ECEN 240 Brother Jack!

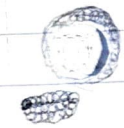
Analog - continuous
Digital - 0{or}1 → on {or} off

Chapter 1: Intro to Digital Systems Design

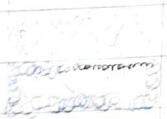
Digital → binary values
Analog → continuous range

Combinational circuit → simple & direct function of their inputs
Sequential circuit → possesses memory
↳ information + state = outcome

♡♡
Isabel
Mooney



Seth



Chapter 2 – Number Systems & Binary

Encoding

$735 = 7 \times 10^2 + 3 \times 10^1 + 5 \times 10^0$

$73.502 = 7 \times 10^1 + 3 \times 10^0 + 5 \times 10^{-1} + 2 \times 10^{-3}$

$52.1_8 = 5 \times 8^1 + 2 \times 8^0 + 1 \times 8^{-1} = 40 + 2 + 1/8 = 42.125_{10}$

2.1.1 Conversion from Decimal to other Bases

$112_{10} \rightarrow$ base 6?

1. Determine highest power of 6 that is less than 112
2. Us 6^1 less than the r. 4
3. Repeat for 6^0

$$\begin{array}{r} 112 \\ -108 \\ \hline 4 \\ -0 \\ \hline 4 \\ -4 \\ \hline 0 \end{array} \quad \begin{array}{l} 3 \times 6^2 \\ 0 \times 6^1 \\ 4 \times 6^0 \end{array} \quad = 304_6$$

$12.625_{10} \rightarrow$ base 2?

$$\begin{array}{r} 12.625 \\ -8 \\ \hline 4.625 \\ -4 \\ \hline 0.625 \\ -0.5 \\ \hline 0.125 \\ -0.125 \\ \hline 0 \end{array} \quad \begin{array}{l} (1 \times 2^3) \\ (1 \times 2^2) \\ (1 \times 2^{-1}) \\ (1 \times 2^{-3}) \end{array}$$

$2^3\ 2^2\ 2^1\ 2^0\ 2^{-1}\ 2^{-2}\ 2^{-3}$

$= 1100.101_2$

10100111_2
A F

2.2 Hexadecimal

Digit Symbol	Decimal Equivalent
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
A	10
B	11
C	12
D	13
E	14
F	15

$1A2F_{16} = 1\times16^3 + 10\times16^2 + 2\times16^1 + 15\times16^0 = 6,703_{10}$

$FF_{16} = 15\times16^1 + 15\times16^1 = 255_{10}$

0000 1010 0110 1000 → $0A68_{16}$

* Remember to add zeros to the left of binary to make groups of 4 before converting to hex *

$FADE27_{16} = 1111\ 1010\ 1101\ 1110\ 0010\ 0111_2$

2.3 Binary-Coded Decimal

263_{10} → 0010 0110 0011 $_{BCD}$

$263_{10} = 100000111_2$

2.4.1 ASCII Code

"American Standard Code for Information Interchange"

* Table in Appendix A *

ex;

Hex value	Character
40	@
41	A
42	B
...	...

2.4.2 Gray Codes

Two adjacent values in the code differ by a single bit.
Not unique, many ways to write them.

ex;

Decimal Value	Gray Code
0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	100

(7 also loops back to 0)

ex;

Decimal	Gray Code
0	101
1	011
2	010
3	110
4	100
5	000
6	001
7	101

(7 also loops back to 1)

★ON EVERY TEST★

Ex; largest number possible in x bits: $2^x - 1$

largest w/ 6 bits: $2^6 - 1 = 64 - 1 = 63$

How many numbers can 6 bits represents?: 64 (includes zero)

Class:

$213_{10} = ???_2$

256 128 64 32 16 8 4 2 1

1 0 0 1 0 1 0 1

128 192 208

$11010_2 = 16 + 8 + 2 = 26$

1.) $0.1_2 = \frac{1}{2}$ 0.5

3.) $0.101_2 = \frac{1}{2} + \frac{1}{8} = \frac{5}{8}$ 0.625

HW

Carlson

1.) $0001\,1100\,1110_2$
1 C E

2.) decimal value of AC_{16}
$10 \cdot 16^1 + 12 \times 16^0$
$160 + 12 = 172$

$367_{16} = 0 \times 367$
↳ Why?

64 32 16 8 4 2 1

ABE
$10 \times 16^2 + 11 \times 16^1 + 14 \times 16^0 = 2750$

32 16 8 4 2 1
111011 = 59

$B88_{16}$
1011 1000 1000

128
256 64 32 16 8 4 2 1
111 111 111

$1,309_{10}$ → binary
1,309/2 = 1
654/2 = 0
327/2 = 1
163/2 = 1
81 = 1
0
0
0
1
0
1

(9.) $E_{16} = 14$

(10.) $43,346_{10}$

A952

43,346
- 40960 10×16^3
2386
- 2304 9×16^2
82
- 80 5×16^1
2 2

*in computer → $x' = x$

Chapter 4

4.1 Intro to Boolean Algebra and Truth Tables

A	B	A·B
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

OR

A	A'
0	1
1	0

NOT

4.2 Truth Tables for Arbitrary Functions

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

$F = A' \cdot B' + A \cdot B$

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

$F = A \cdot B'$

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

$F = A' \cdot B' + A' \cdot B + A \cdot B' + A$

{or} $F = 1$

Order of operations → NOT, then AND, then OR

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

$F = A' \cdot B \cdot C + A \cdot B' \cdot C + A \cdot B \cdot C' + A \cdot B \cdot C$

{or} $F = A'BC + AB'C + ABC' + ABC$

4.4 Converting Boolean Functions to Truth Tables

A	B	F
0	0	0
0	1	1 } AB
1	0	1 } A
1	1	1 }

$F = A + A'B$

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
BC { 0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1 } AB
BC { 1	1	1	1 }

$F = AB + BC$

The '111' row is covered by both AB as well as BC

4.5 Boolean identities and theorems

Single-Variable Theorems

$A \cdot 0 = 0$

A	0	A·0 = 0
0	0	0
1	0	0

$A \cdot 1 = A$

A	1	A·1 = A
0	1	0
1	1	1

$A + 0 = A$

$A + 1 = 1$

$A' \cdot A = 0$

A'	A	A'·A = 0
1	0	0
0	1	0

$A' + A = 1$

A'	A	A'+A = 1
1	0	1
0	1	1

4.5.2 Two-Variable Theorems

$A + A'B = A + B$

A	B	A'B	A+A'B	A+B
0	0	0	0	0
0	1	1	1	1
1	0	0	1	1
1	1	0	1	1

A+A'B = A+B

→ Can be used to prove more complex theorems:

$(N + M + K') + (N + M + K')' \cdot WXY = (N + M + K') + WXY$

where $(N + M + K') \rightarrow A$ and $WXY \rightarrow B$

$A(A' + B) = AB$

4.5.3 Commutative, Associative, and Distributive Theorems

$A \cdot B = B \cdot A$

$(A \cdot B) \cdot C = A \cdot (B \cdot C)$

$A + B = B + A$

$(A + B) + C = A + (B + C)$

Distributive Laws: (Two theorems)

$A(B + C) = AB + AC$

$A + BC = (A + B)(A + C)$ ⟶ * How does this make sense? *

↳ $AA + AB + AC + BC$

↳ $A + AB + AC + BC = F$

↳ $AB + AC$ is already covered by $A \cdot A$ has to be true for either AB or AC to be true, might as well just write A

4.5.4 The simplification Theorem:

$AB' + AB = A$ ⟶ $A(B' + B) = A(1) = A$

$(A + B')(A + B) = A$

↳ $AA + AB + AB' + B'B$

↳ $A + A + 0 = A$

4.6 Summary of the Boolean Theorems

$A \cdot 0 = 0$	$A + 1 = 1$
$A \cdot 1 = A$	$A + 0 = A$
$A \cdot A' = 0$	$A + A' = 1$
$A + A'B = A + B$	$A(A' + B) = AB$
$AB = BA$	$A + B = B + A$
$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
$A(B + C) = AB + AC$	$A + BC = (A + B)(A + C)$
$AB' + AB = A$	$(A + B')(A + B) = A$

Class 1/17/2024

$F = A' + BC$

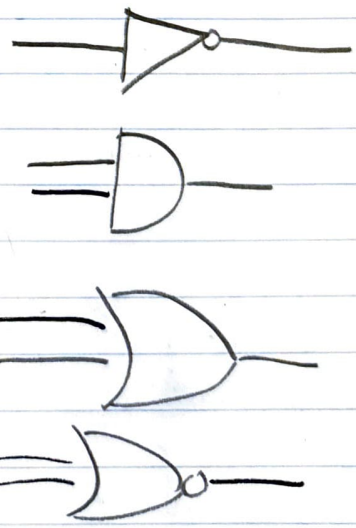
A	B	C	A'	BC	F = A'+BC
0	0	0	1	0	1
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	1	1	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	1	1

ex; Design belt buzzer for car

1.) Car on/off
2.) seat belt attached/unattached
3.) seat is empty

← Buzzer

C	B	E	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

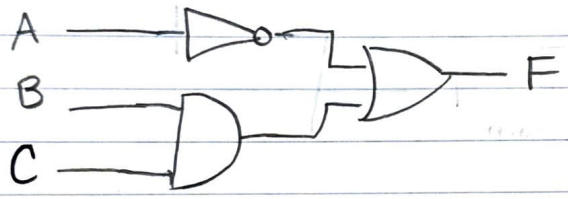


$F = CB'E$

3, 6, 9, 12

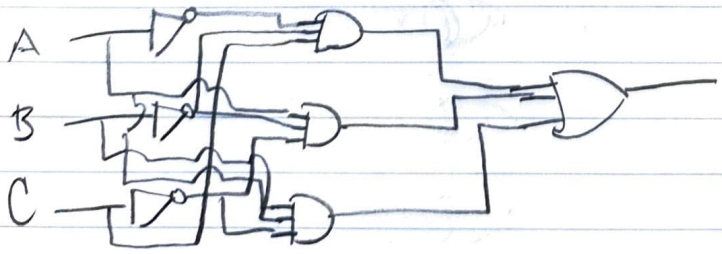
$F = A' + BC$

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

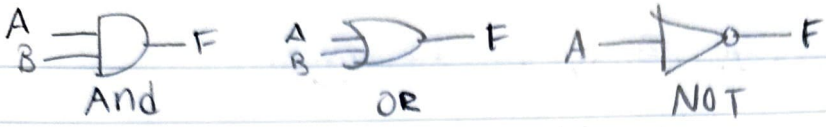


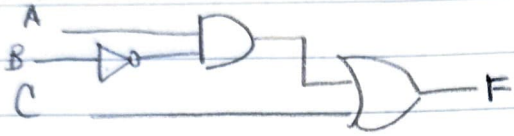
A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

$F = A'B'C + AB'C' + ABC'$



Chapter 5: Logic Gates

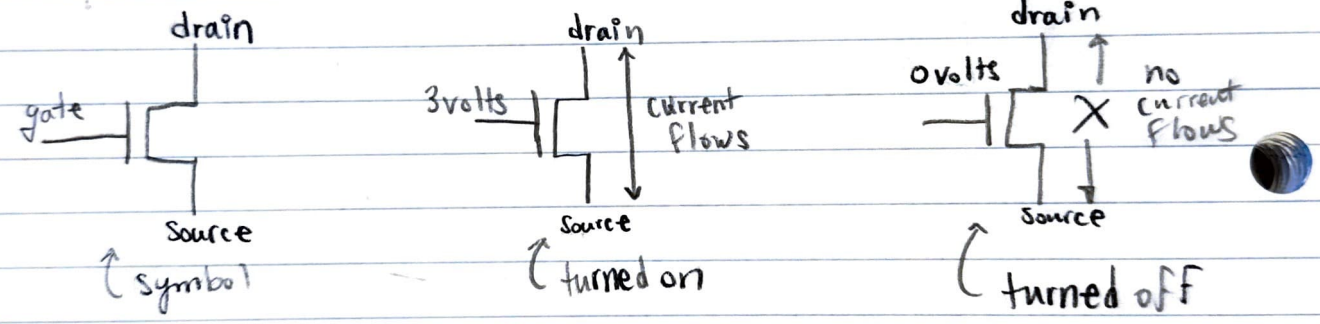




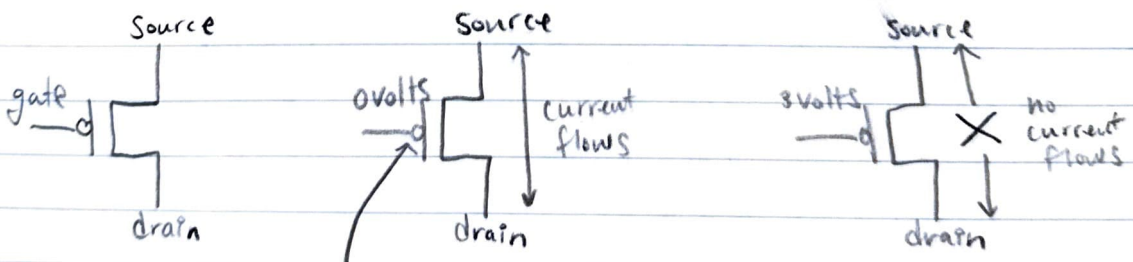
$F = AB' + C$

5.2 Transistors - The Building Blocks of Gates

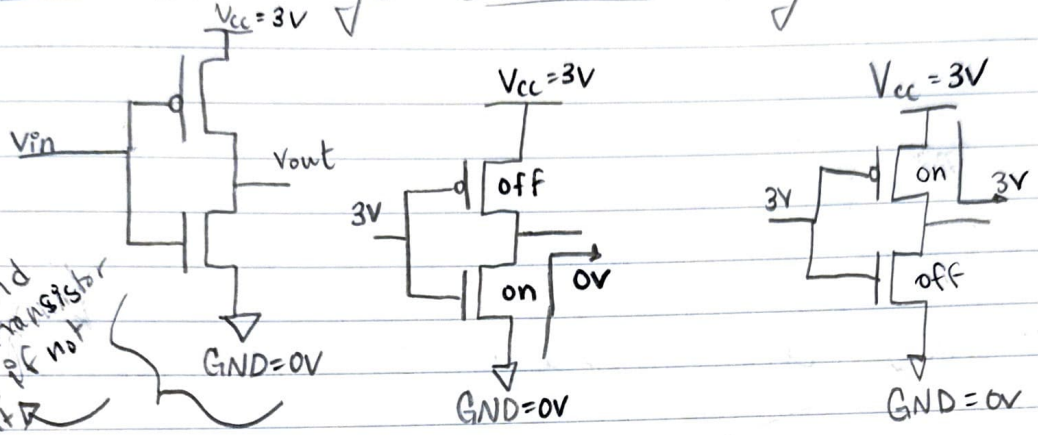
n-type field-effect transistor



p-type FET transistor



5.2.1 Building an Inverter Using FET's

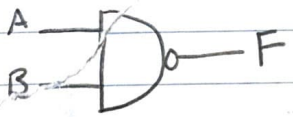


2nd transistor need because if not then it doesn't pull down to ground well

5.3 Other 2-Input Gates

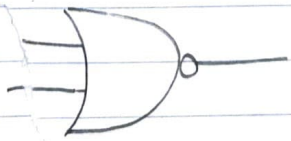
NAND (Not-AND) (anything but and)

A	B	$F = (AB)'$
0	0	1
0	1	1
1	0	1
1	1	0



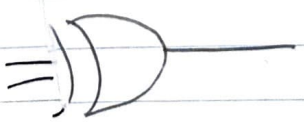
NOR (Not-or) (false when anything is true)

A	B	$F = (A+B)'$
0	0	1
0	1	0
1	0	0
1	1	0



Exclusive-OR (XOR) (anything but or)

A	B	$F = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

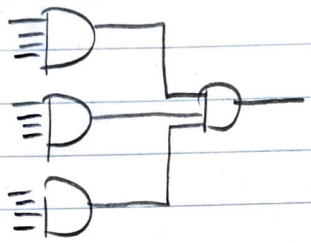


An Equivalence Gate (XNOR Gate) → NOT–XOR

A	B	F = (A == B)
0	0	1
0	1	0
1	0	0
1	1	1

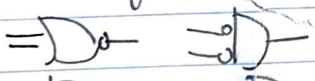
★ A == B ★

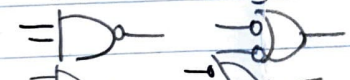
multi-input Gates

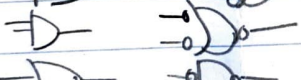


12-input AND gate

Alternate Symbols

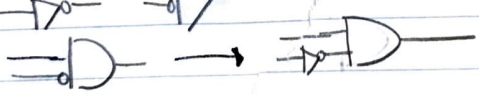
NOR : 

NAND : 

AND : 

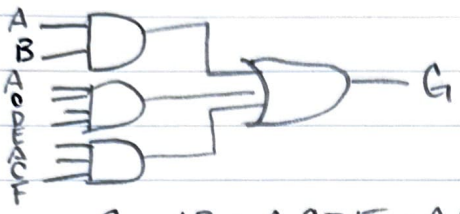
OR : 

NOT : 

mixed : 

Multi-level Logic

#1

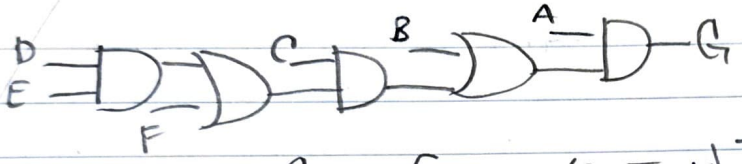


$G = AB + ACDE + ACF$

Time to react: $t_{AND4} + t_{OR3}$

Or

#2



$G = A[B + C(DE + F)]$

time to react: $t_{AND2} + t_{OR2} + t_{AND2} + t_{OR2} + t_{AND2}$

Considerations: Speed of logic → #1 is faster
Circuit area → #1 less gates
↳ #2 less transistors

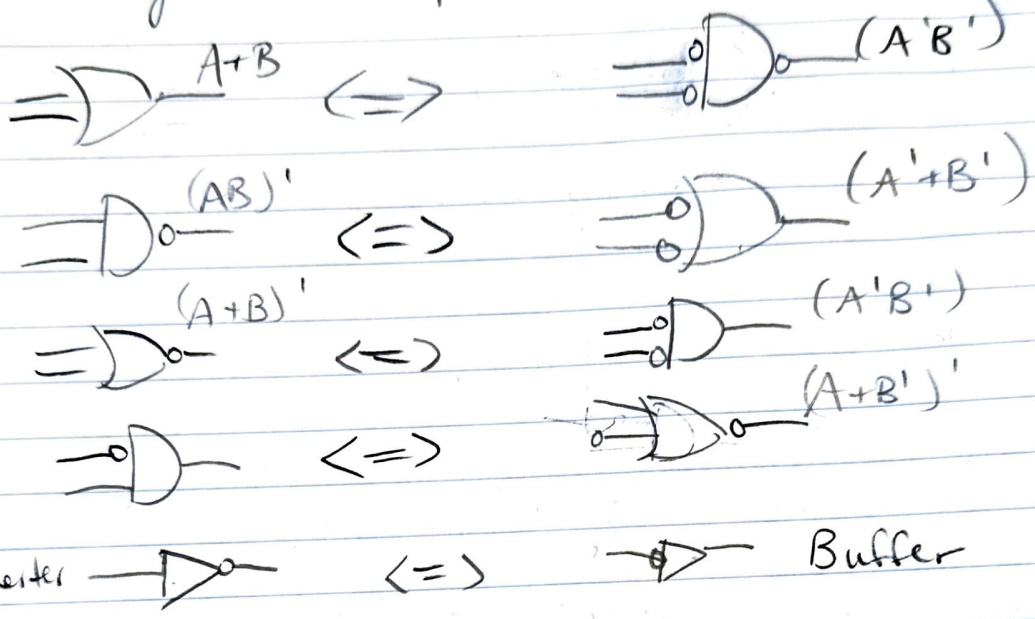
	Two-level circuit	Multi-level circuit
Level of logic	2	5
Delay	$t_{AND4} + t_{OR3}$	$3 \times t_{AND2} + 2 \times t_{OR2}$
Gate count	4	5
Gat inputs	12	10
Largest Gate	4 input	2 input

Multi-level can be converted to other, and visa versa

$AB + ACDE + ACF = A[B + C(DE + F)]$

↖ 3-level logic

DeMorgan's → Flip bubbles + Gate → same



Chapter 6

Boolean Algebra - Part II

6.1 Inverting a Function - DeMorgan's Rule

$(AB)' = A' + B'$ → Flip bubbles & Gates

$[A(1)]' = A' + 0$

$(A+0)' = A'(1)$

$[AB + C]' = [(AB)+C]' = (AB)'C' = (A'+B')C'$

ex; $[(AB+0)D + 1(B+C')']'$ → Parenthesis around AND terms ensures precedence of AND over or

$((AB+0)D)'(1(B+C'')')'$

$= ((AB+0)' + D')(0 + (B+C'))$

$= ((AB)' \cdot 1 + D')((B+C'))$

$= (A'+B'+D')(B+C')$

Start big & slowly go down into each part

6.2 Sum-of-Products and Product-of-Sums Forms

DeMorgan's → alternate way to write equations from truth tables

A	B	F	
0	0	0	← $A'B'$
0	1	1	
1	0	0	← AB'
1	1	1	

$F' = A'B' + AB'$

$F = (A+B)(A'+B)$

$[(A'B')'(AB')']'$

$[(A+B)(A'+B)]'$

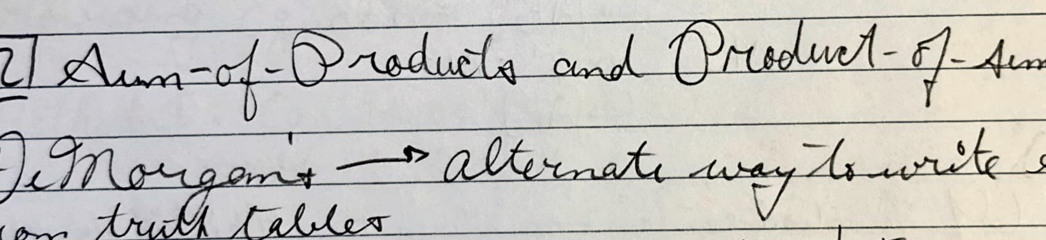
(Product of sums) → POS

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

$F = A'B + AB$

(Sum-of-Products) → SOP

NO PARENTHESIS



6.3 Canonical Forms - Minterm Expansion & Maxterm Expansion

SOP → minterm expansion

Minterm Expression

A	B	F	
0	0	1	m0
0	1	1	m1
1	0	0	m2
1	1	1	m3

$F = A'B' + A'B + AB$

$F = m0 + m1 + m3$

$F = \sum m(0,1,3)$

Maxterm Expression

A	B	F	
0	0	1	M0
0	1	1	M1
1	0	0	M2
1	1	0	M3

$F = (A'B)(A'+B')$ → DeMorgan

$F = M2 \cdot M3$

$F = \Pi M(2,3)$

To find M2, $m2' = M2$

↳ $[AB']' = A'+B$

M3, $m3' = M3$

↳ $[AB]' = A'+B'$

If function contains minterm m_2, then it does NOT contain maxterm M2.

6.4 Boolean Minimization

CMOS — complementary metal-oxide-semiconductor

ex; $A'B + AB' + AB = A'B + AB' + AB + AB$ $\quad AB = AB + AB$

$= A'B + AB + AB' + AB$ Comm. theorem

$= (A' + A)B + A(B' + B)$ Distr. theorem

$= (1)B + A(1)$ $A' + A = 1$

$= B + A$

ex; $(A'B + AB)' = (A + B')(A' + B')$ DeMorgan's

$= AA' + AB' + A'B' + B'B'$ Multiply Out

$= 0 + (A + A')B' + B'$

$= B' + B'$

$= B'$

Proving Equalities

Method 1: Construct a truth table

If equivalent, they're the same

Method 2: Convert both sides to Minterm or Maxterm

"Undo" minimization $\{AB = ABC + ABC' \quad B = A'BC' + A'BC + ABC' + ABC\}$

Write using minterms / maxterms

$AC + A'B + AC' \stackrel{?}{=} ? \; BC' + BC + AB'$

$ABC + AB'C + A'BC' + A'BC + A'BC' + ABC' = A'BC' + ABC' + A'BC + ABC + ABC' + AB'C$

$m5 + m7 + m2 + m3 + m4 + m6 = m2 + m6 + m3 + m7 + m4 + m5$

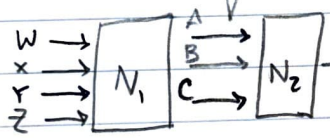
$m2 + m3 + m4 + m5 + m6 + m7 = m2 + m3 + m4 + m5 + m6 + m7$

Method 3: Manipulate One Side So that it equals the other → self-explanatory, right?

Method 4: Perform the same operation on both sides
↳ Only a few operations can be applied to both sides
↳ no OR or AND, NOT is okay

$$[A + (B'+C')(B'+C)](A'+C) \stackrel{?}{=} (A'+C)(A+B')$$
$$A'(BC+BC') + AC' \stackrel{?}{=} AC' + A'B \quad \text{(Inverted previous line)}$$
$$A'B + AC' = A'B + AC'$$

6.5 Incompletely Specified Functions and Boolean Minimization



Known: N_1 does not generate output ABC with values 101 or 111

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

minterm: $F = \sum m(0,1,3,6) + \sum d(5,7)$ → "don't cares"

maxterm: $F = M2M4D5D7$

$= \Pi M(2,4)\Pi D(5,7)$

Finding the smallest solution → plug in $\begin{matrix}0\\1\\1\end{matrix}$ into XX
↳ 7.7 produces smallest equation
↳ $F = A'B' + C + AB$

(Will learn a better way later)

HW

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

$F = A'B'C + A'BC + AB'C + ABC$

↳ $F = C$

$Z = (D+E+F)(D+E+F)F'$

$= DF' + EF' + FF'$

$= (D+E)F'$

$A=1, B=1, C=1$ $\quad F = ABC + A'BC$

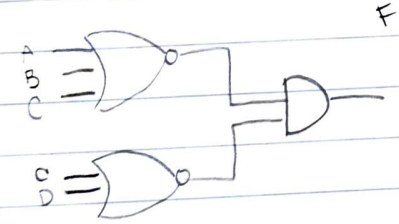
$A=0, B=1, C=1$

$Y = ABC\left[AB + C(BC+AC)\right]$

$= ABC\left[AB + BC + AC\right]$

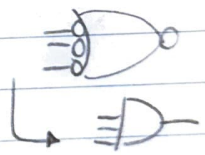
$= ABC$

$F = (A+B+C)'(C+D)'$



$X = (A'+B'+C')'$

$= ABC$



Write the minterm expansion

A	B	C	F	
0	0	0	1	m0
0	0	1	1	m1
0	1	0	0	m2
0	1	1	1	m3
1	0	0	0	m4
1	0	1	1	m5
1	1	0	1	m6
1	1	1	0	m7

$F = m_0 + m_1 + m_3 + m_5 + m_6$

$= A'B'C' + A'B'C + A'BC + AB'C + ABC'$

$= A'B'(C'+C) + C(A'B + AB') + ABC'$

$= A'B' + C($

$= A'B'C' + ABC' + A'B'C + A'BC + AB'C$

$= C'(A'B' + AB) + C(A'B' + A'B + AB')$

$= \quad + C(A'(B'+B) + AB')$

$+ C(A' + AB')$

$+ C(A+B)$

$A'B'C' + ABC' + C(A+B)$

$AC + BC$

$((A+B')+C'D)' = (A+B')'(C'D)'$

$= A'B \cdot (C+D')$

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

$F = ABC$

$= (A'+B'+C')'$

$F = (AB'+CD)'$

$= (AB')'(CD)'$

$F = (A'+B)(C'+D')$

$F = (A'B' + AB')'$

$= (A'B')'(AB')'$

$= (A+B)(A'+B)$

ex; $F = ABC' + A'B'C + A'BC'$

$= BC'(A+A') + A'B'C$

$= BC' + A'B'C$

ex; $A'B + B(A'D+C)$

$= A'B + A'BD + BC$

$= A'B(1+D) + BC$

$= A'B + BC$

$$F = A'BC' + A'B'C + A'BC + AB'C + ABC'$$

$$= A'B'(C' + C) + A'BC + AB'C + ABC'$$

$$= A'B' + A'BC + AB'C + ABC'$$

$$= A'(B' + BC) + AB'C + ABC'$$

$$= A'(B' + C) + AB'C + ABC'$$

$$= A'B' + A'C + AB'C + ABC'$$

$$= A'B' + C(A' + AB') + ABC'$$

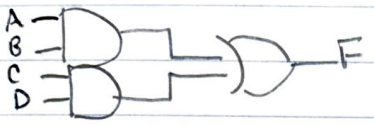
$$= A'B' + C(A' + B') + ABC'$$

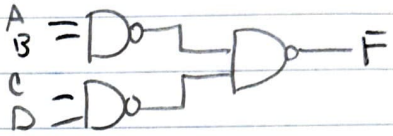
$$= A'B' + A'C + B'C + ABC'$$

Gates → Part II

$AB + CD = (AB + CD)''$

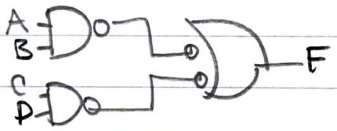
$= ((AB)'(CD)')'$



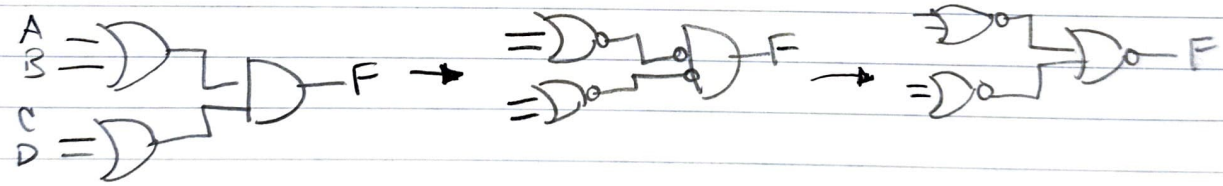


AND-OR = two levels of NAND

Preffered :

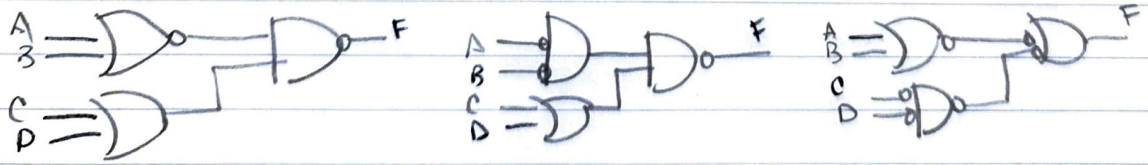


OR-AND = two levels of NOR



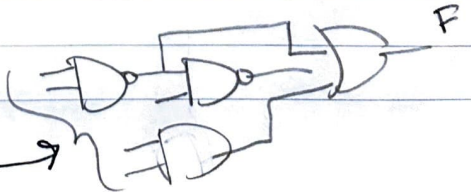
Bubble Matching

Match lines w/ no bubbles together, & bubbles w/ bubbles



* Problem
* One solution
* another solution

(Not possible when output goes to multiple inputs → Reconvergent fanout)

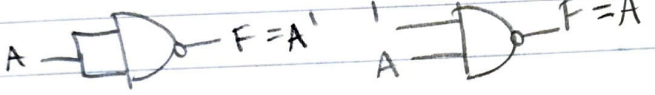


Functionally Complete Logic Sets

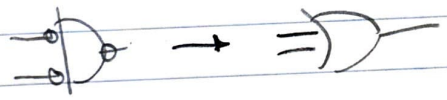
AND/OR/NOT can be used to make any truth table

NAND is also
NOR is also

NAND as NOT



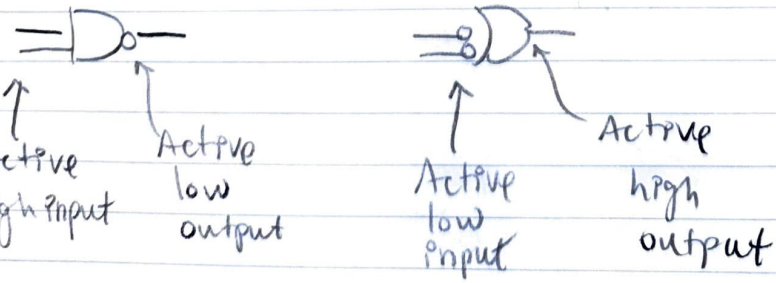
NAND as OR



NAND as AND & NOT

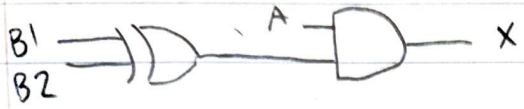
So, NAND is functionally complete

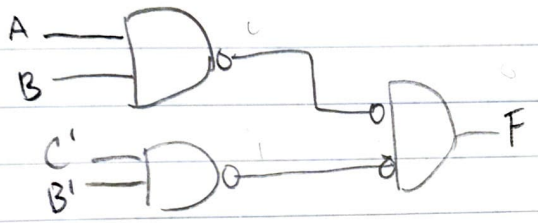
(Similarly, NOR can be written for NOT/AND/OR, and is also functionally complete)



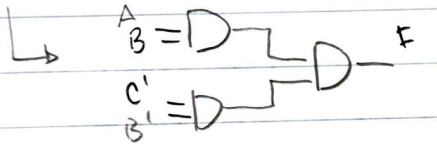
HW

$X = A(B1'B2 + B1B2')$

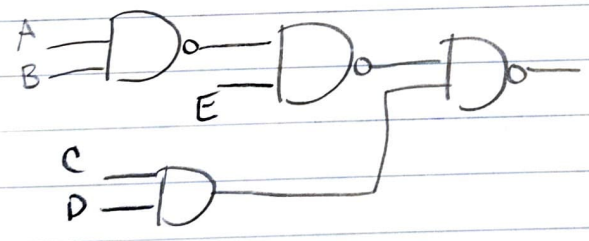




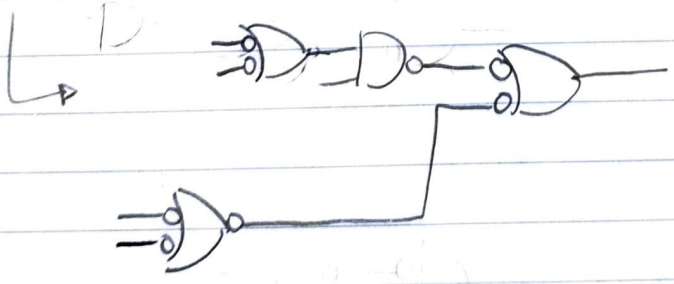
A B C	F
0 0 0	0
0 0 1	
0 1 0	
0 1 1	
1 0 0	
1 0 1	
1 1 0	
1 1 1	0



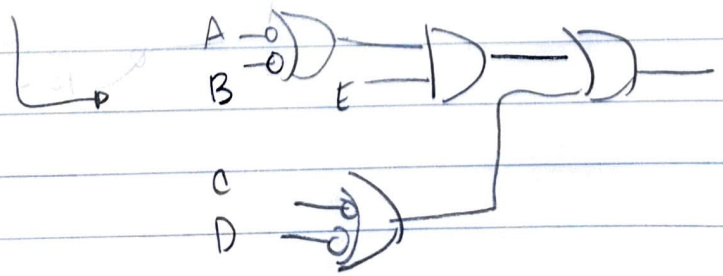
$(ABC'B') = F$



Bubble match



$(A' + B')E + (C' + D')$



$A'B'C + AB'C + ABC = F$

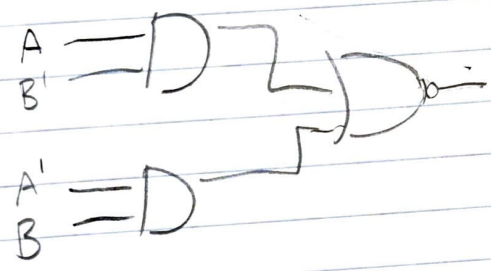
$C(A'B' + AB' + AB) = F$

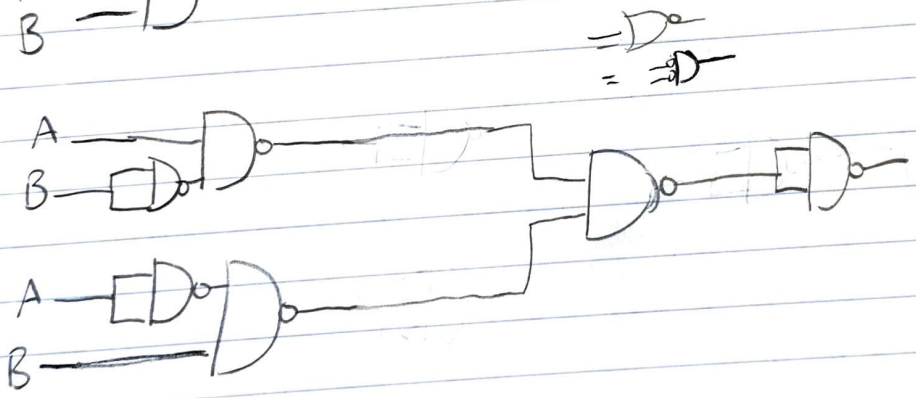
$C(A'B' + A(B' + B)) = F$

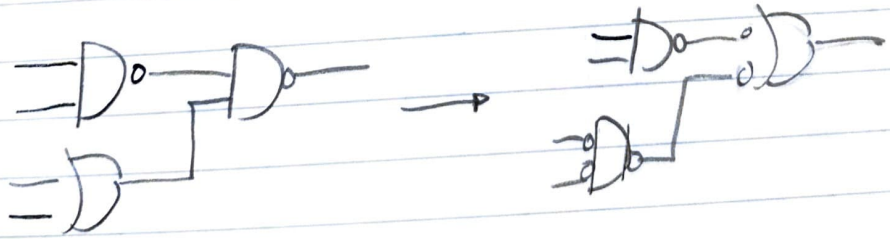
$C(A'B' + A) = F$

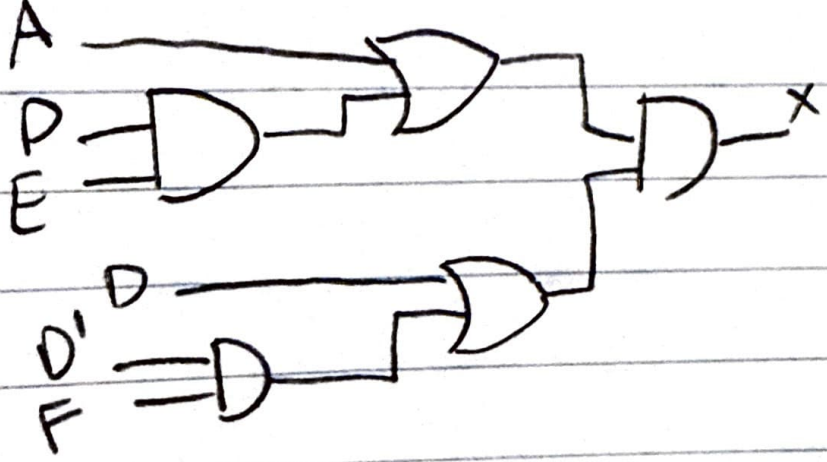
$C(B' + A) = F$

$B'C + AC = F$









$$X = (A + DE)(D + D'F)$$

$$= AD + AD'F + DDE + DED'F$$

$$= AD + AD'F + DE + \cancel{0}$$

$$= A(D + D'F) + DE$$

$$= A(F + D) + DE$$

$$= AF + AD + DE$$

$\bar{F}D' + D$

$$F = C'V'T + CV'T + CVT$$

$$= V'T(C' + C) + CVT$$

$$= V'T + CVT$$

$$= T(V' + CV)$$

$$= T(C + V')$$

$$= TC + TV'$$

$$= T(C + V')$$

Chapter 12

Karnaugh Maps (KMaps)

12.1 Truth Tables to KMaps

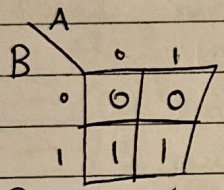
KMaps – A different form of writing truth tables

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

Normal order

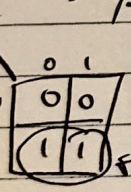
A	B	F
0	0	0
0	1	1
1	1	1
1	0	0

Gray Code Order



KMap for above truth table
- A values on top
- B values on bottom
- Results in the middle

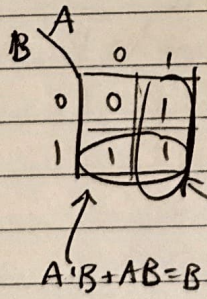
ex;



$A'B + AB = B$

ex;

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



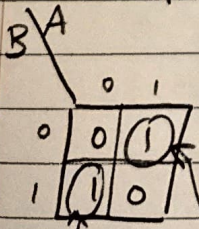
two circles, can only circle two at a time

$A'B + AB = B$

$AB' + AB = A$

$F = A + B$

ex;



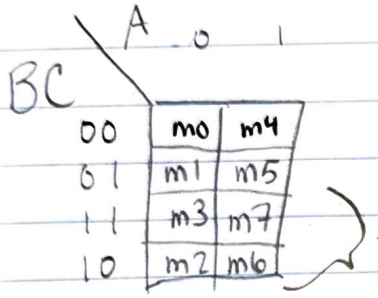
* Can't be reduced, individual circles

AB'

$A'B$

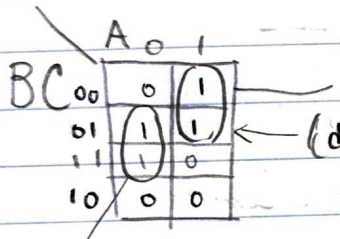
$F = A'B + AB' = A \oplus B$

12.2 Three-Variable K Maps



← Out of Order bc pt's in Gray Code Order

ex;



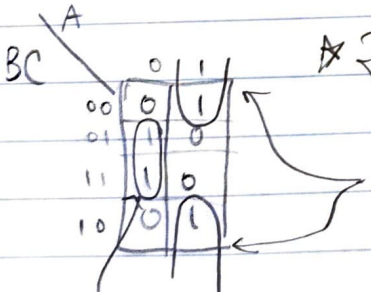
$AB'C' + AB'C = AB'$

(don't have to circle middle two bc pt's already covered)

$F = A'C + AB'$

$A'B'C + A'BC = A'C$

ex;



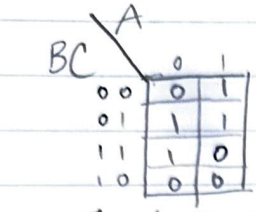
* Top & Bottom rows are adjacent in the gray code

$AB'C' + ABC' = AC'$

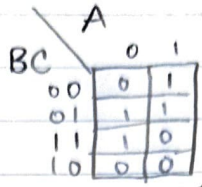
$A'B'C + A'BC = A'C$

$F = A'C + AC'$

12.3 Minterm and Maxterm Expansions and K Maps



$F = \sum m(1,3,4,5)$

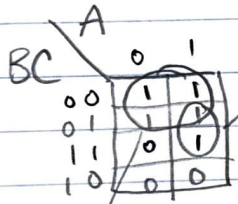


$F = \Pi M(0,2,6,7)$

12.3.1 Circling More Than Two 1's

Three Rules to K Maps
- Only adjacent 1's can be circled
- All ones must be circled
- 1's must be circled in power-of-2 groups

ex;



$AB'C + ABC = AC$

$A'B'C' + AB'C' + A'B'C + AB'C = B'$

$F = B' + AC$

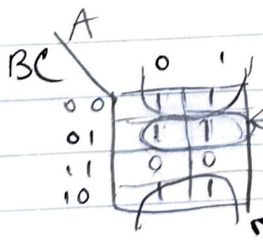
When circle 4 at once, you apply simplification theorem twice

$F = A'B'C' + AB'C' + A'B'C + AB'C$

$= B'C' + B'C$

$= B'$

ex;



$A'B'C + AB'C = B'$

$A'B'C + AB'C' + A'BC' + ABC' = C'$

$F = B' + C'$

12.4 Four-Variable K Maps

CD \ AB	00	01	11	10
00	m0	m4	m12	m8
01	m1	m5	m13	m9
11	m3	m7	m15	m11
10	m2	m6	m14	m10

ex;

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

AB'C'

D

A'BC

$F = A'BC + AB'C' + D$

ex; *The four corners are adjacent*

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

4 corners

$F = A'B'C'D' + AB'C'D' + A'B'CD' + AB'CD' = B'D'$

Plotting Min & Maxterm expansions

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

$F = \sum m(1, 3, 4, 15)$

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

$F = \prod M(3, 7, 10, 13)$

9	
10	A
11	B
12	C
13	D
14	E
15	F

2.5 Plotting a Boolean Equation on a K Map

$F = A + BC$

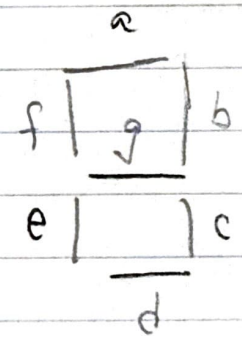
* Pretty easy, just fill in the 1's.

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

HW

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

$F = A' + D + AB'C'$



Lab 4 Part 1

A	B	C	D	X	
0	0	0	0	1	m0
0	0	0	1	1	m1
0	0	1	0	1	m2
0	0	1	1	0	m3
0	1	0	0	1	m4
0	1	0	1	1	m5
0	1	1	0	0	m6
0	1	1	1	0	m7
1	0	0	0	1	m8
1	0	0	1	1	m9
1	0	1	0	1	m10
1	0	1	1	0	m11
1	1	0	0	1	m12
1	1	0	1	1	m13
1	1	1	0	0	m14
1	1	1	1	0	m15

$$X = A'B'C'D' + A'B'C'D + A'B'CD' + A'BC'D' + A'BC'D + AB'C'D' + AB'C'D + AB'CD' + ABC'D' + ABC'D$$

$$= A'B'C'(D'+D) + B'CD'(A'+A) + A'BC'(D'+D) + AB'C'(D'+D) + ABC'(D'+D)$$

$$= A'B'C' + B'CD' + A'BC' + AB'C' + ABC'$$

$$= A'C'(B'+B) + B'CD' + AB'C' + ABC'$$

$$= A'C' + ABC' + B'CD' + AB'C'$$

$$= C' + B'CD'$$

$$= C' + B'D'$$

Rule 12D

$XY' + Y = X + Y$

$XY + Y' = X + Y'$

1000 0000
0010 1010

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

$X = A'C' + ABC' + B'D' + AB'C$

$C' + B'D'$

$1.1(6.242 \times 10^{18}) \cdot 5 = 3.4331 \times 10^{19}$

xY	xY' + Y	x + Y
00	0	0
01		1
10		1
11		1

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

$F = C'D + A'BCD'$

$F = A + B'$

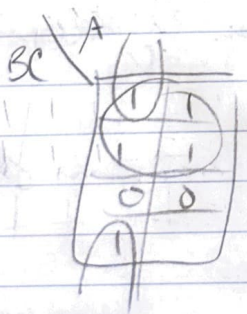
B \ A	0	1
0	1	1
1	0	1

$F = A + B'C$

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

$F = AB + C'D$

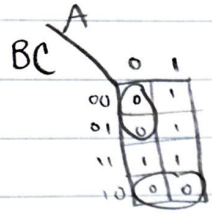
CD \ AB	00	01	11	10
00	0	0	1	0
01	1	1	1	1
11	0	0	1	0
10	0	0	1	0



$F = B' + A'C'$

* Better to do bigger groups, even if overlap
  → less variables

12.6 Deriving Product of Sum Expressions from K Maps

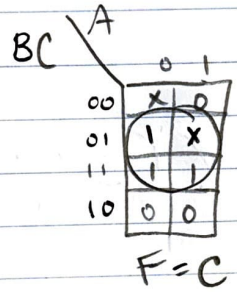


$F' = A'B' + BC'$

$F = (A+B)(B'+C)$ * DeMorgan's

12.7 Solving a K Map With Don't Cares

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



F = C

Circle them when they help, ignore them when they don't

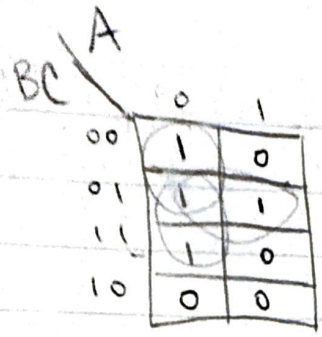
12.8 Finding Optimal Solutions Using K Maps

Implicant → circled group of 2^x

Prime Implicant → largest group possible

essential prime implicant → must be used in any minimum solution

Matt Rucker & Isabel Mooney



$F = A'B' + A'C + B'C$

Xor

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

$$G = X'Y'Z' + XYZ + XY'Z' + X'YZ' + X'Y'Z + XYZ'$$

$$= XY + X'Y'(Z'X + ZX')$$

$$= XY + X'Y'(ZX + Z'X')$$

$$= XY + ZX + Z'X'$$

$$Y = ABC + AC' + AB' + A'B'C$$

$$((ABC)(AC')(AB')(A'B'C))'$$

$$= ((A'+B'+C')(A'+C)(A'+B)(A+B+C'))'$$

$$F = ((A + B') + C'D)'$$

$$(A+B')(C'D)$$

$$= (A'B)(C+D')$$

Chapter 10

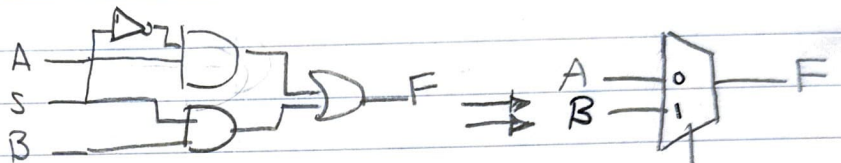
Higher Level Building Blocks: Multiplexers, Decoders, and Lookup Tables

Multiplexer: (Mux)

When $S = 0 \rightarrow F = A$

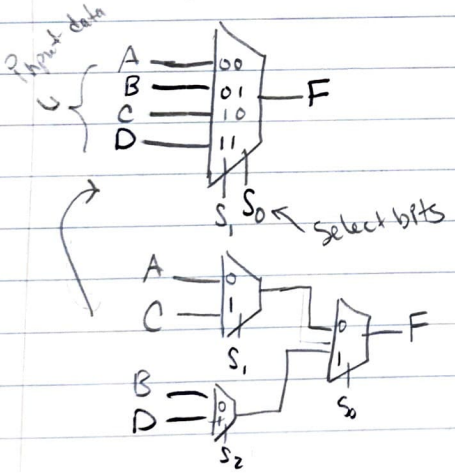
$S = 1 \rightarrow F = B$

S	A	B	F
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



$$F = S'A + SB$$

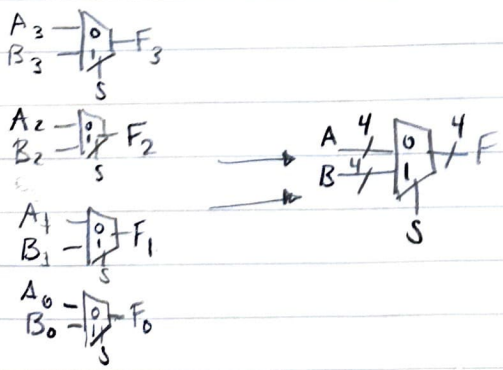
4:1 Multiplexer:



S_1	S_0	F
0	0	A
0	1	B
1	0	C
1	1	D

$$F = S_1'S_0'A + S_1'S_0B + S_1S_0'C + S_1S_0D$$

Multi-Bit Wires in Schematics:

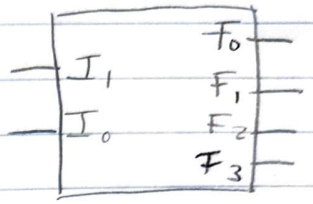


10.4 Decoders:

- Multiple Outputs → different way of writing tables

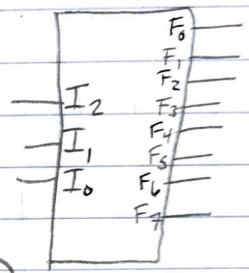
I_1	I_0	F_0	F_1	F_2	F_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

$F_0 = I_1' I_0'$
$F_1 = I_1' I_0$
$F_2 = I_1 I_0'$
$F_3 = I_1 I_0$



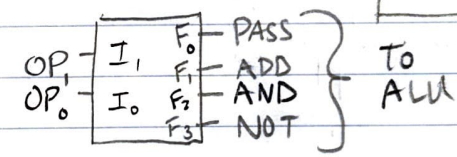
Sub on F → what it's true for
F_2 → 10

*Can be used for more bits ex; 3bits



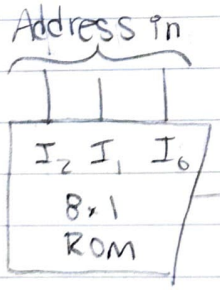
*Can be used in ALU

OP	Pass	Add	AND	NOT
00	1	0	0	0
01	0	1	0	0
10	0	0	1	0
11	0	0	0	1

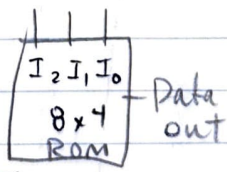


Read-only Memories (lookup tables): (ROM)

- Stores data, accessible w/ the address

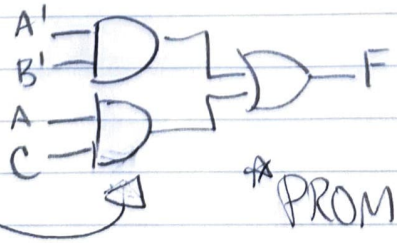


Address	Data
0	1
1	1
2	0
3	0
4	1
5	0
6	1
7	0



Address	Data
0	1101
1	1011
2	1111
3	0000
4	0111
5	1010
6	1110
7	1001

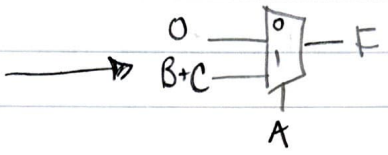
A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0



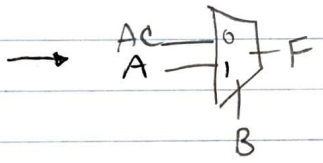
*PROM → Programmable ROM

Using Multiplexer for logic examples

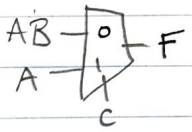
A B C	F
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	0
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1



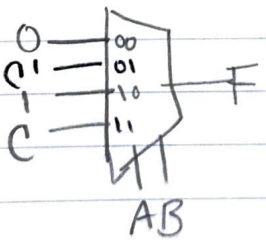
B A C	F
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	0
1 1 0	1
1 1 1	1



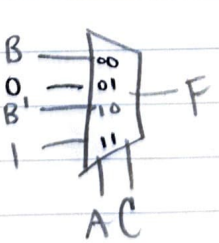
C A B	F
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	0
1 1 0	1
1 1 1	1

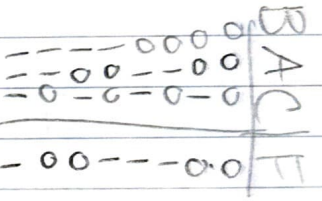


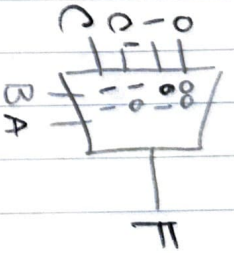
A B C	F
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	0
1 1 1	1



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



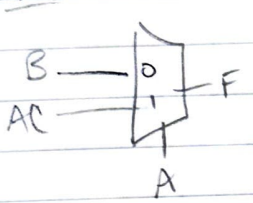




Lab 5

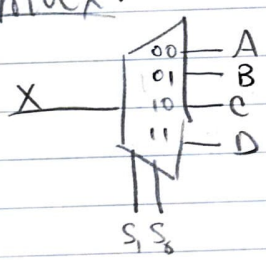
L Front = LC
L Back = LC + SL' + SR
R Front = RC
R Back = RC + SR' + SL

Class:

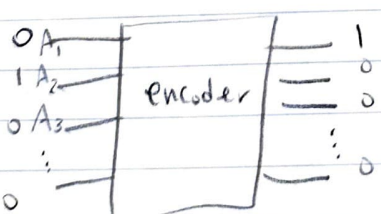


A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Demux:



Encoder:



one high at a time

N-bit output

Decimal → binary

Chapter 8: An introduction to Gate-Level Design Using SystemVerilog

Hardware description language (HDL) → code describing logic
↳ Computer-aided design (CAD)
↳ synthesizer → performs simplification, and generates description of needed gates

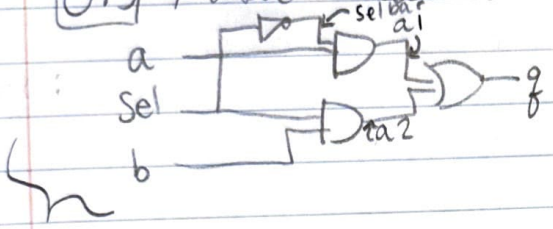
Verilog → HDL language

* HDL is not writing a computer program. You are describing a set of hardware! *

8.2 Levels of Abstraction

structural design:
```
and(q, a, b, c);
```
dataflow design:
```
assign q = a & b & c
```
behavorial verilog:
```
always_comb
  if(sel)
    q = b;
  else
    q = a;
```
→ gate & memory elements designed for you!

8.3 Basic structural System Verilog Design




```
Module mux21(
    // Define the types and directions of the ports
    output logic q,
    input logic sel, a, b
    );

    // Declare some internal signals
    logic selbar, a1, a2;

    not (selbar, sel);
    and (a1, selbar, a);
    and (a2, sel, b);
    or (q, a1, a2);

end module
```

} can be any order

Sel, a, b → logic datatypes
Selbar, a1, a2 → additional internal signals

*local wires are optional to declare
* I/O → not optional

Chapter 11: Continuing on with SystemVerilog - Hierarchial Design, Constants, and Multi-Bit Signals

Code for 4:1 MUX

```
module mux41(
        output logic q,
        input logic [1:0] sel,      ← (array)
        input logic a, b, c, d);

    logic tmp1, tmp2;

    mux21 M0 (tmp1, sel[0], a, b);
    mux21 M1 (tmp2, sel[0], c, d);
    mux21 M2 (q, sel[1], tmp1, tmp2);

endmodule
```

} 3 instances of mux21

↳ in order it was declared in mux21

or → `mux21 M0(.q(tmp1), .s(sel[0]), .a(a), .b(b));`

11.2 Specifying Constants

#bits'base Value

ex;
- 4'b0001 → binary 0001
- 4'h1 → 1 in hex
- 4'd1 → 1 in decimal

11.3 Accessing Bits of Multi-Bit wires

declare multi-bit:

```
logic [hi:lo] wireName;
```

ex;

```
logic [7:0] myByte;
logic [15:0] busA, busB, busC;
```

accessing:

ex; mybyte[7]
myNibble[0]
myNibble[2:1] → access two bits (must be hi:lo)

11.5

Compiling in command line

```
vlog mux21.sv
vlog mux41.sv
```

Chapter 3
Signed Number Representations, Negation, and Arithmetic

3.1 Addition of Unsigned Numbers

```
  ...        ...        111
  0011       0011       0011
+ 0100     + 0010     + 0111
------     ------     ------
  0111       0101       1110
No carries  w/ carries  w/ multiple carries
```

3.2 Signed Numbers: Sign-Magnitude

"Signbit"

$0011_{SM} = +(011_2) = +3$ $\quad 0 \rightarrow +$

$1011_{SM} = -(011_2) = -3$ $\quad 0 \rightarrow -$

"Sign-magnitude"

two problems: + or − 0

$0000_{SM} = +(000_2) = +0$

$1000_{SM} = -(000_2) = -0$

: adding/subtracting is complex

$5 + (-2) = 5 - 2 = 3$

$5 - (-2) = 5 + 2 = 7$

$-5 - 1 = -(5+1) = -6$

3.3 Signed Numbers: 2's Complement

$1111_{2C} = -1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = -8 + 4 + 2 + 1 = -1_{10}$

$1011_{2C} = -1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 = -8 + 2 + 1 = -5_{10}$

$0111_{2C} = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 4 + 2 + 1 = +7_{10}$

$1000_{2C} = -1 \times 2^3 = -8$

* MSB is given negative weight, all others are positive

* 2's complement ≠ sign-magnitude

ex; $1111_{SM} = -7$

$1111_{2C} = -1$

$1111_{SM} \neq 1111_{2C}$

ISABEL MOOMEY

SETH

3.3.1 Sign-extension

— adding the same bit before the sign results in the same number

ex; $0011_{2C} = 00011_{2C} = 0000000011_{2C} = 3_{10}$

$10011_{2C} = 110011_{2C} = 1111111110011_{2C} = -13_{10}$

(-16 8 4 2 1 = -13) (-32 16 8 4 2 1 = -13)

* Same for reverse *

↳ $-4_{10} = 1111100_{2C} = 100_{2C} \neq 00_{2C}$

3.3.2 Negating a 2's Complement Number

— invert all bits and add 1

$+5_{10} = 0101_{2C} \rightarrow 1010_{2C} + 1 = 1011_{2C} = -5_{10}$ (Same for − to +)

$0000_{2C} \rightarrow 1111_{2C} + 1 = 0000_{2C} = 0_{10}$

$-8_{10} = 1000_{2C} \rightarrow 0111_{2C} + 1 = 1000_{2C} = -8_{10}$???

↳ Doesn't work bc the range for 2C w/ 4 bits is $-8 \rightarrow +7$

3.3.3 Adding 2's Complement Numbers

* Same for normal binary → until there is an overflow

ex;

$$\begin{array}{r} 0011 \\ +\ 0100 \\ \hline 0111 \end{array}$$

$$\begin{array}{r} {\scriptstyle 1\ 1\ 1\ 1} \\ 1111 \\ +\ 0111 \\ \hline 0110 \end{array}$$

$$\begin{array}{r} {\scriptstyle 0\ 0\ 0\ 0} \\ 1000 \quad -8_{10} \\ +\ 0111 \quad 7_{10} \\ \hline 1111 \quad +15_{10} \end{array}$$

→ WRONG answer

* Sign extension required

$$\begin{array}{r} {\scriptstyle 0\ 0\ 0\ 0\ 0\ 0} \\ 1\ 1\ 0\ 0\ 0 \quad -8_{10} \\ +\ 0\ 0\ 1\ 1\ 1 \quad 7_{10} \\ \hline 1\ 1\ 1\ 1\ 1\ 1 \quad -1_{10} \end{array}$$

√√

SETH
IZZY

SETH
ISABEL

2's complement class practice

+5	000101
-5	111011
+20	010100
-20	101100
+31	011111
-31	100001
+32	X
-32	100000
+0	000000

SEIII + ISABEL

Chantel Mooney

BEANS + PICKLES

LEWIS & DOTTY

1.) 2C of 0010 → 1101 + 1 → $1110_{2c} = -2$

$= -2$

2.) 0000010

3.) 010101 → 101011 → $= -21$

4.) 1101 → 0011 → $= +3$

↘ NOT -5

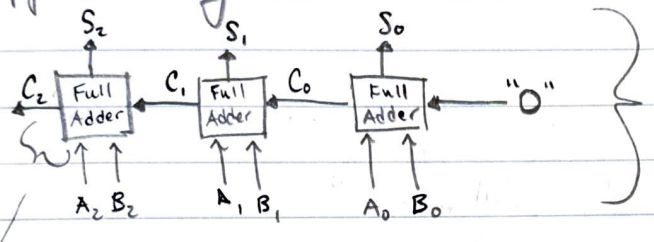
SEIII + ISABEL

Chapter 9

Gate-Level Arithmetic

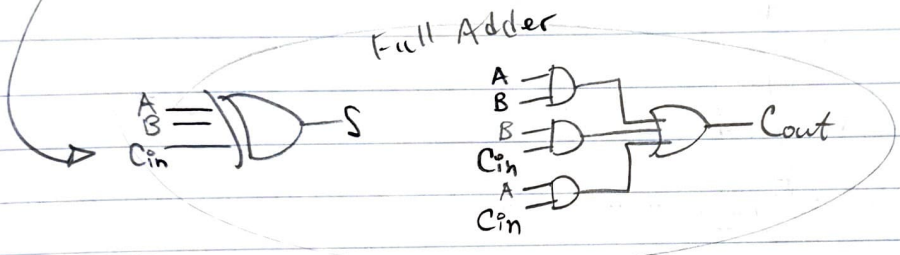
★ Everything uses 2C! ★

Ripple-carry adder



Performs:

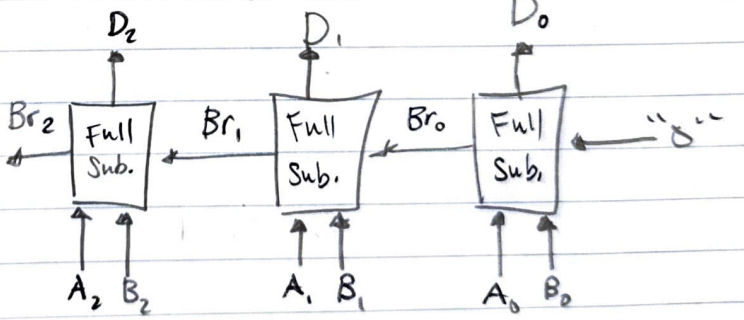
$$\begin{array}{r} A_2 A_1 A_0 \\ + \; B_2 B_1 B_0 \\ \hline S_2 S_1 S_0 \end{array}$$



9.2 Design of a Subtracter

A	B	Brin ("borrow-in")	Brout	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

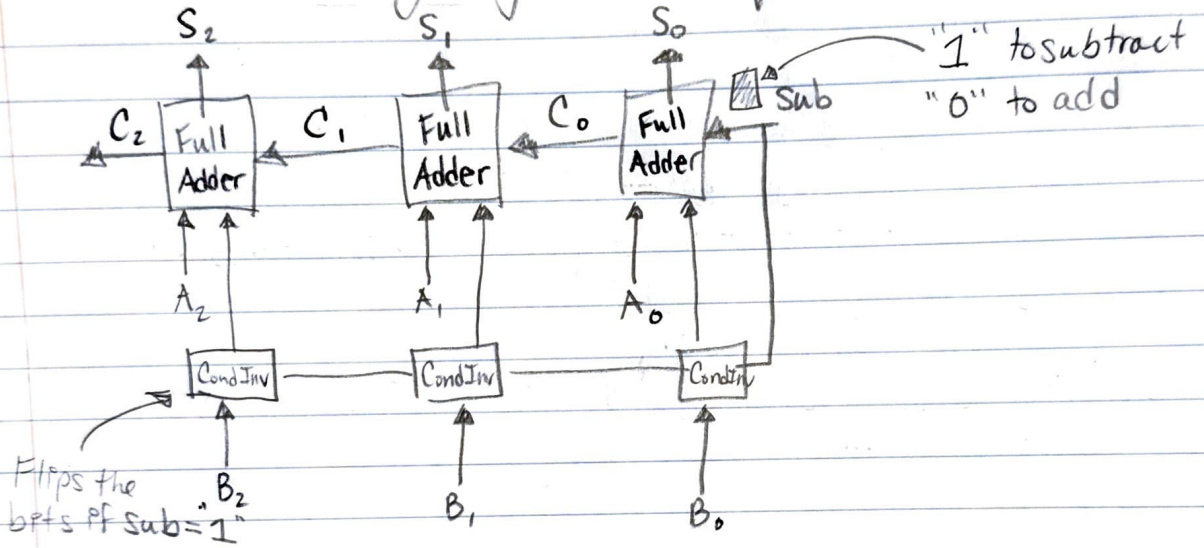
$0 - 1 - 0 \rightarrow -1 = -11_{2C}$



1101 → 0110 1101 → 1010

9.3 An Adder/Subtractor Module

* Subtraction is just $A-B = A+(-B)$
  - ↳ or negating and adding



$A'B' + AB' = Y$

$(A'+A)\cdot B' = 0$

B'

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

$AB' + A'B$

-7 0111 → 1001

+9 01001

32 16 8 4 2 1

19 +40 0 1 0 1 0 0 0

-40 Flip 1 0 1 0 1 1 1

add 1 1 0 1 1 0 0 0

+19 0 0 1 0 0 1 1

-21 1 1 0 1 0 1 1

0 0 0 0 1 0 0

0 0 1 0 1 0 0

0 0 1 0 1 0 1 +21 yay!

-3 0101

-3 1010

1011

+ 1011

1 1 1 1 0

+ 1011

1011

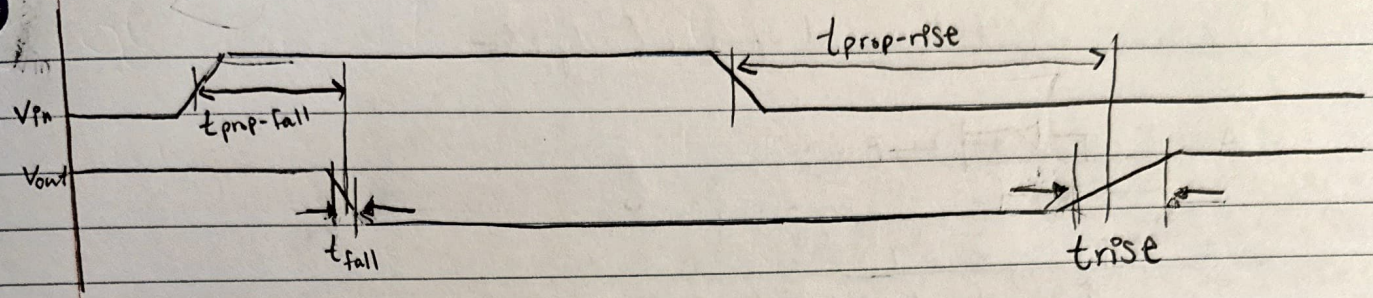
10110

01001

01010

Chapter: Gate Delays and Timing in Combinational Circuits

100 MHz > 75 MHz

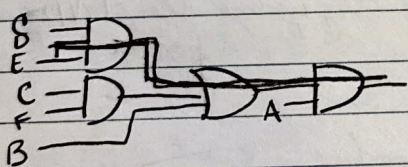


$t_{prop-fall}$ and $t_{prop-rise}$ → measured from 50% input swing to 50% output swing

? → { t_{rise} → 10% output swing to 90% of output swing
{ t_{fall} → 90% output swing to 10% output swing

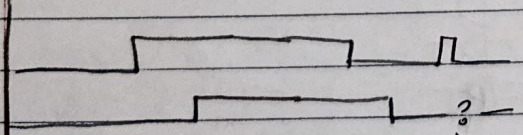
propagation delay (t_{prop})
↳ time from 50% input swing to 50% output swing
(One might not be equal to another → $t_{prop-fall} \neq t_{prop-rise}$)

13.3 Critical Path Analysis (Worst Case Analysis)



slowest path because it goes through the most gates
(And3 slower than And2)
↳ add time each gate = delay

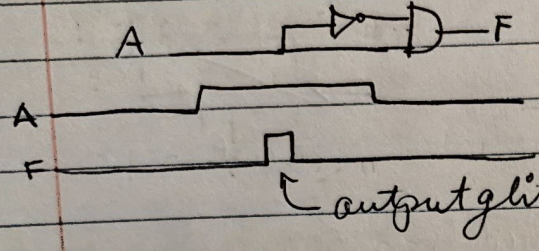
Input Glitches



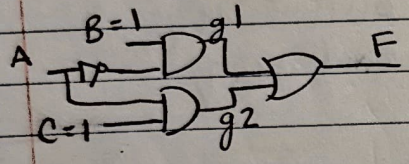
? could be Π — or Π, depending on system

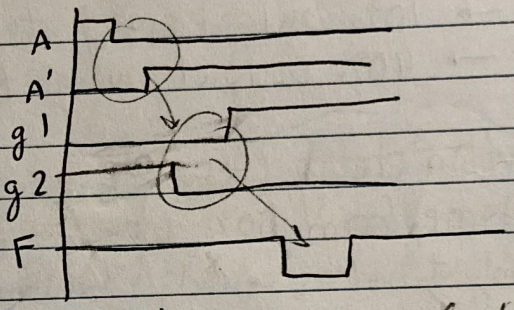
Aeth Ricks

Output Glitch and False Outputs



↳ output glitch due to the delay of the inverter





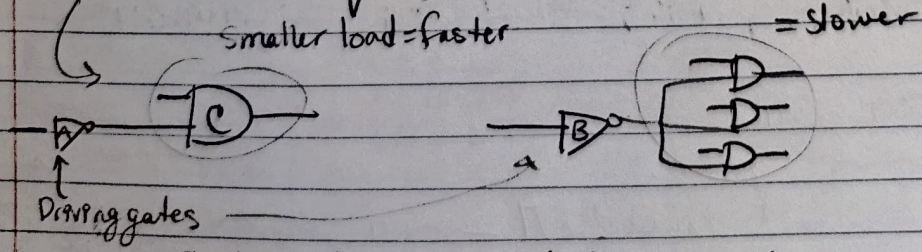
* short period of time when A = A' = 0. Causes both AND to be low.

* DO NOT try to fix this by adjusting the AND gate delay.

13.6 Gate Delay Variation

↳ not a constant value

- Temperature
- Power supply
- manufacturing variations
- how heavily loaded it is



- Takes driving gate longer when load is larger

Chapter 14
Dataflow System Verilog

14.1 A basic 2:1 MUX

```
module mux21(
    output logic q,
    input logic sel, a, b); {or}
  assign q = (~sel & a) | (sel & b);
endmodule
```

```
module mux21(
    output logic q,
    input logic sel, a, b);
  assign q = sel ? b : a;
endmodule
```

14.2 Dataflow Operators → table in book pg. 140

14.2.2 Reduction Operators

```
  assign z = &x;
(same as) assign z = x[2] & x[1] & x[0];
```

z → 1 bit
x → 3 bit

```
  assign z = ~|y;
(same as) assign z = ~(y[2] | y[1] | y[0]);
```

14.2.3 Concatenation and Replication Operators

→ 2 or more wires {} → wider bit

```
logic [3:0] x, y;
logic [7:0] z, q, w, t;
assign x = 4'b1100;
assign y = 4'b0101;
assign z = {x, x};   // {2{x}}     // z is 8'b11001100
assign q = {2'b11, y, 2'b00}      // q is 8'b11010100
```

```
logic [1:0] a;
logic [7:0] b;
assign a = 2'b01;
assign b = {{2{a}}, {2{2'b00}}};   // b is 8'b01010000
```

12.2.4 Operator Precedence

Higher precedence

* unary −, unary +, !, ~
* * / %
* + −
* << >>
* < <= > >=
* == !=
* & ~&
* ^ ~^
* | ~|
* &&
* ||
* ?:

Lower Precendence

14.2.5 Matching Wire Widths

```
logic [3:0] a;
logic [2:0] b;
logic [3:0] q;
assign q = a&b ; // Error → a & b not same width
```

* another way for 2:1 MUX →

```
assign q = ({4{~sel}} & a | ({4{sel}} & b);
```

14.3 Example - a 2:4 Decoder

```
module decode24(
    input logic [1:0] a,
    output logic [3:0] q);
assign q = (a==2'b00)? 4'b0001:
           (a==2'b01)? 4'b0010:
           (a==2'b10)? 4'b0100:
           4'b1000;
endmodule
```

{or}

```
module decode24(
    input logic [1:0] a,
    output logic [3:0] q);
assign q = (4'b0001 << a);
endmodule
```

14.4 Parameterization

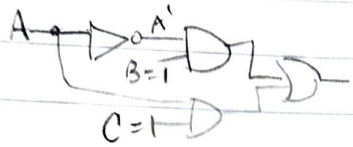
optional

```
mode muxN1n #(parameter WID=16)(
    output logic [WID-1:0] q,
    input logic sel,
    input logic [WID-1:0] a, b);
  assign q = sel? b:a;
endmodule
```

→
```
muxN1n #(3) M0(q, Sel, a, b); // 3-bit version
muxN1n #(4) M1(q, Sel, a, b); // 4-bit version
```

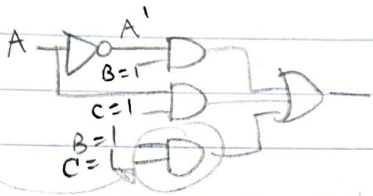
Logic Hazards

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



$X = A'B + AC$

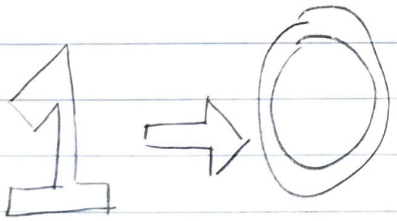
BC \ A	0	1
00	0	0
01	0	1
11	1	1
10	1	0



$X = A'B + AC + BC$

Stays on even during A transitions

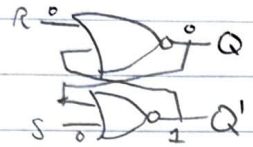
* Make extra circles to fix logic hazards



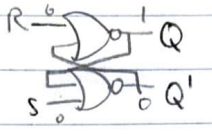
Chapter 15
Latches and Flip Flops

Sequential circuit → ability to remember what part of the process you are in

15.1 Bistability and Storage: The SR Latch



reset state



Set state

to set latch, $S=1 \rightarrow Q'=0 \rightarrow Q=1 \rightarrow S=0$

to reset latch, $R=1 \rightarrow Q=0 \rightarrow Q'=1 \rightarrow R=0$

(useless state → $S=R=1$)

$$Q^+ = S + R' \cdot Q$$

↳ Q will turn to one if $S=1$ or ($R=0$ & $Q=1$)

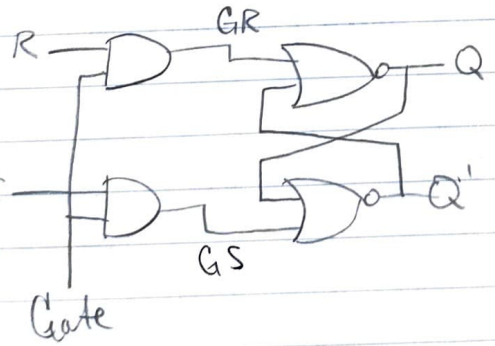
$SR=00 \rightarrow Q^+ \rightarrow$ no change (Gate holds last inputted value)

$SR=01 \rightarrow Q^+ \rightarrow 0$

$SR=10 \rightarrow Q^+ \rightarrow 1$

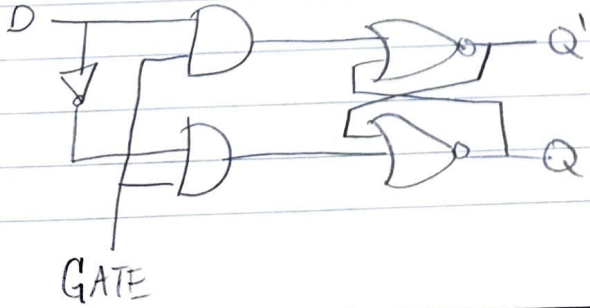
$SR=11 \rightarrow$ N/A

15.2 The Gated Latch

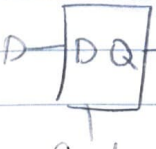


Gated D latch

* Only allows inputs when Gate = 1

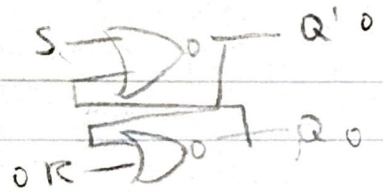


* When Gate = 1, Q = D *



already set | set latch

$$Q^+ = Gate' \cdot Q + Gate \cdot D$$

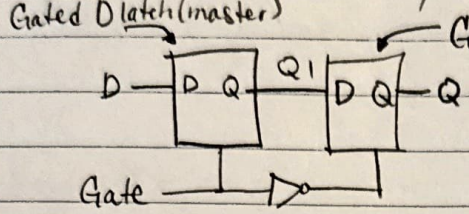


S	R	Q	Q+	Q'+
0	1→0	0	0	1
1→0	0	1	1	0
0	0→1	0	0	1
0	0→1	1	0	1
0→1	0	0	1	0
0→1	0	1	1	0
0→1	0→1	0	NA	NA
0→1	0→1	1	NA	NA

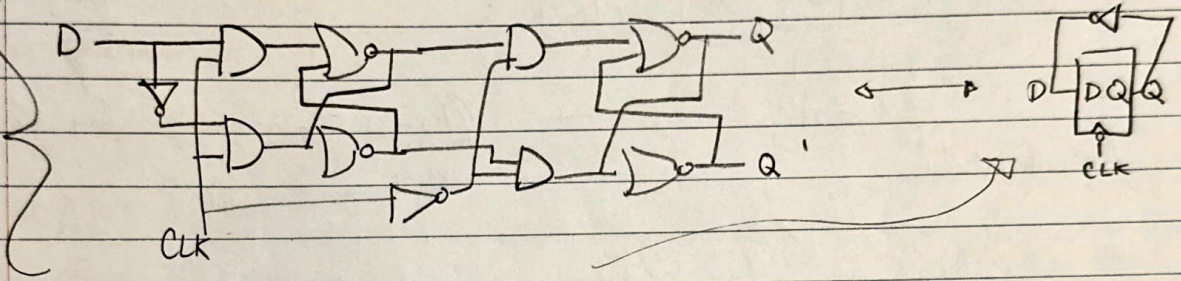
0 0

15.3 - 15.6
17.1 - 17.2

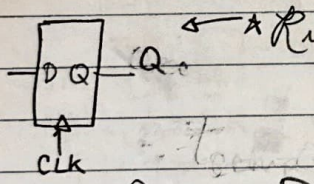
15.3 The Master/Slave Flip Flop



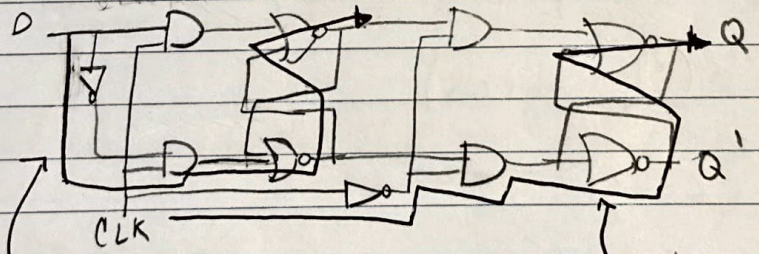
* When Gate = 1, master loaded
  - = 0, slave loaded to output



* falling edge flip-flop *
  - each CLK 1→0 results in One change output



* Rising edge flip-flop *



$t_{setup} = t_{NOT} + t_{AND} + 2t_{NOR}$

$t_{CLK-Q} = t_{NOT} + t_{AND} + 2t_{NOR}$

* t_{setup} has to be fast enough to get loaded *

"Metastable" → when the output is undetermined because of a timing error
  - can be for ANY amount of time!

Chapter 17
Behavioral SystemVerilog for Registers

17.2 The always_ff Block
↳ sequential circuits (flip-flops/registers)

```
module behavLoadableReg(
    output logic q,
    input logic clk, load, d);

always_ff @(posedge clk)
    if (load)
        q <= d;
endmodule
```

Change to negedge for falling edge flip flops

* If load is false, then nothing happens

```
// Variable-Width Loadable Register
module behavLoadableReg #(parameter WID=4)(
    output logic [WID-1:0] q,
    input logic clk, load,
    input logic [WID-1:0] d);

always_ff @(posedge clk)
    if (load)
        q <= d;
```

```
// Clearable/Loadable Register
always_ff @(posedge clk)
    if (clr)
        q <= 0;
    else if (load)
        q <= d;
```

```
//
always_ff @(posedge clk)
    if (clr && !load)
        q <= 0;
    else if (load && !clr)
        q <= d;
```

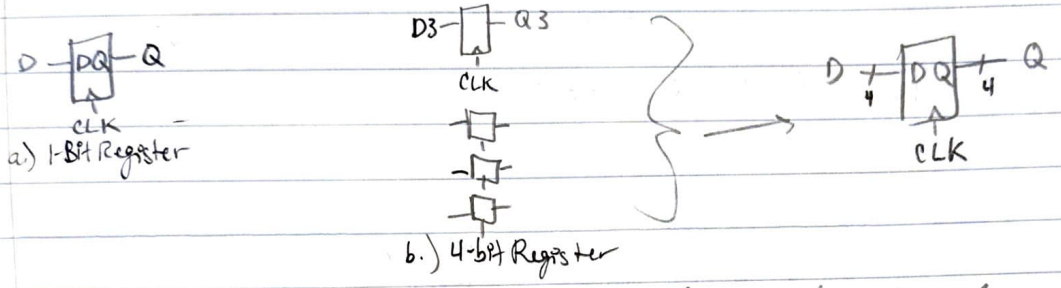
Ch 16
17.3-17.6

Chapter 16

Registers & RTL-Based Design

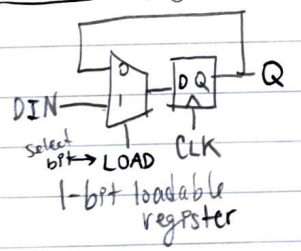
Connection of flip-flops - register
RTL - Register Transfer Level design

16.1 Flip-Flop-Based Registers



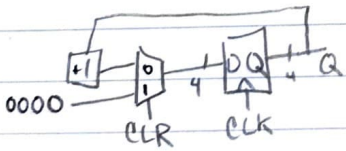
* NO logic is to be done on the clock signal - causes glitches and delays

16.1.2: Loadable Registers

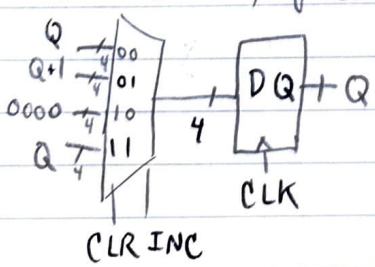


can:
- load same value (no operation)
- zero (clear)
- data input (load)
- any other desired input

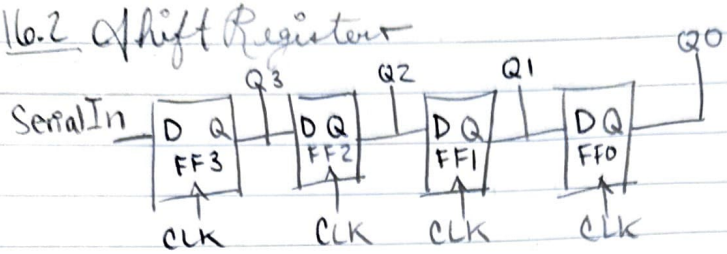
ex; 4-bit clearable counter



ex; 4-bit clearable, up counter



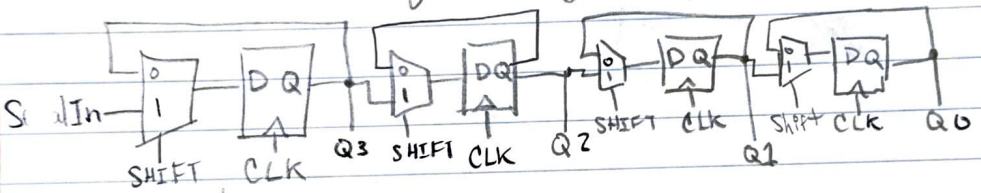
16.2 Shift Register



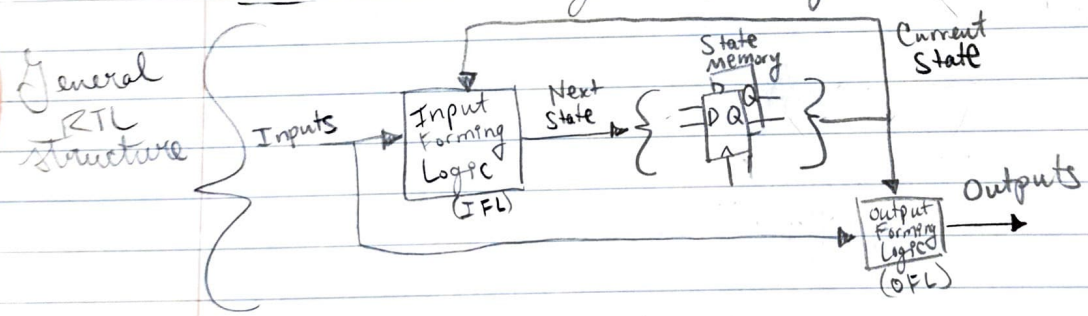
ex; If current value is 1011 and S=0, (1 per clock cycle) ← transition

1011 → 0101 → 0010 → 0001 → 0000

Conditional Shift Register



1.4 Intro to Register Transfer Level (RTL) Design



RTL → the flow of signals between registers

IFL ex; → Loadble Register

↳ nextState = (LOAD == 1) ? DIN : Q

↳ Shift Register

↳ nextState = {SerialIN, Q[3:1]}

★ Circuits can be made of multiple register/IFL/OFL combinations, such as a counter or an average circuit (page 179) ★

7.3 Shift Register Design w/ Sys. Verilog

```
module delay4 (
        input logic clk, SerialIn,
        output logic[3:0] Q);

  always_ff @(posedge clk)
  begin
     Q[3] <= SerialIn
     Q[2] <= Q[3];
     Q[1] <= Q[2];
     Q[0] <= Q[1];
  end
endmodule
```

↳ (simplified)

```
always_ff @(posedge clk)
   Q <= {SerialIn, Q[3:1]};
```

why no "begin" ?

"<=" ⟷ store into

↳ (Add a shifter)

```
always_ff @(posedge clk)
   if(shift)
      Q <= {SerialIn, Q[3:1]};
```

7.4 Semantics of always_ff Block

"<=" assignments are done ALL AT ONCE, NOT SEQUENTIALLY

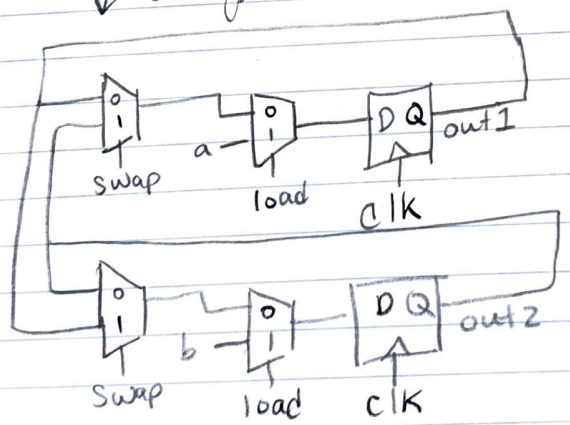
- Don't need an intermediate variable to swap like in C

```
//C code
tmp = a;
a = b;
b = tmp;
```

ex; A swapper Circuit

```
module Swapper (
        input logic clk, load, swap,
        input logic [3:0] ina, inb,
        output logic [3:0] a, b);
always_ff @(posedge clk)
    if (load)
    begin
        a <= ina;
        b <= inb;
    end
    else if (swap)
    begin
        a <= b;
        b <= a;
    end
endmodule
```

↓ Implemented Design



17.5 Reset Problems with Registers

- "logic" type contains "0", "1", "X", "Z"

X → unknown value
↳ when a flip flop does not have a value yet, it is X

BAD ex;

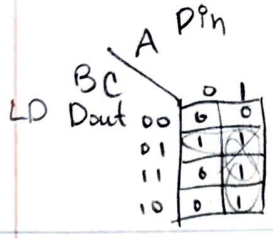
```
always_ff@(posedge clk)          // XXXX+0001 = XXXX
  q <= q+1;                        uh-oh!
```

FIXED ex;

```
always_ff@(posedge clk)
  if (reset)
    q <= 0;
  else
    q <= q+1;
```

FIXED ex;

```
always_ff@(posedge clk)
  if (load)
    q <= dataIn;
  else
    q <= q+1;
```

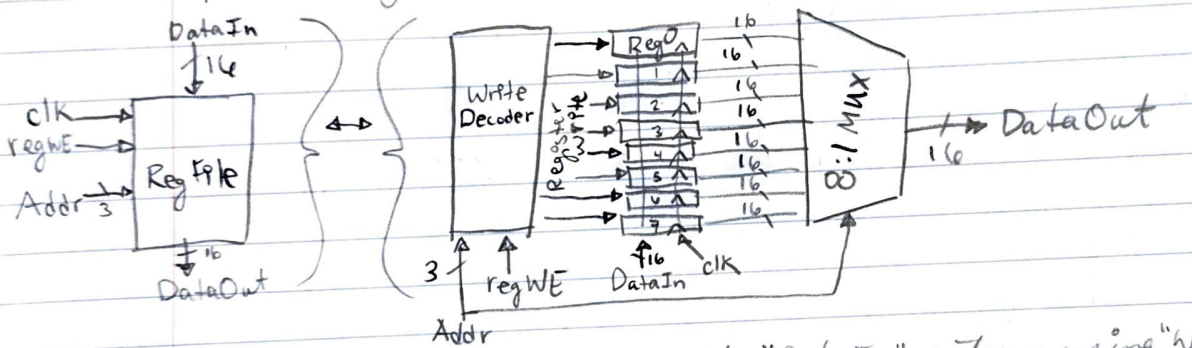


$$Dout+ = (LD)'(Dout) + (Din)(\cancel{Dout}) + (Din)(LD)$$
$$= (LD)'(Dout) + (Din)(LD)$$

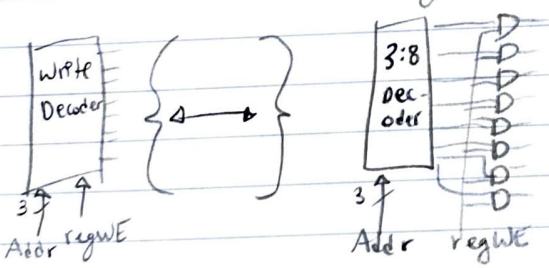
Chapter 19 : Memories

19.1 Small Memories → array of registers
Large " → transistor level

19.2 Register Files



☆ Choose which register the "DataIn" enters using "Write Decoder", and which register to read using "8:1 mux."



☆ Write Decoder is just 3:8 Decoder w/ and gates on the outputs

☆ Asynchronous → doesn't need a clock (mux above)
Synchronous → needs clock (Registers above)

take 340 before intern

19.3 Register File Design Using Behavioral System Verilog

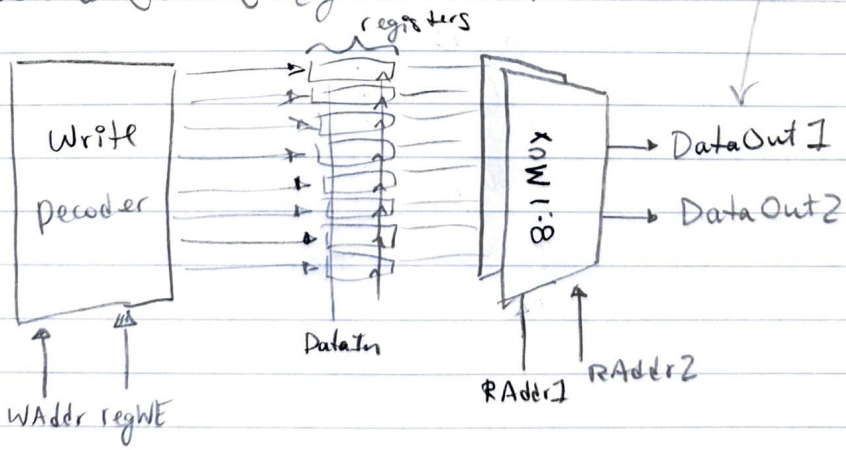
```
module regFile(
    input logic clk, regWE,
    input logic[2:0] Addr,    // To make Multi-ported use. WAddr, RAddr1, RAdd2
    input logic [15:0] DataIn,
    output logic [15:0] DataOut); // Add another DataOut for Multi ported

// An 8-element array, each location holding 16 bits
    logic [15:0] registers [8];

// The asynchronous read logic is a stand-alone dataflow statement
    assign DataOut = registers [Addr]; // Add another for another mux

// Synchronous write logic
    always_ff @(posedge clk)
      if (regWE)
        registers[Addr] <= DataIn;
endmodule
```

19.4 Multi-Ported Register Files



* Only difference is that there are two MUXes, one Write command, and 2 read commands

19.6 Larger Memories
- Transistor-level blocks can be already found
- can also write your own, and verilog will pick one for you

19.7 Read-Only Memories (ROM)
- hardware implementation of a look-up table

```
// Verilog for ROM
module myROM(
        input logic [2:0] Addr,
        output logic [3:0] DataOut);

  always_comb
  begin
    DataOut = 0; // Default
    case (Addr)
      0 : DataOut = 5;
      2 : DataOut = 9;
      3 : DataOut = 10;
      4 : DataOut = 2;
      5 : DataOut = 11;
    endcase
  end
endmodule
```

```
// Verilog for ROM ⟶ read ROM from a file
module dual-port-rom(
    input logic clk
    input logic [7:0] addr,
    output logic [15:0] q );

  // Declare ROM as an array
  logic [15:0] rom [256];
  initial // Read ROM contents from a file
  begin
    $readmemb("rom_init.txt", rom);
  end

  // Access the ROM
  assign q = rom[addr];
endmodule
```

```
//rom_init.txt
0101_1111 // location 0
0010_1001 // location 2
   ...       ...
0000_1111 // location 12
```

19.9 Multi-ported Register Files w/ Bypass

↳ "write-before-read"

```
logic [15:0] registers [8];

assign DataOut1 = (regWE == 1 && WAdder == RAdder1 ? DataIn :
                                    registers[Raddr1]);
assign DataOut2 = (regWE == 1 && WAdder = RAddr2) ? DataIn:
                                    registers[Raddr2]);

always_ff @ (posedge clk)
    if (regWE)
        registers[WAddr] <= DataIn;
```

Chapter 20:
Simple Sequential Circuits: Counters

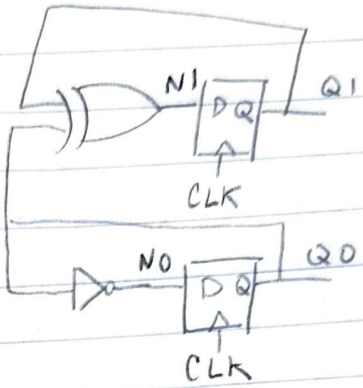
Two-bit binary counter

Transition table:

Q1	Q0	N1	N0
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

$N1 = Q1 \oplus Q0$

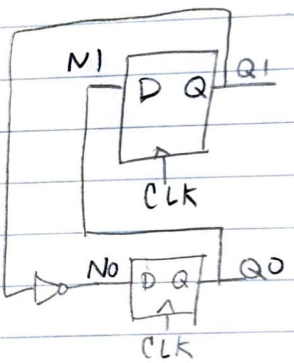
$N0 = Q0'$



A Two-Bit Gray Code Counter:

Counting in gray code $00 \rightarrow 01 \rightarrow 11 \rightarrow 10 \rightarrow 00 \rightarrow \ldots$

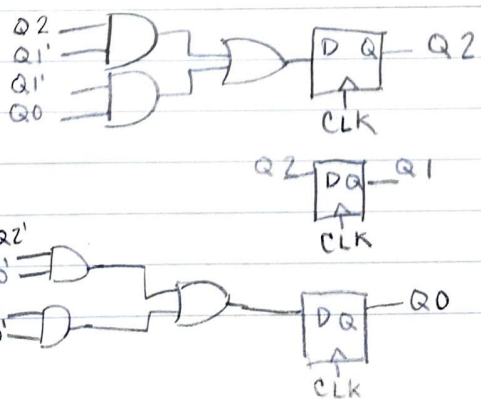
Q1	Q0	N1	N0
0	0	0	1
0	1	1	1
1	1	1	0
1	0	0	0



A Counter Example w/ Incomplete Transition Table:

Counting: $000 \rightarrow 001 \rightarrow 100 \rightarrow 110 \rightarrow 000 \rightarrow \ldots$

Q2	Q1	Q0	N2	N1	N0
0	0	0	0	0	1
0	0	1	1	0	0
1	0	0	1	1	0
1	1	0	0	0	0
0	1	1	X	X	X
1	0	1	X	X	X
1	1	1	X	X	X



20.4 Counters with Output Signals:

Q2	Q1	Q0	Z
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Z is 1 every multiple of 3

$Z = (Q2')(Q1')(Q0') + (Q2')(Q1)(Q0) + (Q2)(Q1)(Q0')$

Q2	Q1	Q1	N2	N1	N0	Z
"	"	"	0	0	1	"
"	"	"	0	1	0	"
"	"	"	0	1	1	"
"	"	"	1	0	0	"
"	"	"	1	0	1	"
"	"	"	1	1	0	"
"	"	"	1	1	1	"
"	"	"	0	0	0	"

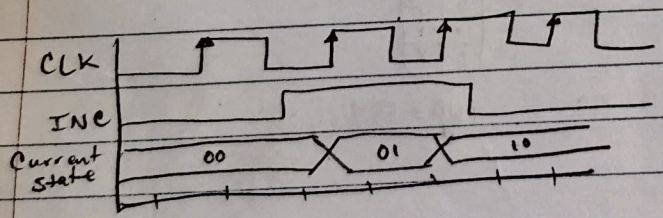
↖ IFL (Inputting Forming Logic)

20.5 Counters with Additional Inputs:

Additional "INC" signal to indicate that the counter can increment

INC	Q1	Q0	N1	N0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

← output is 1 greater than output

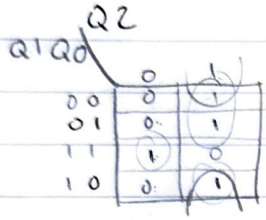


Lab 10

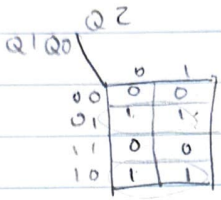
Up-Counter:

0	4
1	5
3	7
2	6

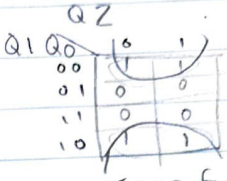
Q2	Q1	Q0	N2	N1	N0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0



Kmap for N2



Kmap for N1



Kmap for N0

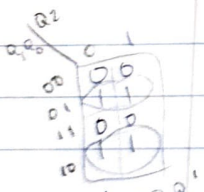
$N2 = Q2'Q1Q0 + Q2Q1' + Q2Q0'$

$N1 = Q1'Q0 + Q1Q0'$

$N0 = Q0'$

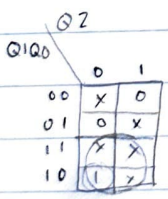
Ring Counter Design

Current State			Next State		
Q2	Q1	Q0	N2	N1	N0
0	0	0	x	x	x
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	x	x	x
1	0	0	0	0	1
1	0	1	x	x	x
1	1	0	x	x	x
1	1	1	x	x	x

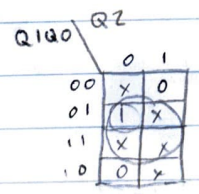


$Q1^+ = Q_1'Q_0 + Q_1Q_0'$

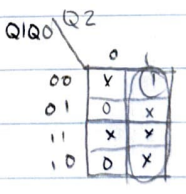
$Q_0^+ = Q_0'$



$N2 = Q1$

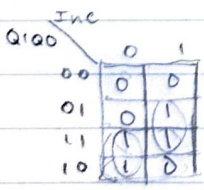


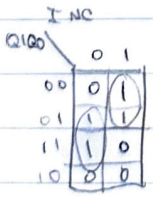
$N1 = Q0$



$N0 = Q2$

Gray Code Counter

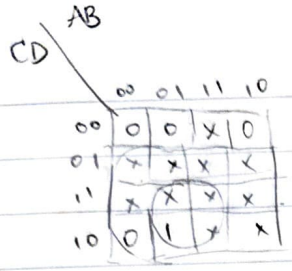




$N1 = INC'Q1 + INC\,Q0$

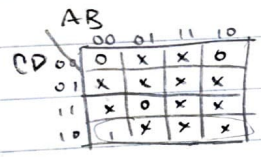
$N2 = INC'Q0 + INC\,Q1'$

Current			Next	
INC	Q1	Q0	N1	N0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	1
1	0	1	1	1
1	1	0	0	0
1	1	1	1	0

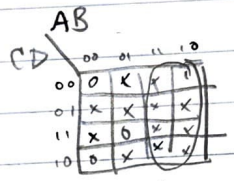


$N_1 = BC$

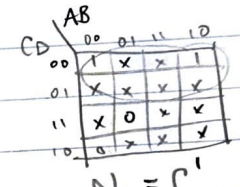
A	B	C	D	N_3	N_2	N_1	N_0
0	0	0	0	0	0	1	0
0	0	0	1	X	X	X	X
0	0	1	0	1	0	0	0
0	0	1	1	X	X	X	X
0	1	0	0	X	X	Y	X
0	1	0	1	X	X	X	X
0	1	1	0	X	X	X	X
0	1	1	1	0	0	0	0
1	0	0	0	0	1	1	1
1	0	0	1	X	X	X	X
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X



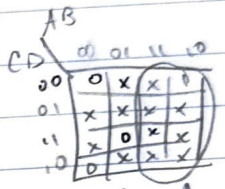
$N_3 = CD'$



$N_2 = A$

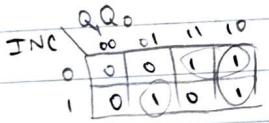


$N_1 = C'$

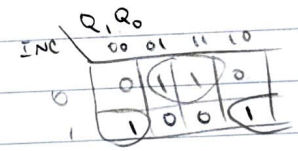


$N_0 = A$

Class

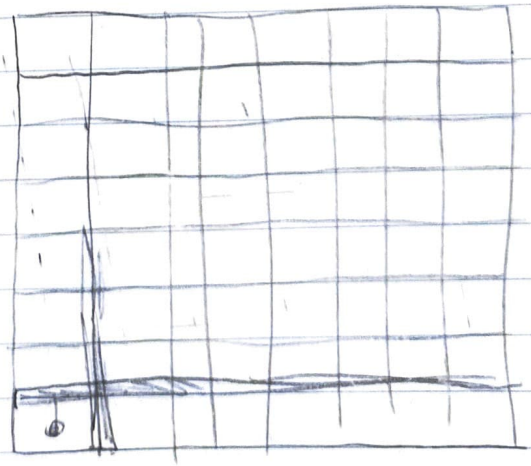


$N_1 = INCQ_1'Q_0 + INC'Q_1 + Q_1Q_0'$



$N_0 = INCQ_0' + INC'Q_0$

$= INC \otimes Q_0$



Moore / Static output - only depends on current state

20.6 Counters With Dynamic Outputs:

Dynamic / Mealy outputs - depend on both the current state as well as the external inputs

ex; Given table:

INC	Q1	Q0	N1	N0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

example Static : $Z = Q1 \cdot Q0$

example Dynamic : $Y = INC \cdot Q1 \cdot Q0$ (Q1·Q0: current state; INC: external input)

20.7 Counter Design w/ Behavioral System Verilog

```
// Simple Up Counter
always_ff @(posedge clk)
    if (clr)
        q <= 0;
    else if (ena)        // ena: enable
        q <= q+1;
```

20.7.2 // A Modulo-10 Counter With Enable

```
always_ff @(posedge clk)
    if (clr)
        q <= 0;
    else if (ena)
    begin
        if (q == 9)
            q <= 0;
        else
            q <= q+1;
    end
```

20.7.3 // 2-Bit Gray Code Counter

```
always_ff@(posedge clk)
  if (clr)
    Q <= 2'b00;
  else
    case (Q)
      2'b00: Q <= 2'b01;
      2'b01: Q <= 2'b11;
      2'b11: Q <= 2'b10;
      2'b10: Q <= 2'b00;
    endcase
```

For bigger blocks, could use a ROM

20.7.4

Q2	Q1	Q0	N2	N1	N0	Z
0	0	0	0	0	1	1
0	0	1	0	1	0	0
0	1	0	0	1	1	0
0	1	1	1	0	0	1
1	0	0	1	0	1	0
1	0	1	1	1	0	0
1	1	0	1	1	1	1
1	1	1	0	0	0	0

```
always_ff@(posedge clk)
  Q <= Q+1;
assign Z = (Q==0 || Q==3 || Q==6) ? 1:0;
```

outside always_ff block

Chapter 18
Behavioral SystemVerilog for Combinational Logic

// 2:4 Decoder

```
module combAlwaysDecoder(
        output logic [3:0] q,
        input logic[1:0]a);
  always_comb
  begin
     q[3] = (a==3);
     q[2] = (a==2);
     q[1] = (a==1);
     q[0] = (a==0);
  end
endmodule
```

// Multiple Computations

```
always_comb
begin
  t1 = (large amount of logic)
  t2 = (another large amount of logic)
  if (t1<t2)
     f = t1*t2;
  else
     f = t2-t1;
end
```

// Case statements

```
always_comb
  case (sel)
    0: q=a;
    1: q=b;
    default: q=X;
  endcase
```

→

```
always_comb
  q = X;
  case (sel)
    0: q=a;
    1: q=b;
  endcase
```

default "null" statement

18.2 The Problem w/ Latches in always_comb blocks

ex; BAD Verilog

```
always_comb
  if (ena)
    x=d|e;
```

← no "else", so if ena=0, then x holds its value in a latch

or default

ex; FIXED

```
always_comb
 begin
   x=1;
   y=1;
 if (ena)
 begin
   x=d|e;
   y=d&e;
 end
end
```

18.3 — The same latch problem w/ Case statements

ex; Bad Verilog

```
always_comb
 case (ena)
 0:
   begin
     x=d|e;
     y=d&e;
   end
   x=d;
default:
   begin
     x=1;
     y=1;
   end
endcase
```

no y → could be a latch!

ex; FIXED

```
always_comb
 begin
   x=d;
   y=0;
case (ena)
 0:
   begin
     x=d|e;
     y=d&e;
   end
 1:
   x=d;
endcase
end
```

defaults → no latches

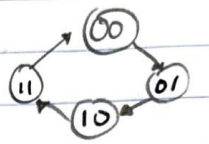
Chapter 21

State Graphs

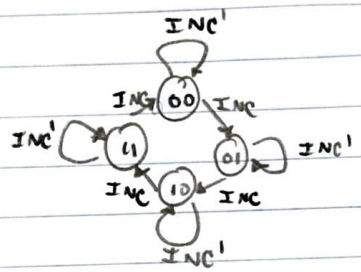
State graphs - graphical way to represent state/next behavior

ex;

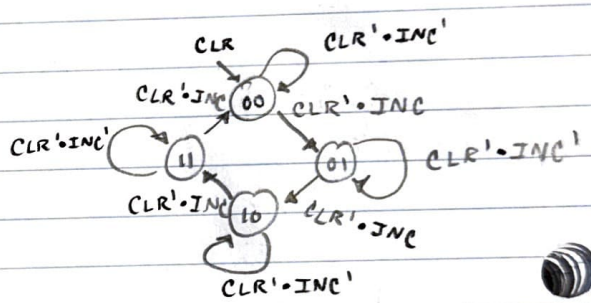
Q1	Q0	N1	N0
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0



ex; Counters w/ inputs

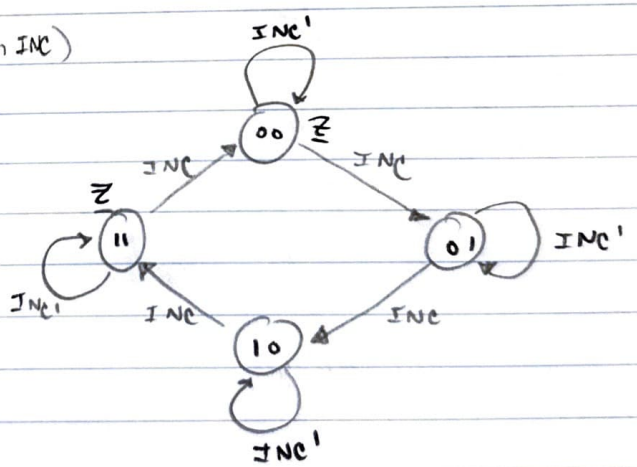


ex; Counters w/ Multiple inputs



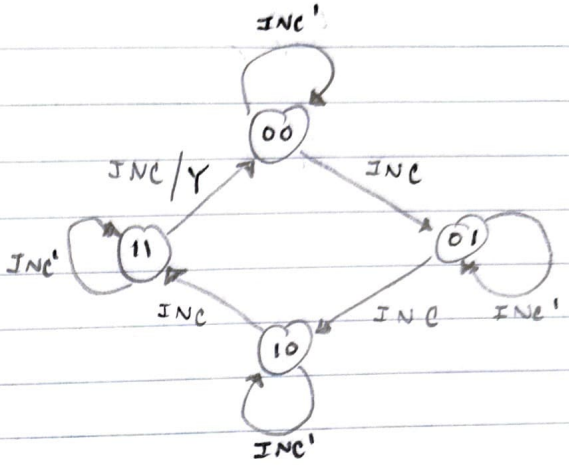
ex; More (static) Outputs (Z doesn't depend on INC)

INC	Q1	Q0	N1	N0	Z
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	0	1	1
1	0	1	1	0	0
1	1	0	1	1	0
1	1	1	0	0	1

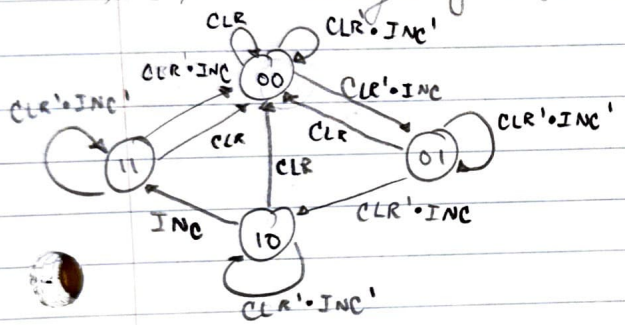


ex; Mealy (Dynamic Outputs)

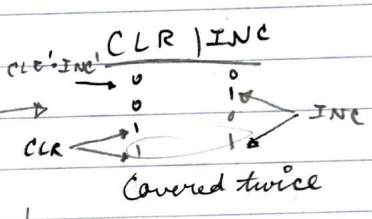
INC	Q1	Q0	N1	N0	Y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	1	0
1	1	1	0	0	1



ex; BAD state graph with conflicts

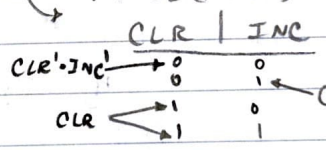


★ For 10 ★



Covered twice

FIXED (for 10)

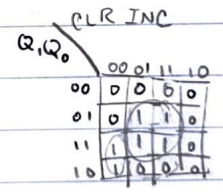


Conflicts:
★ And, Must be 0 by pairs

Completeness:
★ Or all of them, must be 1

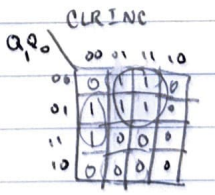
HW:

CLR	INC	Q_1	Q_0	N_1	N_0	Z
0	0	0	0	0	1	0
0	0	0	1	1	0	1
0	0	1	0	1	1	1
0	0	1	1	0	1	0
0	1	0	0	1	1	1
0	1	0	1	0	0	1
0	1	1	0	0	1	1
0	1	1	1	0	0	1
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	0	1
1	0	1	1	0	0	1
1	1	0	0	0	1	0
1	1	0	1	1	1	1
1	1	1	0	0	0	1
1	1	1	1	1	0	1



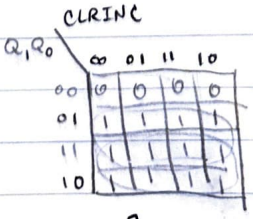
SD + Cab

$N_1 = INCQ_0 + CLR'INC'Q_1$



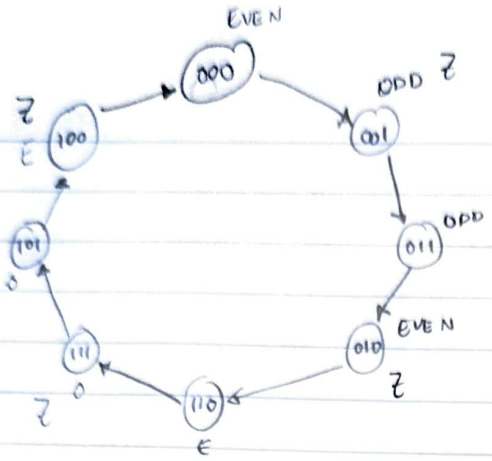
Bc + Dab

$N_0 = CLR'INC'Q_0 + INCQ_0'$



$Z = Q_0 + Q_1$

ex;

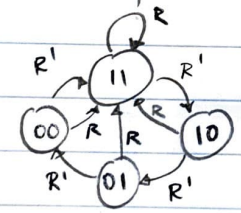


3 bit up/down counter w/ CLR, INC, DEC

order of priority →

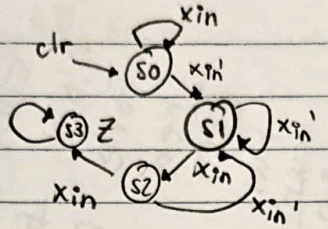
CLR	INC	DEC	$Q_2 Q_1 Q_0$
			000
			001
			010
			011
			100
			101
			110
			111

ex; 2-bit down counter, Clockwise, R input resets to 11



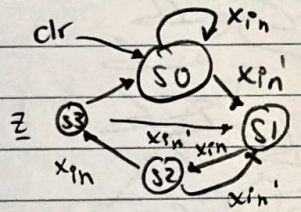
Chapter 22
Finite State Machines

22.1 A simple State Machine - A Sequence Recognizer



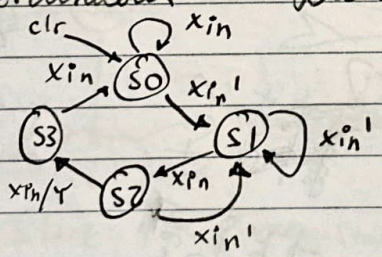
→ If the values "0-1-1" appear on three successive clock cycles, it will signal it with the output Z → stay

22.1.1 A Continuous "011" Detector → Moore version

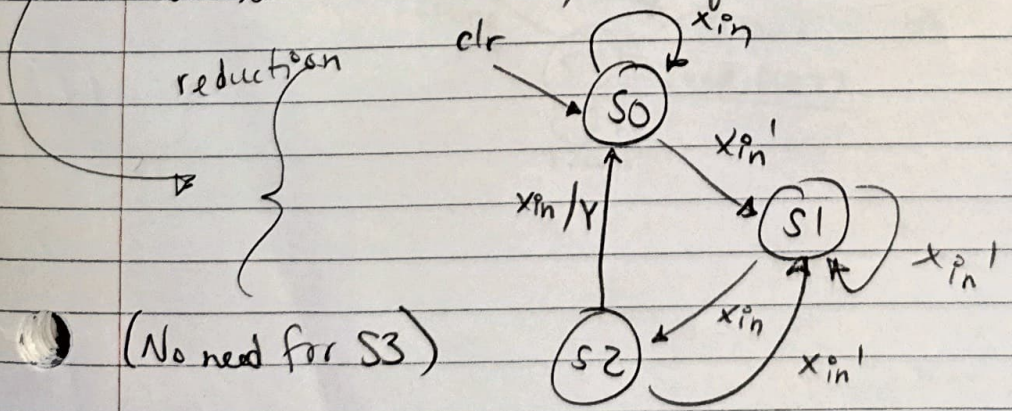


Same thing - except it resets → after 011 it searches again

22.1.2 - A Continuous "011" Detector - Mealy version



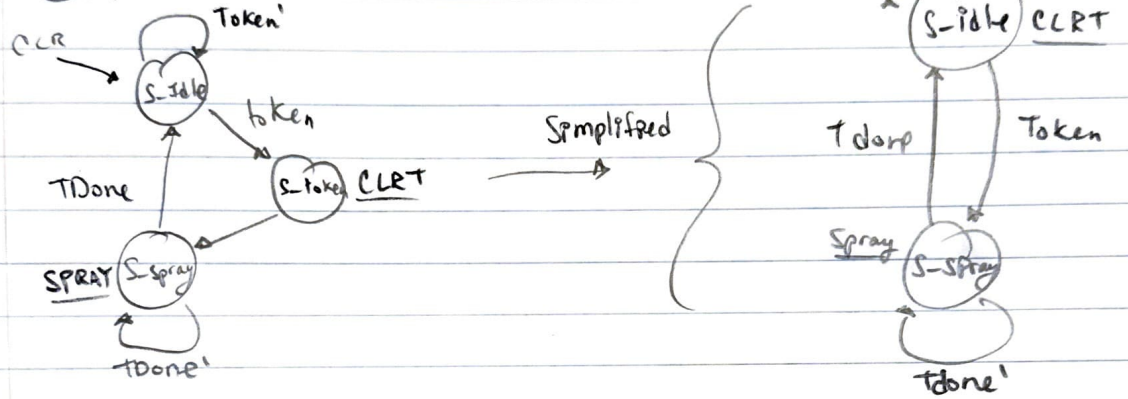
Same thing → but happens a cycle earlier because Y is in the transition, not after.



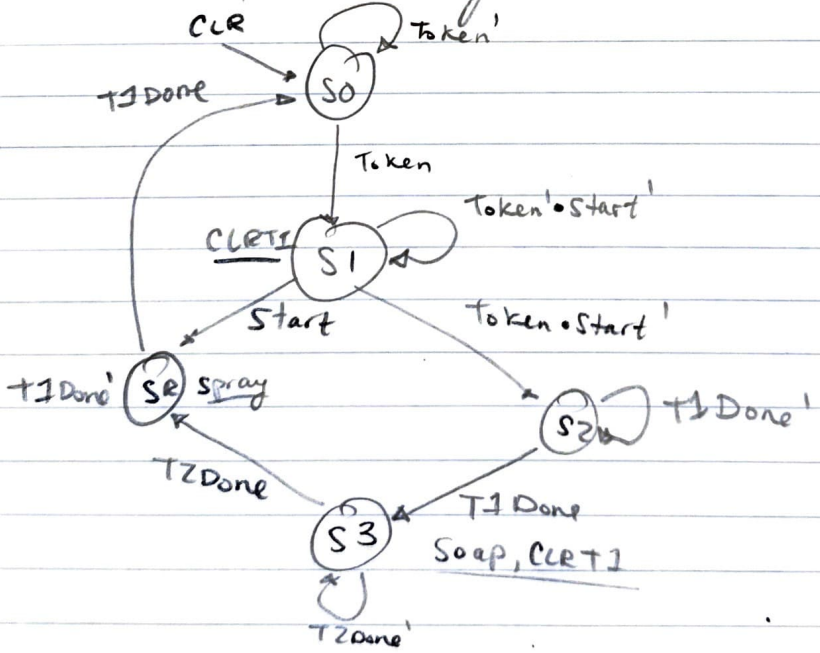
22.3 Resetting State Machines

↳ put them in every graph, whether CLR or a flip-flop

ex; Car wash



ex; Car wash w/ two settings



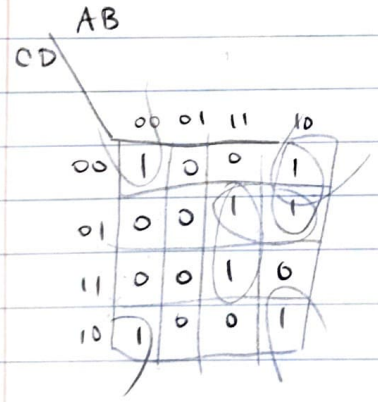
```
LD r0, 0x0   1 0 0 0
LD r1, 0xA   1 1 0 a
Disp, r0, r1 3 0 0 1
```

4 + 2 =
4 • 2 =
(Way 1) 4 - 2 =
(Way 2) 4 - 2 =

```
     LD r0, 0x0   1 0 0 0
     LD r1, 0x4   1 1 0 4
4    Disp r0, r1  3 0 0 1
```

```
     LD r4,
     LD r5,
+    Disp r4, r5
     LD r6
     LD r7
2    Disp r6, r7
     LD r0
     LD r1
=    Disp r0, r1
```

Class review:



$F = B'D' + ABD + AC'D + AB'C'$

(minimum times) Set-up - how long to hold d before the clock
hold - how long to leave D after the clock

max times Propagation delay (clk-q) - how long it takes q to change from clk

Rev. Cont.

A	B	C	D_A	D_B	D_C
0	0	0	0	1	0
0	0	1	X	X	X
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	X	0	X
1	1	0	1	0	1
1	1	1			

D_A

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

$D_A = C$

D_B

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

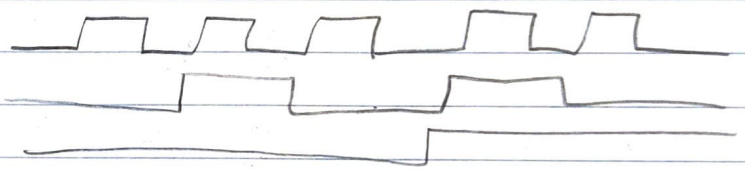
$D_B = A'$

D_C

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

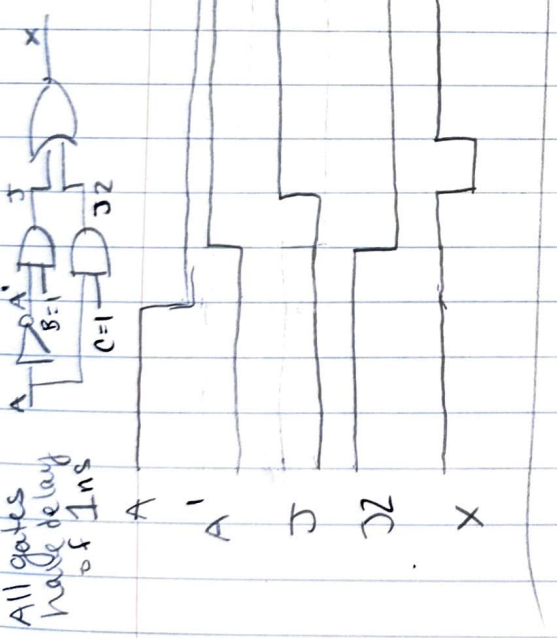
$D_C = B'$

Slowest output frequency of 3-bit counter w/ input 800KHz?



$MSB = CLK/2^n$

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



Practice Exam #3

1.) G

2.)

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

(Hazard free)

$F = B'D' + B'C' + A'C'D + A'BD + AB'C' + ACD'$

A

3.) B

8.) Clk Q change

10.) D can pass through

15.) $8KHz / 2^3 = 1KHz$ C

16.) G

17.)

0110	0000 xxxx
0010	0001 xxxx
1110	0010 1110
0101	0011 0110
1101	0100 xxxx
0111	0101 1101
1011	0110 0010
0011	0111 0000
0110	1000 xxxx
	1001 xxxx
	1010 xxxx
	1011 0011
	1100 xxxx
	1101 0111
	1110 0101
	1111 xxxx

18.) B

19.) ~~D~~ A

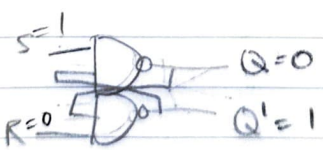
20.) A

Complete → OR pairs = 1

21.) Q=0 Q'=1, S=1 R=0

NAND

↳ A



NAND

00	1
01	1
10	1
11	0

22.) A ~~C~~ D

23.) A

24.) C

0-1 1-2 2-33-4

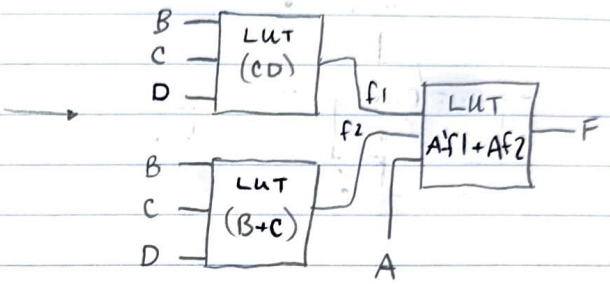
011
100
101

111111

100
011
100

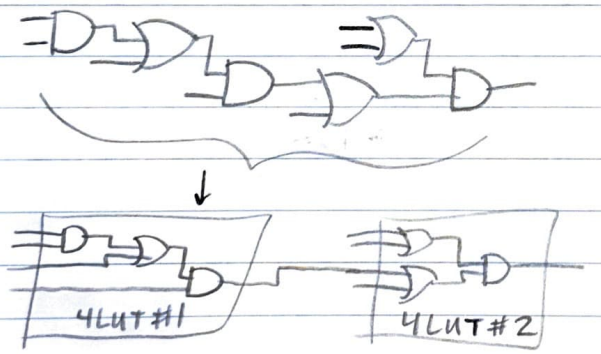
Chapter 25
Intro to Field Programmable Gate Arrays

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

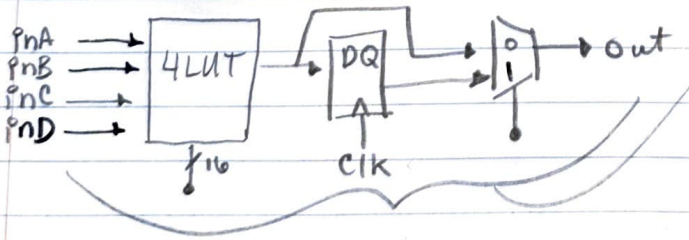


If A=1, Bottom half of table.

25.1.2 Mapping ~~Gate~~-Level Circuits To LUTs

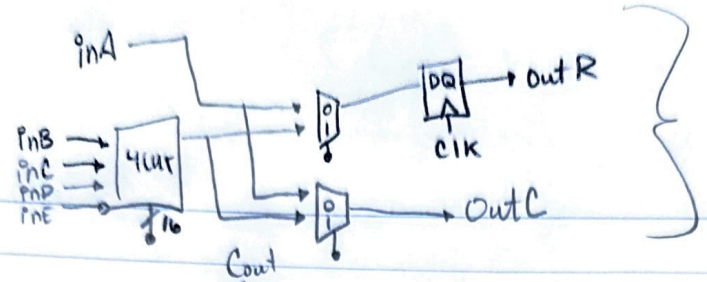


25.2 FPGA Logic Elements

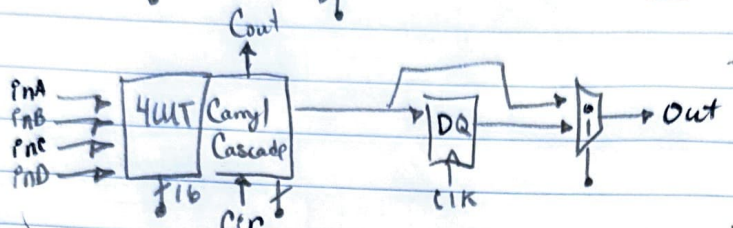


"Logic Element" (LE)

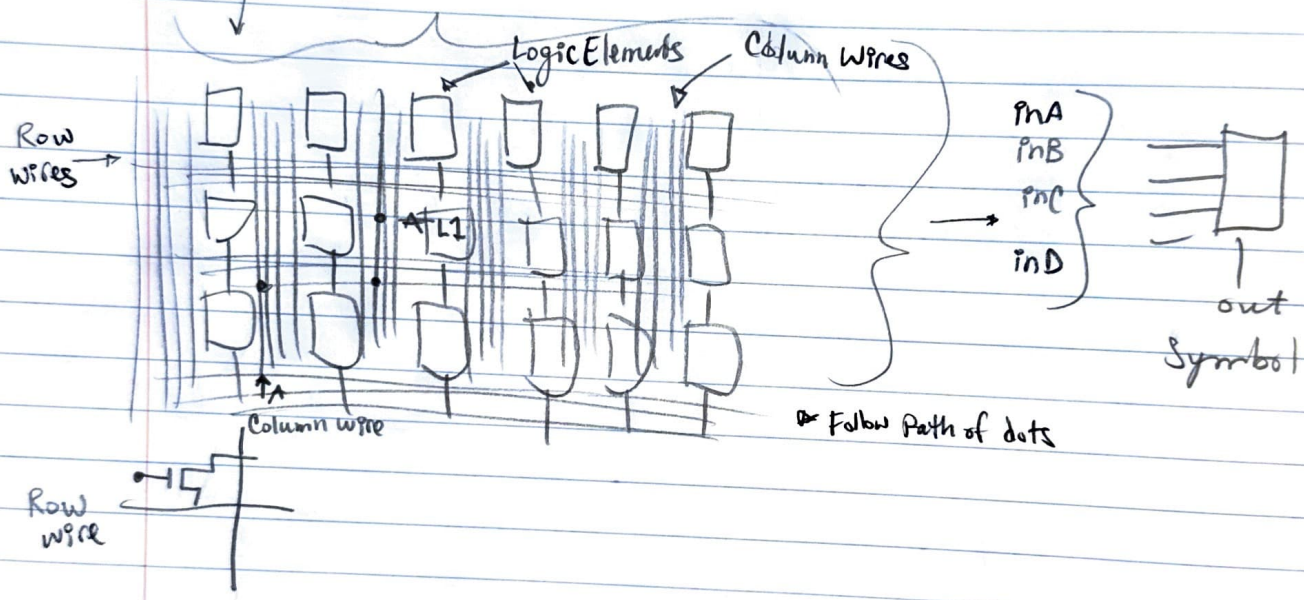
* Black dots are Configuration bits



Can be used for combinational or sequential logic

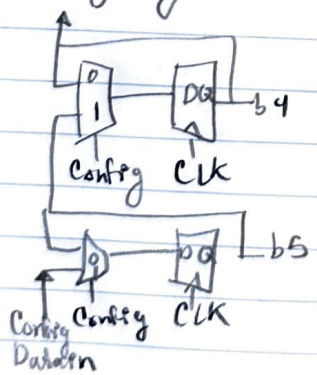


Performs 1-bit addition

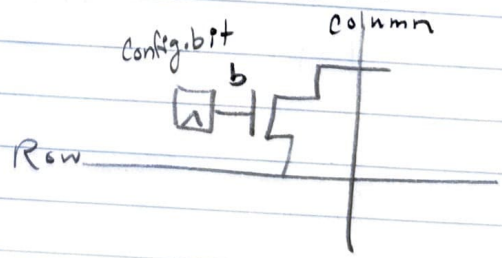


* Follow Path of data

Initializing the configuration bits



* Data is shifted in



if b=1, wires are connected