II. COMBINATIONAL LOGIC DESIGN

Combinational Logic ⇒ output of digital system is only dependent on current inputs (i.e., no memory)

(a) Boolean Algebra

- developed by George Boole in 1850s

- algebra defined on a set of 2 elements, {0, 1}, with binary operators multiply (AND), add (OR), and invert (NOT):

$X \cdot Y \equiv X$ AND Y

$X + Y \equiv X$ OR Y

X' or $\overline{X} \equiv$ NOT (X)

- Boolean algebra theorems:

One Variable Theorems		
Label	"·"	"+"
Identities	$X \cdot 1 = X$	$X + 0 = X$
Null elements	$X \cdot 0 = 0$	$X + 1 = 1$
Idempotency	$X \cdot X = X$	$X + X = X$
Complements	$X \cdot X' = 0$	$X + X' = 1$
Involution	$(X')' = X$	
Two/Three Variable Theorems		
Commutativity	$X \cdot Y = Y \cdot X$	$X + Y = Y + X$
Associativity	$(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$	$(X + Y) + Z = X + (Y + Z)$
Distributivity	$(X + Y) \cdot (X + Z) = X + Y \cdot Z$	$X \cdot Y + X \cdot Z = X \cdot (Y + Z)$
DeMorgan's	$(X \cdot Y)' = X' + Y'$	$(X + Y)' = X' \cdot Y'$

- *duality*:

→ to get "+" column from "·" column (and vice versa), swap "+" with "·" operators and swap 0s and 1s

e.g., Prove that $X + X \cdot Y = X$.

e.g., Prove distributivity for "·" using other theorems.

- two/three variable theorems can be generalized to *n* variables

→ for example, DeMorgan's theorem

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

$(ABCD\ldots)' = A'+B'+C'+D'\ldots$

Note: will often leave out "·" operator for convenience.

- *literal* ≡ primed or unprimed variable

- more on DeMorgan's:

AND + NOT ≡ NAND

by DeMorgan's

NOTs + OR

NAND ≡

- similarly, for NOR can show:

e.g., Given F = X′YZ′ + X′Y′Z, find F′ using DeMorgan's.

F′ =

- generalized DeMorgan to get F′, given F:

→ take dual of function F and complement literals

e.g., Given F = X′YZ′ + X′Y′Z, dual is

F^{dual} = (X′+Y+Z′)(X′+Y′+Z)

F′ = (X+Y′+Z)(X+Y+Z′) as expected.

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

XYZ	Minterm	Maxterm
000	m_0 =	M_0 =
001	m_1 =	M_1 =
010	m_2 =	M_2 =
011	m_3 =	M_3 =
100	m_4 =	M_4 =
101	m_5 =	M_5 =
110	m_6 =	M_6 =
111	m_7 =	M_7 =

e.g.,

XYZ	F
000	0
001	1
010	0
011	0
100	1
101	0
110	0
111	1

- *canonical sum of products* (SOP) form of Boolean function F
  ≡ sum of minterms corresponding to F = 1
  (also called "*standard sum of products*")

- *canonical product of sums* (POS) form of Boolean function F
  ≡ product of maxterms corresponding to F = 0
  (also called "*standard product of sums*")

- sometimes minterm list or maxterm list notation is used:

$$F = m_1 + m_4 + m_7$$
$$= \Sigma_{XYZ}(1, 4, 7)$$

and
$$F = M_0M_2M_3M_5M_6$$
$$= \Pi_{XYZ}(0, 2, 3, 5, 6)$$

- SOP and POS forms can usually be simplified to minimize literals → no longer "canonical"

e.g., simplified SOP: $F_1 = Y' + XY + X'YZ'$
simplified POS: $F_2 = X(Y'+Z)(X'+Y+Z')$

(b) Realization of Circuits

Design Objectives

(1) minimum number of gates
(2) minimum number of inputs to a gate
(3) minimum propagation time through circuit
(4) minimum number of interconnections

Boolean Function Input

e.g., $F = Y' + XY + X'YZ'$

- SOP form leads to 2 levels of logic:
  AND-OR logic $\rightarrow$ AND gates followed by OR gates
  (ignoring NOTs and assuming that any number of inputs to a gate is allowed)
- similarly, POS form leads to 2 levels of logic:
  OR-AND logic $\rightarrow$ OR gates followed by AND gates
  (ignoring NOTs and assuming that any number of inputs to a gate is allowed)

Truth Table to Gates

e.g., (same as previous example)

XYZ	F
000	1
001	1
010	1
011	0
100	1
101	1
110	1
111	1

SOP: F =

Note: using canonical SOP directly to gates often takes many gates and gates are large (in this example, 7 input OR gate!)

→ large gates can be built using smaller gates

e.g.,

6 2-input ORs ≡ 1 7-input OR, but 3 layers of logic gates
⇒ longer propagation delay ⇒ circuit slower

- in order to minimize circuit, desirable to simplify canonical SOP

- from canonical SOP, using Boolean algebra:

F =

=

=

=

= (reduced SOP form)

→ a lot of work and difficult to know exactly what steps to take

→ not guaranteed to find mimimal circuit

⇒ for this example, F = X + Y′ + Z′
(easily derived using canonical POS form)

→ some standard reduction/minimization/simplification methods exist such as Karnaugh maps for small functions and software packages such as Espresso for larger functions

- any logic function can be implemented using AND, OR, and NOT gates (by starting with SOP or POS forms), but CMOS technology lends itself to efficient implementation of NANDs and NORs

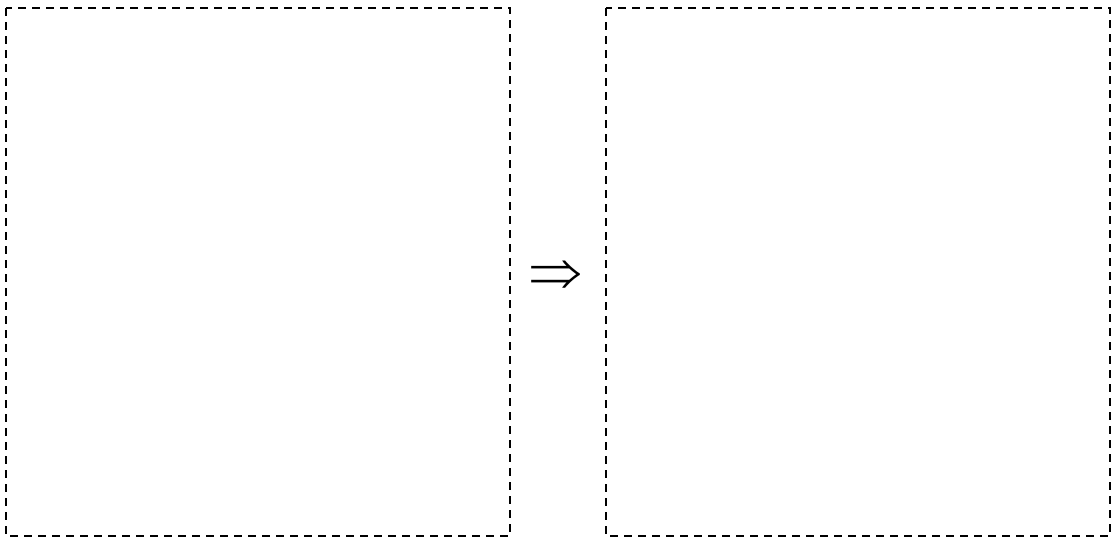
→ any logic function can be implemented with exclusively NAND (or NOR) gates

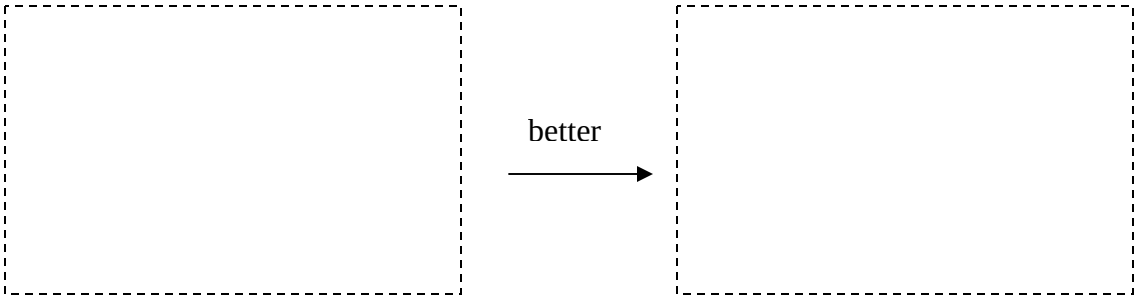
OR	2 NOTs + NAND	NOR + NOT
AND	NAND + NOT	2 NOTs + NOR
NOT	NAND	NOR

e.g.,

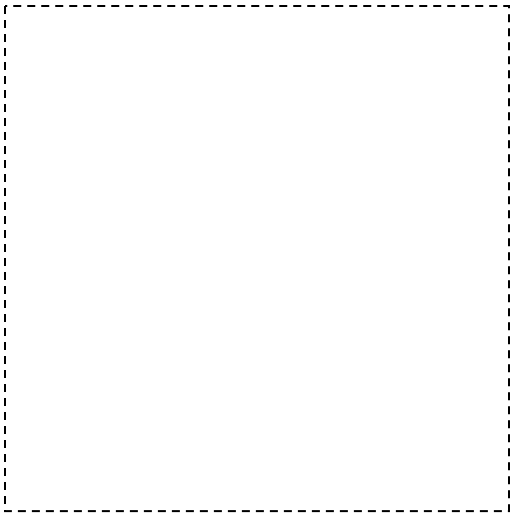
- similarly all NOR circuit can be derived (most easily for OR-AND circuit)

Example: Implementing a circuit using NANDs/NORs/NOTs





final circuit:



(c) Logic Minimization

- as we have seen, can use Boolean algebra theorems to reduce number and size of gates in a circuit
  → *logic minimization* or *logic simplification*

- also, there are sophisticated computer tools for minimization, such as the Espresso algorithm, which can find minimal or near-minimal circuit for most expressions with dozens of inputs and hundreds of product terms

- generally assume that X′ is readily available and to use X′ at gate input has no more cost that using X

Karnaugh Maps

- good systematic visual method for minimizing 3 and 4 input Boolean functions

3 input map

F:

X \ YZ	00	01	11	10
0	m_0	m_1	m_3	m_2
1	m_4	m_5	m_7	m_6

} Note: adjacent columns differ in only 1 bit $\Rightarrow$ adjacent squares differ in only 1 bit

e.g.,

(1) K-map of F:

X \ YZ	00	01	11	10
0	0	0	1	1
1	1	1	0	0

(2) K-map of F:

X \ YZ	00	01	11	10
0	0	0	1	0
1	1	0	1	1

(3) K-map of F:

X \ YZ	00	01	11	10
0	0	1	1	1
1	0	1	1	0

- for 3 input K-map:
  - 1 square ≡ term with 3 literals
  - 2 squares ≡ term with 2 literals
  - 4 squares ≡ term with 1 literal

4 input map

F:

WX \ YZ	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

→ again adjacent squares only differ in 1 bit

e.g.,

(1) K-map of F:

WX \ YZ	00	01	11	10
00	1	1	0	1
01	1	1	0	1
11	1	1	0	1
10	1	1	0	0

F =

(2) K-map of F:

WX \ YZ	00	01	11	10
00	1	1	0	1
01	0	0	0	1
11	0	0	0	0
10	1	1	0	1

F =

(3) K-map of F:

WX \ YZ	00	01	11	10
00	0	0	1	0
01	1	0	1	1
11	1	1	1	1
10	0	0	1	0

F =

K-maps for POS

e.g.,

X \ YZ	00	01	11	10
0	1	1	0	1
1	1	0	0	1

F =

F =

(Note also, F =

using K-map for SOP)

Don't Cares

- in some cases, outputs for given inputs can be either "0" or "1", whichever is convenient for design ⇒ "don't care"

  → indicated by "X" in values of truth table and K-map

- "don't cares" can be exploited to help minimize circuit

e.g.,

XYZ	**F**
000	1
001	X
010	1
011	1
100	0
101	0
110	X
111	1

X \ YZ	00	01	11	10
0	1	X	1	1
1	0	0	1	X

F =

(Compare to $X' \cdot Z' + Y \cdot Z$ if "don't cares" assumed to be "0".)

Multiple Output Minimization

- in many cases, there are multiple circuit outputs and considering them together can result in fewer gates

e.g., $F = XY + XZ + Y'Z$
$G = X'Y + XYZ$
$H = X'YZ + XZ$

$\rightarrow$ implemented independently:

5 2-input ANDs
2 3-input ANDs
2 2-input ORs
1 3-input OR

$\rightarrow$ implemented together:

6 2-input ANDs (+1)
0 3-input ANDs (–2)
2 2-input ORs
1 3-input OR

(d) Combinational Logic Design Examples

Summary of Combinational Logic Design

(1) Inputs
→ wording, truth table, Boolean function, K-map
(2) Objectives
→ minimize # and size of gates, minimize timing delay
(3) Constraints
→ NANDs only, maximum timing delays, gate driving capabilities, limitations on gate size
(4) Tools
→ Boolean algebra, SOP/POS forms, Karnaugh maps (for small circuits with ≤ 6 inputs) , Espresso (for large circuits)

Example 1: Temperature Controller

- temperature sensor produces following inputs to controller:

Temperature	4-bit Input Code	Action
<15°	0000	heat on, fan on high
15°	0001	
16°	0010	
17°	0011	heat on, fan on low
18°	0100	
19°	0101	
20°	0110	heat off, AC off
21°	0111	AC on
22°	1000	
23°	1001	
24°	1010	
25°	1011	
>25°	1100	

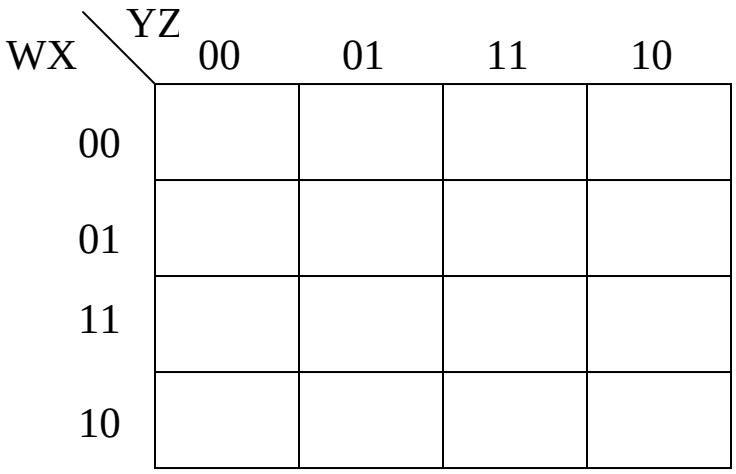
- controller should control furnace/AC unit with 3 outputs with the objective of keeping the temperature at 20°:

H (heat+fan on/off) 1 ≡ on, 0 ≡ off
F (fan low/high) 0 ≡ low, 1 ≡ high
C (AC on/off) 1 ≡ on, 0 ≡ off

Design a NAND-only circuit to implement the controller logic.

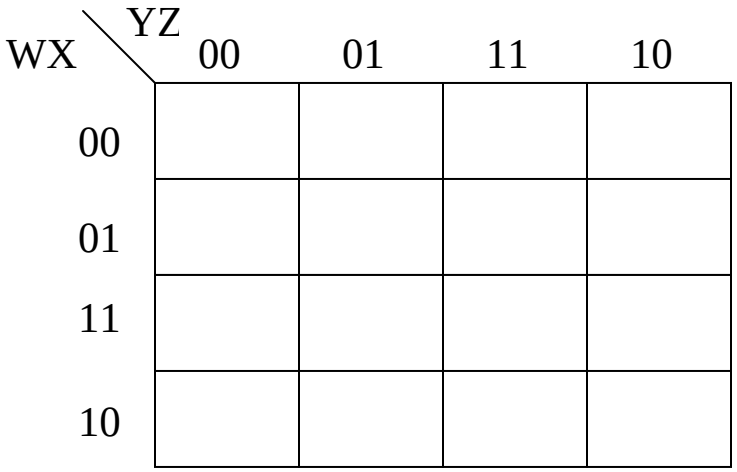
WXYZ	**H**	**F**	**C**
0000			
0001			
0010			
0011			
0100			
0101			
0110			
0111			
1000			
1001			
1010			
1011			
1100			
1101			
1110			
1111			

H:



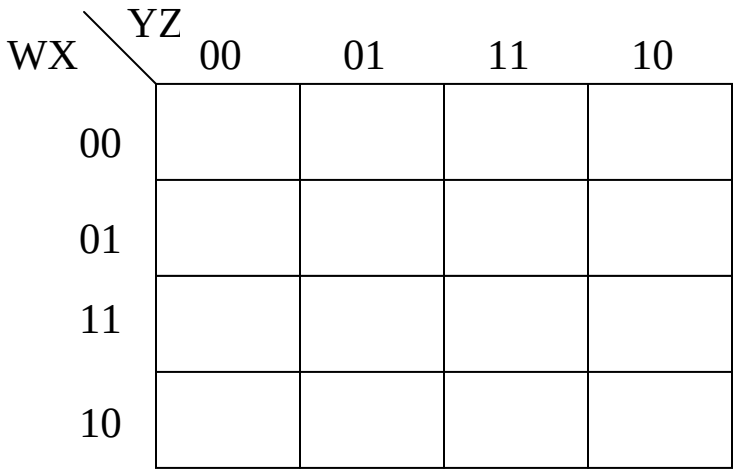
H =

F:



F =

C:



C =

Resulting circuit using NANDs only:

Example 2: 2-out-of-5 Encoding

- 2-out-of-5 encoder encodes digits as follows:

Digit	Code
0	11000
1	00011
2	00101
3	00110
4	01001
5	01010
6	01100
7	10001
8	10010
9	10100

(Aside: What is value of 2-out-of-5 encoding?)

Design a logic circuit to convert a binary representation of a digit to a 2-out-of-5 code.

Truth table:

WXYZ	A	B	C	D	E
0000					
0001					
0010					
0011					
0100					
0101					
0110					
0111					
1000					
1001					
1010					
1011					
1100					
1101					
1110					
1111					

A:

WX \ YZ	00	01	11	10
00				
01				
11				
10				

A =

B:

WX \ YZ	00	01	11	10
00				
01				
11				
10				

B =

C:

WX \ YZ	00	01	11	10
00				
01				
11				
10				

C =

D:

WX \ YZ	00	01	11	10
00				
01				
11				
10				

D =

E:

WX \ YZ	00	01	11	10
00				
01				
11				
10				

E =

Draw circuit. Be sure to share gates where possible across multiple outputs.

Example 3: Weathervane

Design a logic circuit which takes four binary inputs indicating north, east, south, and west wind components (i.e., N = 1 indicates a component of wind blowing north) and produces an output of "1" when the wind direction is northeast or southwest.

Note: N = S = 1 and E = W = 1 are not possible.

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

NE \ SW	00	01	11	10
00				
01				
11				
10				

F =

Not surprising!

Compare to result based on SOP and not taking into account "don't cares":

F =