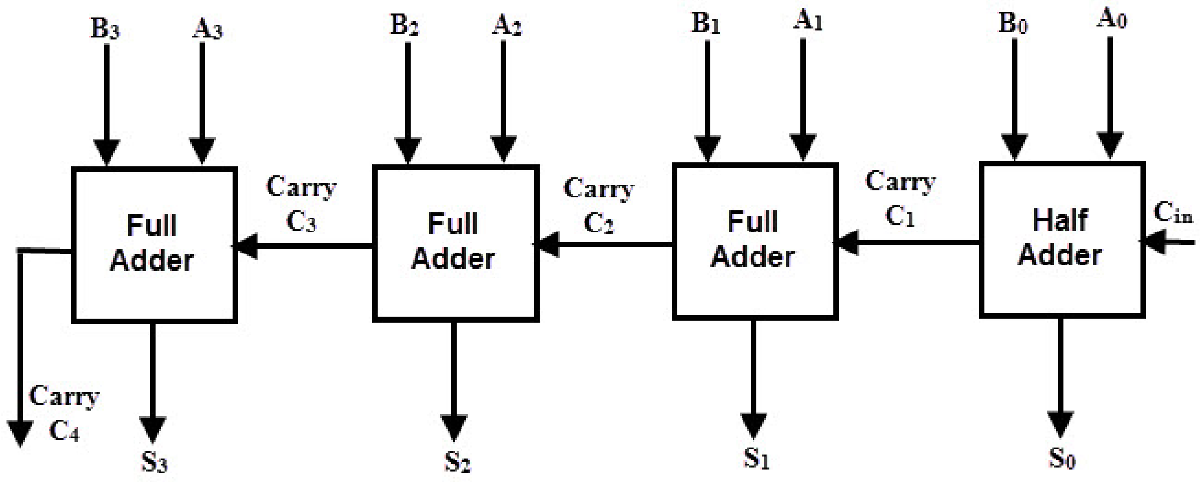
Full Adder→ 2 Half Adder + OR gate

Many bit adders

4-bit adder

Ripple Carry adder
Nice video https://www.youtube.com/watch?v=wvJc9CZcvBc&ab_channel=BenEater
Augend + Addend

Output carry of ones place becomes input to next digit
Output carry of twos place becomes input to fours digit, etc.



✕ **Problems with this method**

We have to do this sequentially, one by one, so it takes time. We have to wait for the carry

Carry look ahead logic

$C_{i+1} = A_iB_i + C_i(A_i \oplus B_i)$

Carry Generate → $G_i = A_iB_i$
Carry Propagate → $P_i = (A_i \oplus B_i)$

$S_i = P_i \oplus C_i$
$C_{i+1} = G_i + C_iP_i$

C_0 → Input carry
C_1 → $G_0 + P_0C_0$
C_2 → $G1 + P1G_0 + P_1P_0C_0$
C_3 → $G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

Binary Subtraction

+49 →00110001
-49 →1s: 11001110 → Accomplished by XOR gate
2s: 11001111 → Can be done using initial carry C_0

Extra sign bit **M** → M = 0, add; M = 1, subtract

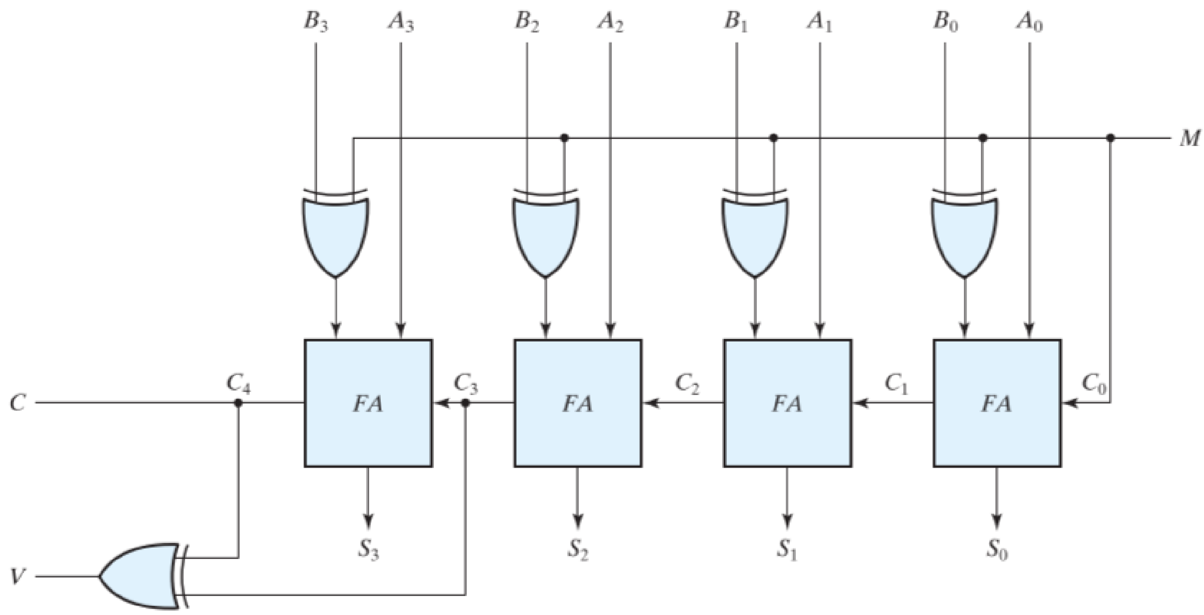


FIGURE 4.13
Four-bit adder–subtractor (with overflow detection)

- Thinking about the addition part:
  - If M is 0, 0 $\oplus$ A = A. So, we just get out regular 4-bit ripple carry adder
- Think about the subtraction part:
  - If M is 1, 1 $\oplus$ A = A'. So, each bit first gets toggled (which gives 1s complement). Also, M acts as the input carry C_0, so we account for the 1 added in 2s complement by setting $C_0 = 1$

The overflow bit V

If $V = C_4 \oplus C_3$ = 0 → Correct
Else → Wrong
If we're adding a positive and negative number, our answer is always correct

- A3 is 0, B3 is 1
- If C3 is 0, C4 is 0, so V is 0 and we're good
- If C3 is 1, C4 is also 1, so V is 0 and we're good

If we're adding two positive numbers, overflow *might* occur.

- C4 is 0 if we're adding two positive numbers
- 0011, 0010
  - C3 is 0
  - C4 is 0
  - So, the output is correct
- 0111, 0100
  - C3 is 1
  - C4 is 0
  - So, the output is wrong and overflow has occurred

If we're adding two positive numbers, overflow *might* occur.

- C4 is 1 if we're adding two positive numbers
- 1011, 1010
    - C3 is 0 (since we're toggling bits here, C3 being 0 means A3 and B2 are like 0, which means they're "large" in magnitude)
    - C4 is 1
    - So, the output is wrong and overflow has occurred
- 1111, 1100
    - C3 is 1
    - C4 is 1
    - So, the output is correct

Eg.
70 → 01000110
80 → 01010000
10010110 = -106

-70 → 10111010
-80 → 10110000
01101010 (C8 = 1) → 106

May 26

Half Subtractor

A - B
B_0 is a boolean that keeps track of if we have to borrow from the next digit
A	B	D	B_0
0	0	0	0
0	1	1	1
1	0	1	0
1	0	0	0

$D = A \oplus B$
$C = A'B$

Full Subtractor

A - B - C
A	B	C	D	B_0
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1

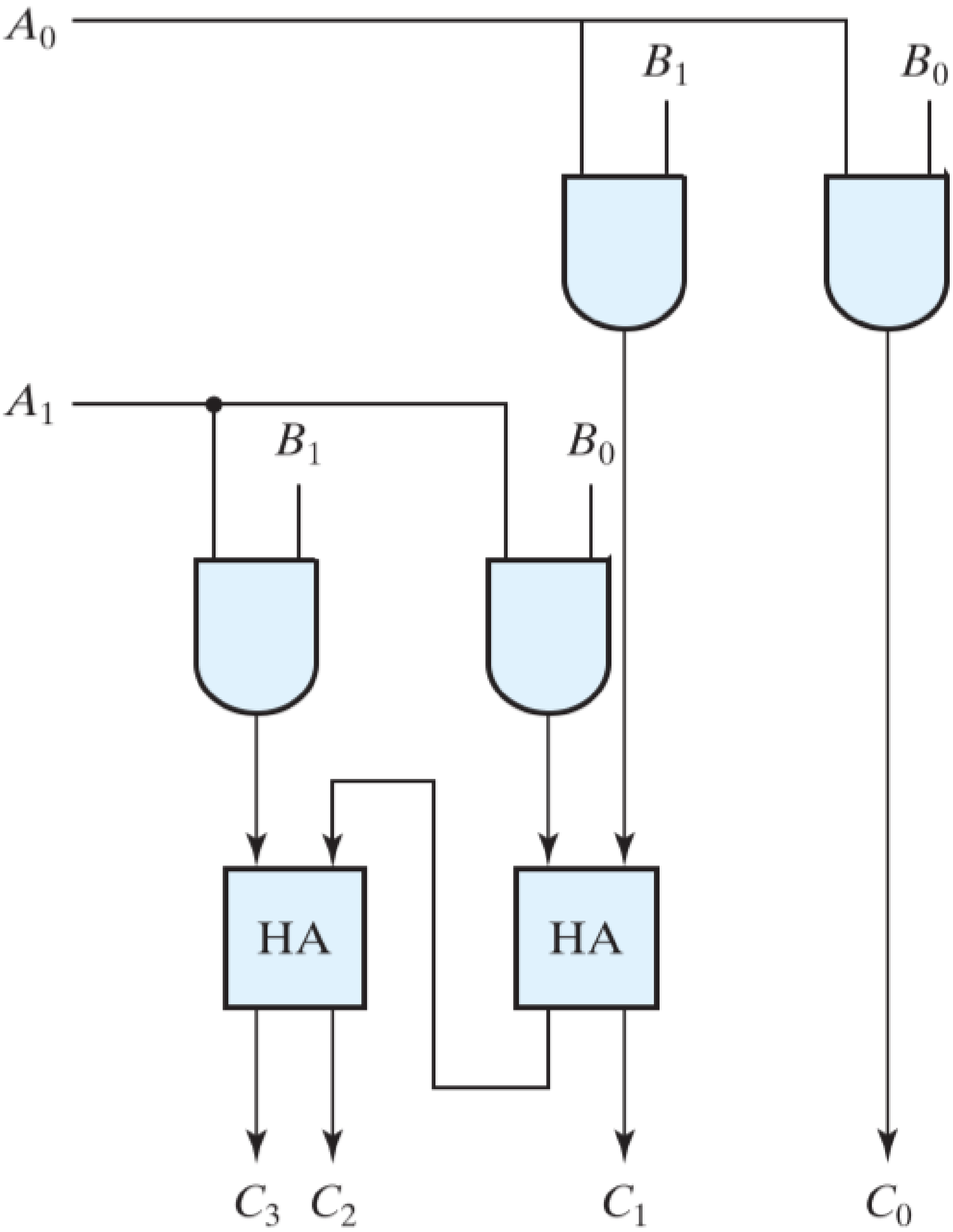
1 | 0 | 0 | 1 | 0
1 | 0 | 1 | 0 | 0
1 | 1 | 0 | 0 | 0
1 | 1 | 1 | 1 | 1
D = $A \oplus B \oplus C$

Binary Multiplication

Multiplicand → B1 B0
Multiplier → A1 A0

B1 B0
A1 A0

P0 = A0B0
P1 = A0B1 + A1B0
P2 = Carry + A1B1
P3 = Carry



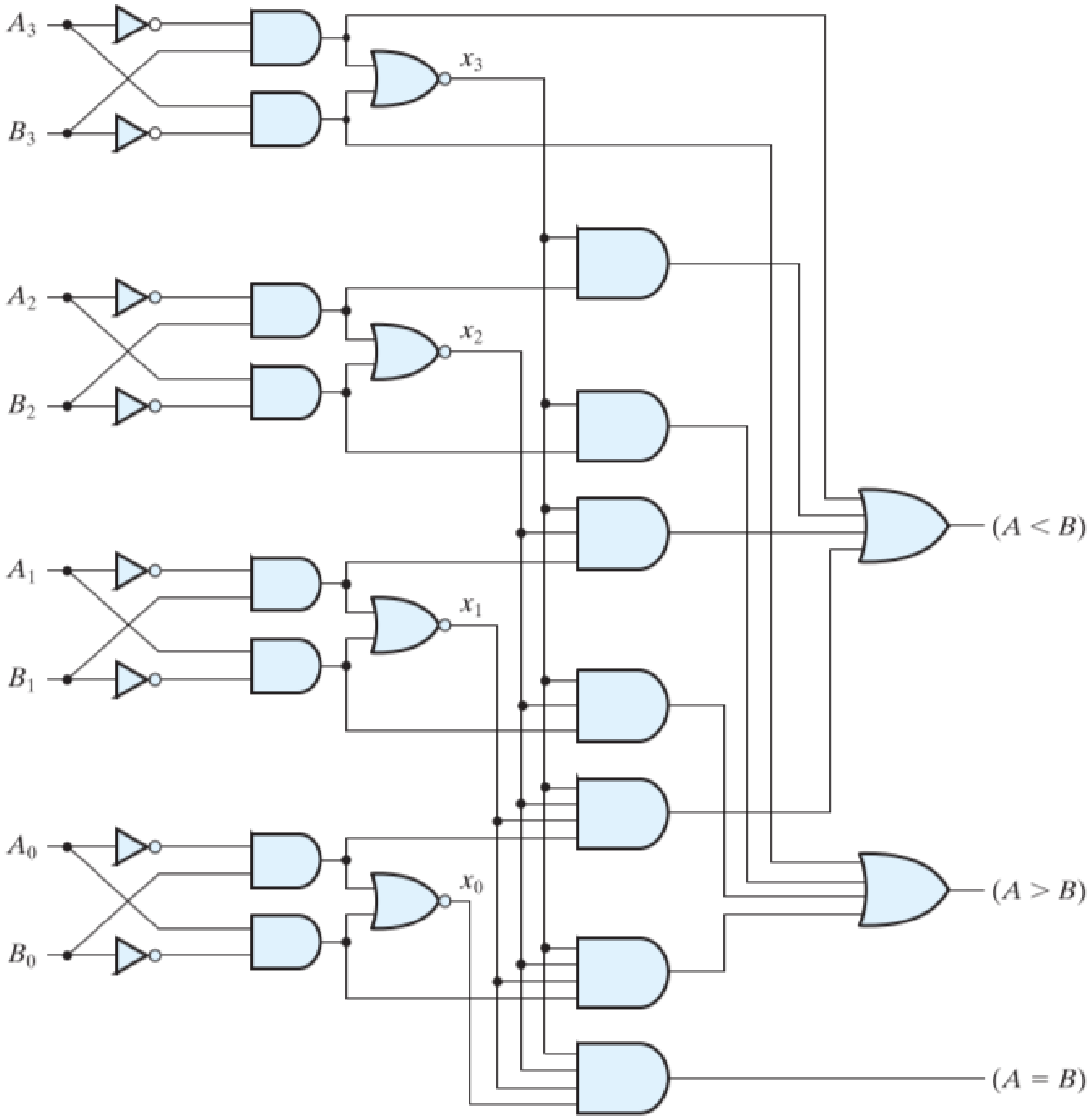
2×2 multiplication →4 AND gates and 2 HA

May 20

Magnitude Comparator

A = A3A2A1A0
B = B3B2B1B0
$A_i \odot B_i = x_i$, where $\odot$ is XNOR gate

- A = B if $A_i = B_i \forall i$
  - A = B if $\Pi A_i \odot B_i = \Pi x_i = 1$
- A > B → $A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$
- A < B → $A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1 + x_3x_2x_1A_0'B_0$



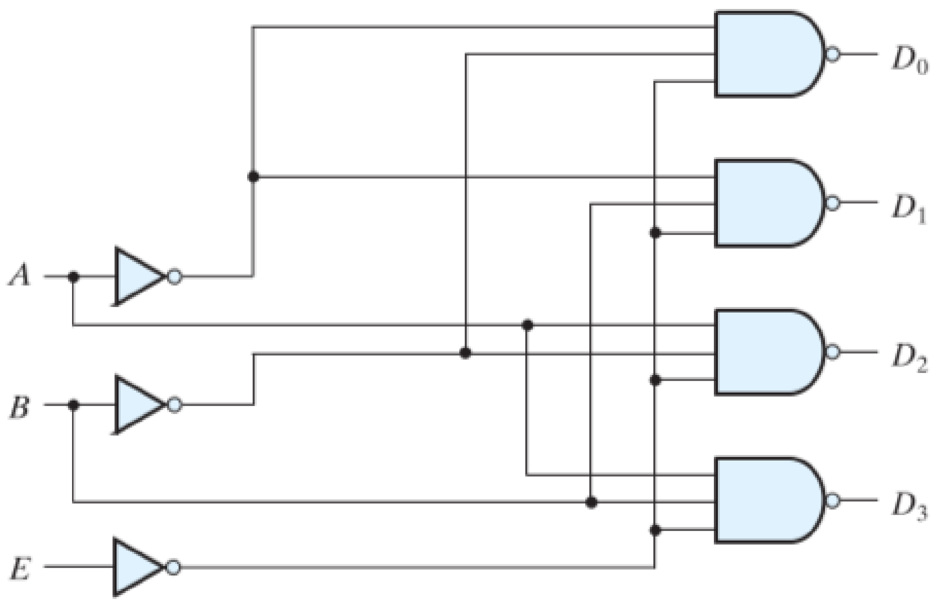
Decoder / Demultiplexer

2 × 4 Decoder with Enable Pin

Enable 1 → Chip not ready

Enable 0 → Chip ready to work

Enable	A	B	D0	D1	D2	D3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0



(a) Logic diagram

Also acts as demultiplexer

- if you want your data to go to say D3, we can give A = 1, B = 1. The output is exactly the same in the required pin, and remains always 1 in the other pins
- after two seconds say we want data to go to D1, we can give A = 0, B = 1

> **Info**
>
> It's possible to use the enable pin as in input, in that case we get $2^{(n+1)}$ outputs

Priority Encoders

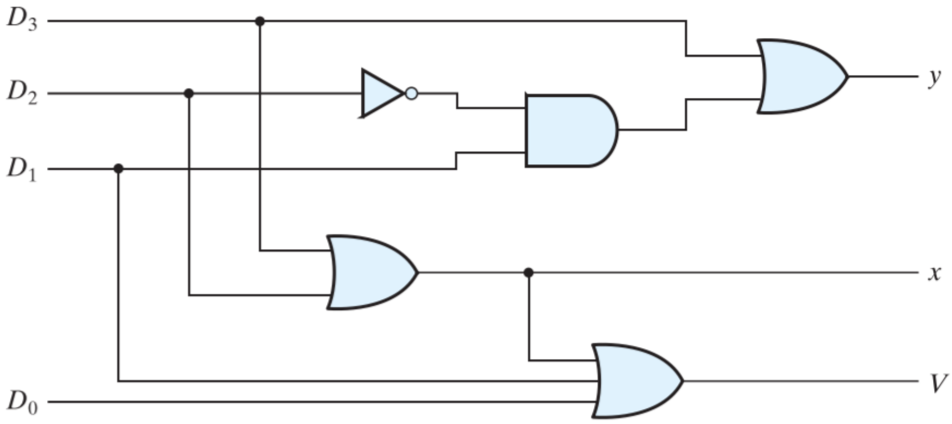
2^n bits to n bits

V → Validity indicator

- 0 if all 0s
- 1 otherwise

Least Priority → D0	D1	D2	Highest Priority → D3	x	y	v
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

- V = D0 + D1 + D2 + D3
- x = D2 + D3
- y = D3 + D2' D1



Multiplexer

S → Select pin
x → Input
y → Output

S1	S0	y
0	0	x0
0	1	x1
1	0	x2
1	1	x3

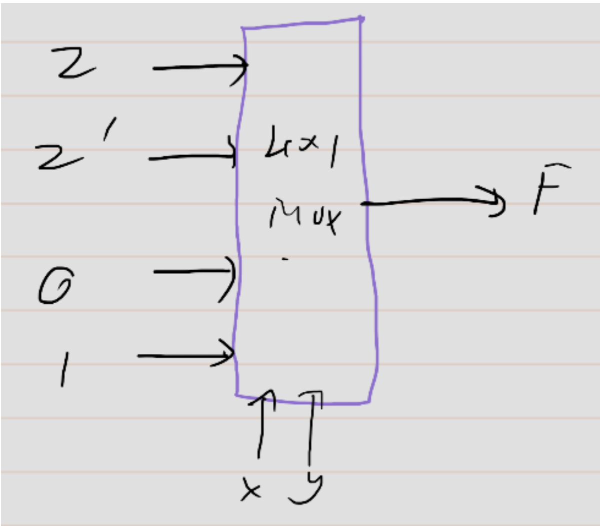
Boolean Function implementation

Using Multiplexer

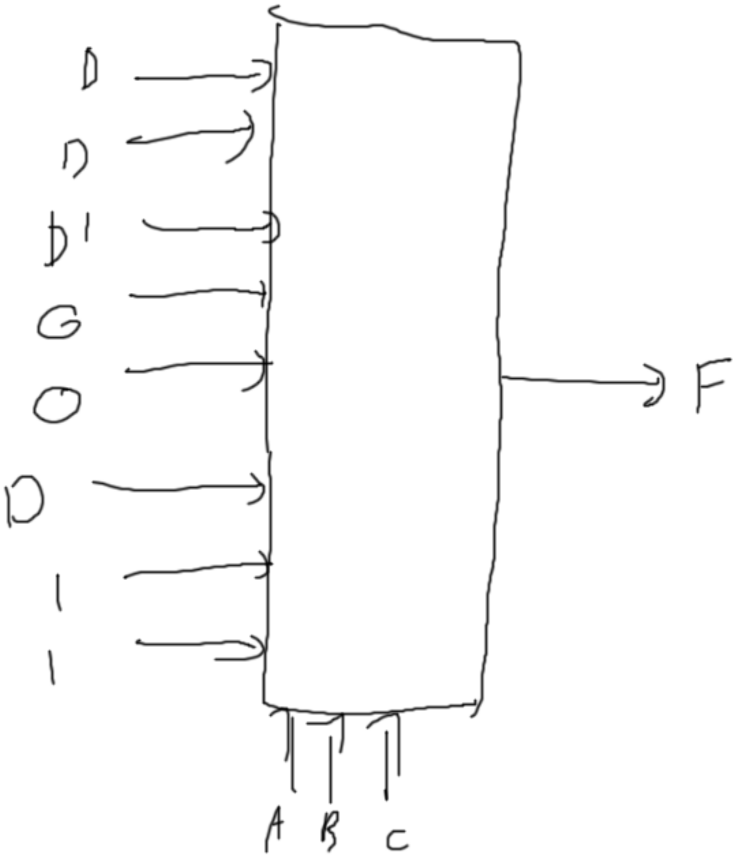
Implement $f(x,y,z) = \sum(1,2,6,7)$ using 4×1 mux

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table



$$F(A, B, C, D) = \sum(1, 3, 4, 11, 12, 13, 14, 15)$$

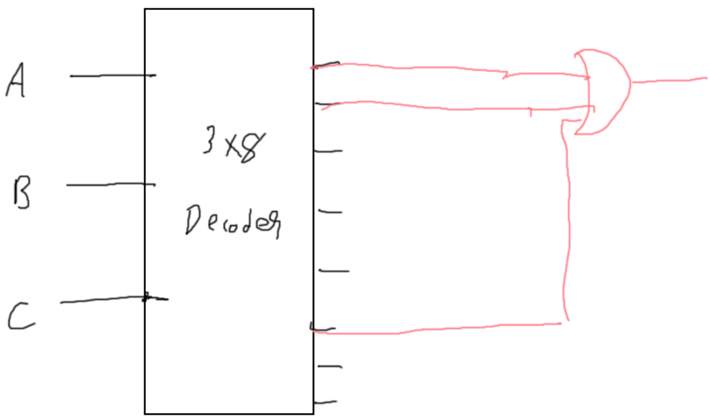


General:
n variables

- n - 1 select pins
- 2^(n - 1) inputs

Using Decoder

f(A, B, C) = A'BC + A B C' + ABC



Binary Coded decimal:
Integers from 0 - 9 → as such
10 → 15 are don't cares for any operation

Gray's Code: adjacent numbers vary only by one bit

	Gray code			
	4	**3**	**2**	**1**
0	0	0	0	0
1	0	0	0	1
2	0	0	1	1
3	0	0	1	0
4	0	1	1	0
5	0	1	1	1
6	0	1	0	1
7	0	1	0	0
8	1	1	0	0
9	1	1	0	1
10	1	1	1	1
11	1	1	1	0
12	1	0	1	0
13	1	0	1	1
14	1	0	0	1
15	1	0	0	0

3's excess: 3's excess number = Binary number + 0011 (i.e. "3"), from 0 - 9

- Taking 9's complement is just toggling all bits!