Combinational logic

- Number Representations
- Basic logic
  - Boolean algebra, proofs by re-writing, proofs by perfect induction
  - logic functions, truth tables, and switches
  - NOT, AND, OR, NAND, NOR, XOR, . . ., minimal set
- Logic realization
  - two-level logic and canonical forms
  - incompletely specified functions
- Simplification
  - uniting theorem
  - grouping of terms in Boolean functions
- Alternate representations of Boolean functions
  - cubes
  - Karnaugh maps

Digital

- Digital = discrete
  - Binary codes (example: BCD)
  - Decimal digits 0-9
  - DNA nucleotides
- Binary codes
  - Represent symbols using binary digits (bits)
- Digital computers:
  - I/O is digital
    - ASCII, decimal, etc.
  - Internal representation is binary
    - Process information in bits

Decimal Symbols	BCD Code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The basics: Binary numbers

- Bases we will use
  - Binary: Base 2
  - Octal: Base 8
  - Hexadecimal: Base 16
- Positional number system
  - $101_2 = 1\times2^2 + 0\times2^1 + 1\times2^0$
  - $63_8 = 6\times8^1 + 3\times8^0$
  - $A1_{16} = 10\times16^1 + 1\times16^0$
- Addition and subtraction

$$\begin{array}{r} 1011 \\ +\ 1010 \\ \hline 10101 \end{array} \qquad \begin{array}{r} 1011 \\ -\ 0110 \\ \hline 0101 \end{array}$$

Binary → hex/decimal/octal conversion

- Conversion from binary to octal/hex
  - Binary: 10011110001
  - Octal: 10 | 011 | 110 | 001=2361_8
  - Hex: 100 | 1111 | 0001=$4F1_{16}$
- Conversion from binary to decimal
  - $101_2 = 1\times2^2 + 0\times2^1 + 1\times2^0 = 5_{10}$
  - $63.4_8 = 6\times8^1 + 3\times8^0 + 4\times8^{-1} = 51.5_{10}$
  - $A1_{16} = 10\times16^1 + 1\times16^0 = 161_{10}$

Decimal→ binary/octal/hex conversion

Binary

	Quotient	Remainder
56÷2=	28	0
28÷2=	14	0
14÷2=	7	0
7÷2=	3	1
3÷2=	1	1
1÷2=	0	1

Octal

	Quotient	Remainder
56÷8=	7	0
7÷8=	0	7

$56_{10}=111000_2$
$56_{10}=70_8$

- Why does this work?
  - $N=56_{10}=111000_2$
  - Q=N/2=56/2=111000/2=11100 remainder 0
- Each successive divide liberates an LSB

Number systems

- How do we write negative binary numbers?
- Historically: 3 approaches
  - Sign-and-magnitude
  - Ones-complement
  - Twos-complement
- For all 3, the most-significant bit (msb) is the sign digit
  - 0 ≡ positive
  - 1 ≡ negative
- Learn twos-complement
  - Simplifies arithmetic
  - Used almost universally

Sign-and-magnitude

- The most-significant bit (msb) is the sign digit
  - 0 ≡ positive
  - 1 ≡ negative
- The remaining bits are the number's magnitude
- Problem 1: Two representations for zero
  - 0 = 0000 *and also* –0 = 1000
- Problem 2: Arithmetic is cumbersome

Add		Subtract			Compare and subtract		
4 + 3	0100 + 0011	4 – 3	0100 + 1011	0100 – 0011	– 4 + 3	1100 + 0011	1100 – 0011
= 7	= 0111	= 1	≠ 1111	= 0001	– 1	≠ 1111	= 1001

Ones-complement

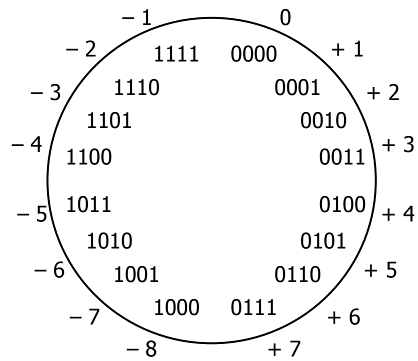
- Negative number: Bitwise complement positive number
  - 0011 ≡ 3_{10}
  - 1100 ≡ $–3_{10}$
- Solves the arithmetic problem

Add		Invert, add, add carry		Invert and add	
4 + 3	0100 + 0011	4 – 3	0100 + 1100	– 4 + 3	1011 + 0011
= 7	= 0111	= 1 add carry:	1 0000 +1	– 1	1110
			= 0001		

- Remaining problem: Two representations for zero
  - 0 = 0000 *and also* –0 = 1111

Twos-complement

- Negative number: Bitwise complement plus one
  - $0011 \equiv 3_{10}$
  - $1101 \equiv -3_{10}$
- Number wheel
- Only one zero!
- msb is the sign digit
  - $0 \equiv$ positive
  - $1 \equiv$ negative



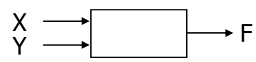
Twos-complement (con't)

- Complementing a complement ➲ the original number
- Arithmetic is easy
  - Subtraction = negation and addition
    - Easy to implement in hardware

Add		Invert and add		Invert and add	
4 + 3	0100 + 0011	4 – 3	0100 + 1101	– 4 + 3	1100 + 0011
= 7	= 0111	= 1 drop carry	1 0001 = 0001	– 1	1111

Possible logic functions of two variables

- There are 16 possible functions of 2 input variables:
  - in general, there are 2**(2**n) functions of n inputs



X	Y	16 possible functions (F0–F15)															
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

0
X and Y
X
Y
X xor Y
X or Y
X nor Y
not (X or Y)
X = Y
not Y
not X
X nand Y
not (X and Y)
1

Cost of different logic functions

- Different functions are easier or harder to implement
  - each has a cost associated with the number of switches needed
  - 0 (F0) and 1 (F15): require 0 switches, directly connect output to low/high
  - X (F3) and Y (F5): require 0 switches, output is one of inputs
  - X’ (F12) and Y’ (F10): require 2 switches for "inverter" or NOT-gate
  - X nor Y (F4) and X nand Y (F14): require 4 switches
  - X or Y (F7) and X and Y (F1): require 6 switches
  - X = Y (F9) and X ⊕ Y (F6): require 16 switches

  - thus, because NOT, NOR, and NAND are the cheapest they are the functions we implement the most in practice

Minimal set of functions

- Can we implement all logic functions from NOT, NOR, and NAND?
  - For example, implementing X and Y
    is the same as implementing not (X nand Y)
- In fact, we can do it with only NOR or only NAND
  - NOT is just a NAND or a NOR with both inputs tied together

X	Y	X nor Y
0	0	1
1	1	0

X	Y	X nand Y
0	0	1
1	1	0

  - and NAND and NOR are "duals",
    that is, its easy to implement one using the other

X nand Y ≡ not ((not X) nor (not Y))
X nor Y ≡ not ((not X) nand (not Y))

- But lets not move too fast . . .
  - lets look at the mathematical foundation of logic

An algebraic structure

- An algebraic structure consists of
  - a set of elements B
  - binary operations { + , • }
  - and a unary operation { ' }
  - such that the following axioms hold:

1. the set B contains at least two elements: a, b

2. closure:	a + b is in B	a • b is in B
3. commutativity:	a + b = b + a	a • b = b • a
4. associativity:	a + (b + c) = (a + b) + c	a • (b • c) = (a • b) • c
5. identity:	a + 0 = a	a • 1 = a
6. distributivity:	a + (b • c) = (a + b) • (a + c)	a • (b + c) = (a • b) + (a • c)
7. complementarity:	a + a' = 1	a • a' = 0

Boolean algebra

- Boolean algebra
  - B = {0, 1}
  - variables
  - + is logical OR, • is logical AND
  - ’ is logical NOT
- All algebraic axioms hold

Logic functions and Boolean algebra

- Any logic function that can be expressed as a truth table can be written as an expression in Boolean algebra using the operators: ’, +, and •

X	Y	X • Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X’	X’ • Y
0	0	1	0
0	1	1	1
1	0	0	0
1	1	0	0

X	Y	X’	Y’	X • Y	X’ • Y’	(X • Y) + (X’ • Y’)
0	0	1	1	0	1	1
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	1	0	0	1	0	1

(X • Y) + (X’ • Y’) ≡ X = Y

Boolean expression that is true when the variables X and Y have the same value and false, otherwise

X, Y are Boolean algebra variables

Axioms and theorems of Boolean algebra

- identity
  1. $X + 0 = X$ — 1D. $X \cdot 1 = X$
- null
  2. $X + 1 = 1$ — 2D. $X \cdot 0 = 0$
- idempotency:
  3. $X + X = X$ — 3D. $X \cdot X = X$
- involution:
  4. $(X')' = X$
- complementarity:
  5. $X + X' = 1$ — 5D. $X \cdot X' = 0$
- commutativity:
  6. $X + Y = Y + X$ — 6D. $X \cdot Y = Y \cdot X$
- associativity:
  7. $(X + Y) + Z = X + (Y + Z)$ — 7D. $(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$

Axioms and theorems of Boolean algebra (cont'd)

- distributivity:
  8. $X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)$ — 8D. $X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$
- uniting:
  9. $X \cdot Y + X \cdot Y' = X$ — 9D. $(X + Y) \cdot (X + Y') = X$
- absorption:
  10. $X + X \cdot Y = X$ — 10D. $X \cdot (X + Y) = X$
  11. $(X + Y') \cdot Y = X \cdot Y$ — 11D. $(X \cdot Y') + Y = X + Y$
- factoring:
  12. $(X + Y) \cdot (X' + Z) = X \cdot Z + X' \cdot Y$ — 12D. $X \cdot Y + X' \cdot Z = (X + Z) \cdot (X' + Y)$
- concensus:
  13. $(X \cdot Y) + (Y \cdot Z) + (X' \cdot Z) = X \cdot Y + X' \cdot Z$ — 13D. $(X + Y) \cdot (Y + Z) \cdot (X' + Z) = (X + Y) \cdot (X' + Z)$

Axioms and theorems of Boolean algebra (cont'd)

- de Morgan's:
  14. $(X + Y + ...)' = X' \bullet Y' \bullet ...$ 14D. $(X \bullet Y \bullet ...)' = X' + Y' + ...$
- generalized de Morgan's:
  15. $f'(X_1,X_2,...,X_n,0,1,+,\bullet) = f(X_1',X_2',...,X_n',1,0,\bullet,+)$
- establishes relationship between • and +

Axioms and theorems of Boolean algebra (cont'd)

- Duality
  - a dual of a Boolean expression is derived by replacing • by +, + by •, 0 by 1, and 1 by 0, and leaving variables unchanged
  - any theorem that can be proven is thus also proven for its dual!
  - a meta-theorem (a theorem about theorems)
- duality:
  16. $X + Y + ... \Leftrightarrow X \bullet Y \bullet ...$
- generalized duality:
  17. $f(X_1,X_2,...,X_n,0,1,+,\bullet) \Leftrightarrow f(X_1,X_2,...,X_n,1,0,\bullet,+)$
- Different than deMorgan's Law
  - this is a statement about theorems
  - this is not a way to manipulate (re-write) expressions

Proving theorems (rewriting)

- Using the axioms of Boolean algebra:
  - e.g., prove the theorem: $X \cdot Y + X \cdot Y' = X$

distributivity (8)	$X \bullet Y + X \bullet Y'$	$=$	$X \bullet (Y + Y')$
complementarity (5)	$X \bullet (Y + Y')$	$=$	$X \bullet (1)$
identity (1D)	$X \bullet (1)$	$=$	X ✓

  - e.g., prove the theorem: $X + X \cdot Y = X$

identity (1D)	$X + X \bullet Y$	$=$	$X \bullet 1 + X \bullet Y$
distributivity (8)	$X \bullet 1 + X \bullet Y$	$=$	$X \bullet (1 + Y)$
identity (2)	$X \bullet (1 + Y)$	$=$	$X \bullet (1)$
identity (1D)	$X \bullet (1)$	$=$	X ✓

Activity

- Prove the following using the laws of Boolean algebra:
  - $(X \cdot Y) + (Y \cdot Z) + (X' \cdot Z) = X \cdot Y + X' \cdot Z$

Proving theorems (perfect induction)

- Using perfect induction (complete truth table):
  - e.g., de Morgan's:

(X + Y)′ = X′ • Y′
NOR is equivalent to AND
with inputs complemented

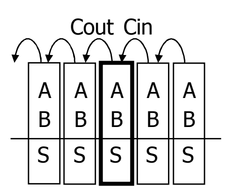
X	Y	X′	Y′	(X + Y)′	X′ • Y′
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

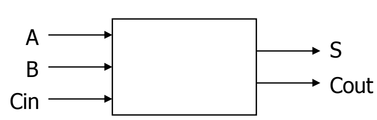
(X • Y)′ = X′ + Y′
NAND is equivalent to OR
with inputs complemented

X	Y	X′	Y′	(X • Y)′	X′ + Y′
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

A simple example: 1-bit binary adder

- Inputs: A, B, Carry-in
- Outputs: Sum, Carry-out





A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S = A′ B′ Cin + A′ B Cin′ + A B′ Cin′ + A B Cin

Cout = A′ B Cin + A B′ Cin + A B Cin′ + A B Cin

Apply the theorems to simplify expressions

- The theorems of Boolean algebra can simplify Boolean expressions
  - e.g., full adder's carry-out function (same rules apply to any function)

Cout = A' B Cin + A B' Cin + A B Cin' + A B Cin
= A' B Cin + A B' Cin + A B Cin' + $\boxed{\text{A B Cin + A B Cin}}$
= A' B Cin + A B Cin + A B' Cin + A B Cin' + A B Cin
= (A' + A) B Cin + A B' Cin + A B Cin' + A B Cin
= (1) B Cin + A B' Cin + A B Cin' + A B Cin
= B Cin + A B' Cin + A B Cin' + $\boxed{\text{A B Cin + A B Cin}}$
= B Cin + A B' Cin + A B Cin + A B Cin' + A B Cin
= B Cin + A (B' + B) Cin + A B Cin' + A B Cin
= B Cin + A (1) Cin + A B Cin' + A B Cin
= B Cin + A Cin + A B (Cin' + Cin)
= B Cin + A Cin + A B (1)
= B Cin + A Cin + A B

adding extra terms creates new factoring opportunities

Activity

- Fill in the truth-table for a circuit that checks that a 4-bit number is divisible by 2, 3, or 5

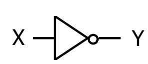
X8	X4	X2	X1	By2	By3	By5
0	0	0	0	1	1	1
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	1	0

- Write down Boolean expressions for By2, By3, and By5

Activity

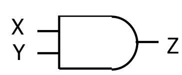
From Boolean expressions to logic gates

- NOT X' $\overline{X}$ $\sim X$



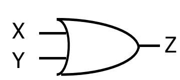
X	Y
0	1
1	0

- AND $X \bullet Y$ XY $X \wedge Y$



X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

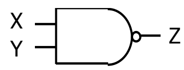
- OR $X + Y$ $X \vee Y$



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

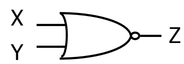
From Boolean expressions to logic gates (cont'd)

- NAND



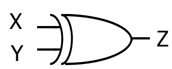
X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

- NOR



X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	0

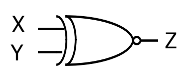
- XOR
  $X \oplus Y$



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

X <u>xor</u> Y = X Y' + X' Y
X or Y but not both
("inequality", "difference")

- XNOR
  X = Y

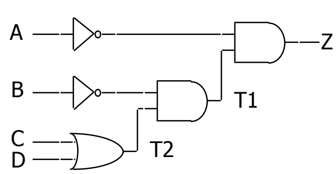


X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

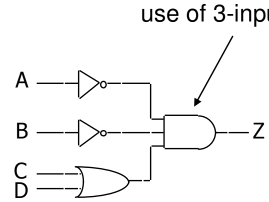
X <u>xnor</u> Y = X Y + X' Y'
X and Y are the same
("equality", "coincidence")

From Boolean expressions to logic gates (cont'd)

- More than one way to map expressions to gates
  - e.g., $Z = A' \cdot B' \cdot (C + D) = (A' \cdot \underbrace{(B' \cdot \underbrace{(C + D)}_{T2})}_{T1})$

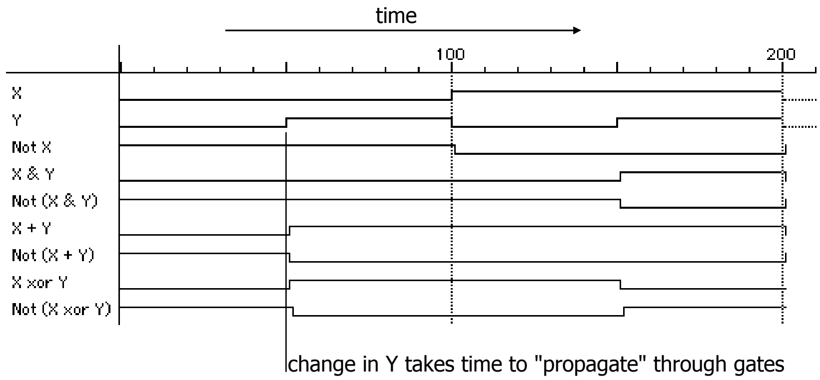


use of 3-input gate



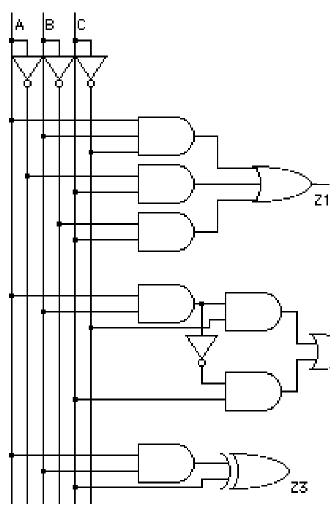
Waveform view of logic functions

- Just a sideways truth table
  - but note how edges don't line up exactly
  - it takes time for a gate to switch its output!



Choosing different realizations of a function

A	B	C	Z
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	1
1	1	1	0



two-level realization
(we don't count NOT gates)

multi-level realization
(gates with fewer inputs)

XOR gate (easier to draw
but costlier to build)

Which realization is best?

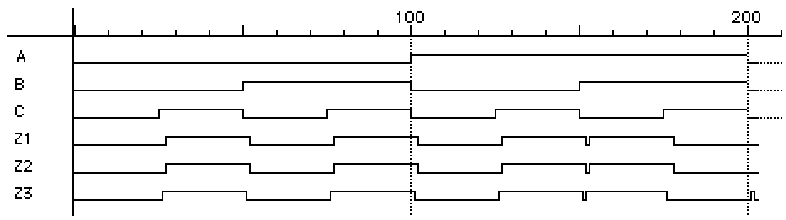
- Reduce number of inputs
  - literal: input variable (complemented or not)
    - can approximate cost of logic gate as 2 transitors per literal
    - why not count inverters?
  - fewer literals means less transistors
    - smaller circuits
  - fewer inputs implies faster gates
    - gates are smaller and thus also faster
  - fan-ins (# of gate inputs) are limited in some technologies
- Reduce number of gates
  - fewer gates (and the packages they come in) means smaller circuits
    - directly influences manufacturing costs

Which is the best realization? (cont'd)

- Reduce number of levels of gates
  - fewer level of gates implies reduced signal propagation delays
  - minimum delay configuration typically requires more gates
    - wider, less deep circuits
- How do we explore tradeoffs between increased circuit delay and size?
  - automated tools to generate different solutions
  - logic minimization: reduce number of gates and complexity
  - logic optimization: reduction while trading off against delay

Are all realizations equivalent?

- Under the same input stimuli, the three alternative implementations have
  almost the same waveform behavior
  - delays are different
  - glitches (hazards) may arise – these could be bad, it depends
  - variations due to differences in number of gate levels and structure
- The three implementations are functionally equivalent



Implementing Boolean functions

- Technology independent
  - canonical forms
  - two-level forms
  - multi-level forms
- Technology choices
  - packages of a few gates
  - regular logic
  - two-level programmable logic
  - multi-level programmable logic

Canonical forms

- Truth table is the unique signature of a Boolean function
- The same truth table can have many gate realizations
- Canonical forms
  - standard forms for a Boolean expression
  - provides a unique algebraic signature

Sum-of-products canonical forms

- Also known as disjunctive normal form
- Also known as minterm expansion

F = *001* *011* *101* *110* *111*

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

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

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

Sum-of-products canonical form (cont'd)

- Product term (or minterm)
  - ANDed product of literals – input combination for which output is true
  - each variable appears exactly once, true or inverted (but not both)

A	B	C	minterms	
0	0	0	A'B'C'	m0
0	0	1	A'B'C	m1
0	1	0	A'BC'	m2
0	1	1	A'BC	m3
1	0	0	AB'C'	m4
1	0	1	AB'C	m5
1	1	0	ABC'	m6
1	1	1	ABC	m7

short-hand notation for minterms of 3 variables

F in canonical form:

F(A, B, C) = Σm(1,3,5,6,7)
= m1 + m3 + m5 + m6 + m7
= A'B'C + A'BC + AB'C + ABC' + ABC

canonical form ≠ minimal form

F(A, B, C) = A'B'C + A'BC + AB'C + ABC + ABC'
= (A'B' + A'B + AB' + AB)C + ABC'
= ((A' + A)(B' + B))C + ABC'
= C + ABC'
= ABC' + C
= AB + C

Product-of-sums canonical form

- Also known as conjunctive normal form
- Also known as maxterm expansion

F = *000* *010* *100*

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

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

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

Product-of-sums canonical form (cont'd)

- Sum term (or maxterm)
  - ORed sum of literals – input combination for which output is false
  - each variable appears exactly once, true or inverted (but not both)

A	B	C	maxterms	
0	0	0	A+B+C	M0
0	0	1	A+B+C′	M1
0	1	0	A+B′+C	M2
0	1	1	A+B′+C′	M3
1	0	0	A′+B+C	M4
1	0	1	A′+B+C′	M5
1	1	0	A′+B′+C	M6
1	1	1	A′+B′+C′	M7

short-hand notation for maxterms of 3 variables

F in canonical form:

F(A, B, C) = ΠM(0,2,4)
= M0 • M2 • M4
= (A + B + C) (A + B′ + C) (A′ + B + C)

canonical form ≠ minimal form

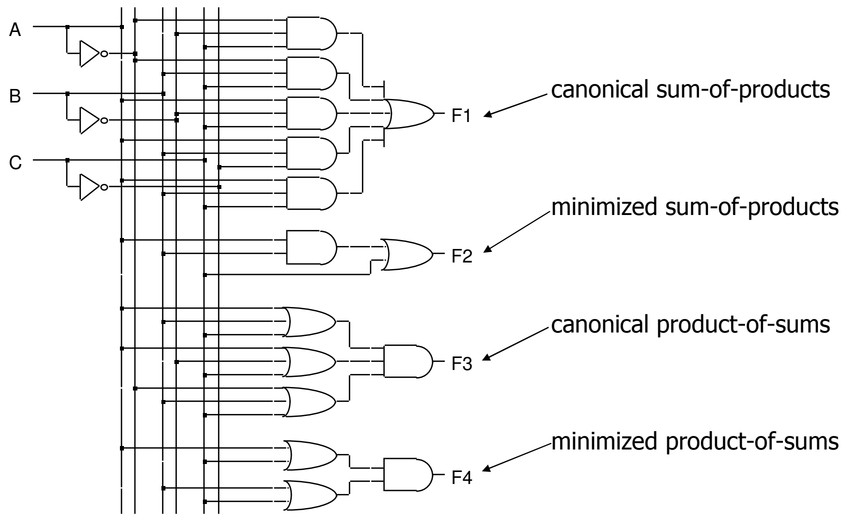
F(A, B, C) = (A + B + C) (A + B′ + C) (A′ + B + C)
= (A + B + C) (A + B′ + C)
(A + B + C) (A′ + B + C)
= (A + C) (B + C)

S-o-P, P-o-S, and de Morgan's theorem

- Sum-of-products
  - F' = A'B'C' + A'BC' + AB'C'
- Apply de Morgan's
  - (F')' = (A'B'C' + A'BC' + AB'C')'
  - F = (A + B + C) (A + B' + C) (A' + B + C)

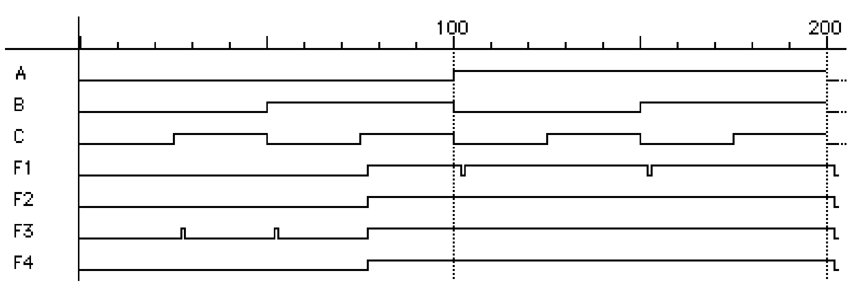
- Product-of-sums
  - F' = (A + B + C') (A + B' + C') (A' + B + C') (A' + B' + C) (A' + B' + C')
- Apply de Morgan's
  - (F')' = ((A + B + C')(A + B' + C')(A' + B + C')(A' + B' + C)(A' + B' + C'))'
  - F = A'B'C + A'BC + AB'C + ABC' + ABC

Four alternative two-level implementations of F = AB + C



Waveforms for the four alternatives

- Waveforms are essentially identical
  - except for timing hazards (glitches)
  - delays almost identical (modeled as a delay per level, not type of gate or number of inputs to gate)



Mapping between canonical forms

- Minterm to maxterm conversion
  - use maxterms whose indices do not appear in minterm expansion
  - e.g., $F(A,B,C) = \Sigma m(1,3,5,6,7) = \Pi M(0,2,4)$
- Maxterm to minterm conversion
  - use minterms whose indices do not appear in maxterm expansion
  - e.g., $F(A,B,C) = \Pi M(0,2,4) = \Sigma m(1,3,5,6,7)$
- Minterm expansion of F to minterm expansion of F'
  - use minterms whose indices do not appear
  - e.g., $F(A,B,C) = \Sigma m(1,3,5,6,7)$ $F'(A,B,C) = \Sigma m(0,2,4)$
- Maxterm expansion of F to maxterm expansion of F'
  - use maxterms whose indices do not appear
  - e.g., $F(A,B,C) = \Pi M(0,2,4)$ $F'(A,B,C) = \Pi M(1,3,5,6,7)$

Incompleteley specified functions

- Example: binary coded decimal increment by 1
  - BCD digits encode the decimal digits 0 – 9
    in the bit patterns 0000 – 1001

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
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

off-set of W

on-set of W

don't care (DC) set of W

these inputs patterns should never be encountered in practice – **<u>"don't care"</u>** about associated output values, can be exploited in minimization

Notation for incompletely specified functions

- Don't cares and canonical forms
  - so far, only represented on-set
  - also represent don't-care-set
  - need two of the three sets (on-set, off-set, dc-set)
- Canonical representations of the BCD increment by 1 function:
  - Z = m0 + m2 + m4 + m6 + m8 + d10 + d11 + d12 + d13 + d14 + d15
  - Z = Σ [m(0,2,4,6,8) + d(10,11,12,13,14,15)]
  - Z = M1 • M3 • M5 • M7 • M9 • D10 • D11 • D12 • D13 • D14 • D15
  - Z = Π [M(1,3,5,7,9) • D(10,11,12,13,14,15)]

Simplification of two-level combinational logic

- Finding a minimal sum of products or product of sums realization
  - exploit don't care information in the process
- Algebraic simplification
  - not an algorithmic/systematic procedure
  - how do you know when the minimum realization has been found?
- Computer-aided design tools
  - precise solutions require very long computation times, especially for functions with many inputs (> 10)
  - heuristic methods employed – "educated guesses" to reduce amount of computation and yield good if not best solutions
- Hand methods still relevant
  - to understand automatic tools and their strengths and weaknesses
  - ability to check results (on small examples)

The uniting theorem

- Key tool to simplification: A (B' + B) = A
- Essence of simplification of two-level logic
  - find two element subsets of the ON-set where only one variable changes its value – this single varying variable can be eliminated and a single product term used to represent both elements

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

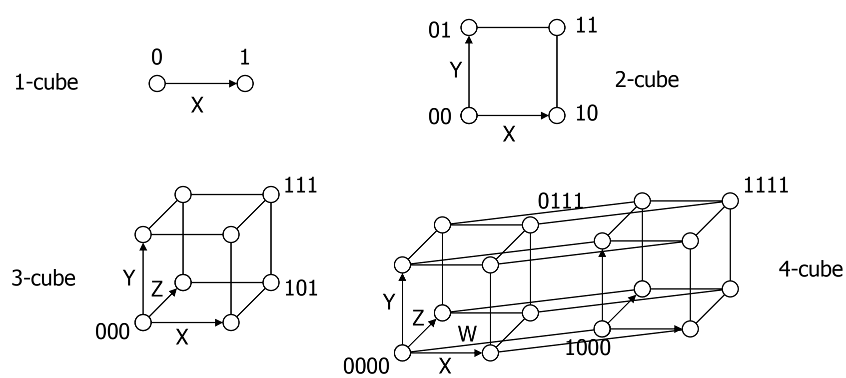
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

B has the same value in both on-set rows
– B remains

A has a different value in the two rows
– A is eliminated

Boolean cubes

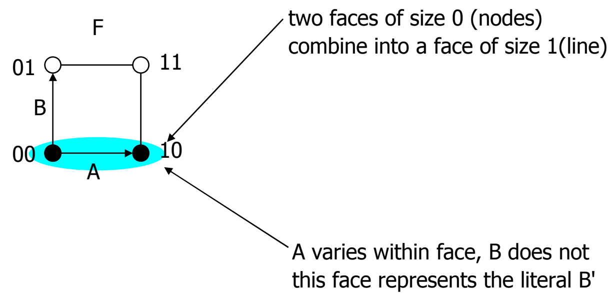
- Visual technique for indentifying when the uniting theorem can be applied
- n input variables = n-dimensional "cube"



Mapping truth tables onto Boolean cubes

- Uniting theorem combines two "faces" of a cube into a larger "face"
- Example:

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

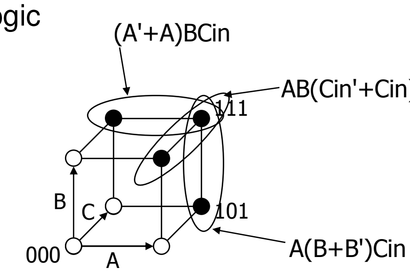


ON-set = solid nodes
OFF-set = empty nodes
DC-set = ×'d nodes

Three variable example

- Binary full-adder carry-out logic

A	B	Cin	Cout
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

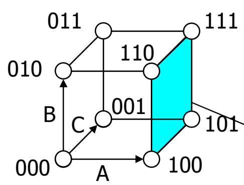


the on-set is completely covered by the combination (OR) of the subcubes of lower dimensionality - note that "111" is covered three times

Cout = BCin+AB+ACin

Higher dimensional cubes

- Sub-cubes of higher dimension than 2



F(A,B,C) = Σm(4,5,6,7)

on-set forms a square
i.e., a cube of dimension 2

represents an expression in one variable
i.e., 3 dimensions – 2 dimensions

A is asserted (true) and unchanged
B and C vary

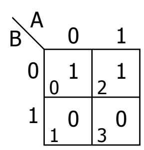
This subcube represents the literal A

m-dimensional cubes in a n-dimensional Boolean space

- In a 3-cube (three variables):
  - a 0-cube, i.e., a single node, yields a term in 3 literals
  - a 1-cube, i.e., a line of two nodes, yields a term in 2 literals
  - a 2-cube, i.e., a plane of four nodes, yields a term in 1 literal
  - a 3-cube, i.e., a cube of eight nodes, yields a constant term "1"
- In general,
  - an m-subcube within an n-cube (m < n) yields a term with n – m literals

Karnaugh maps

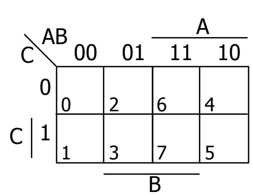
- Flat map of Boolean cube
  - wrap–around at edges
  - hard to draw and visualize for more than 4 dimensions
  - virtually impossible for more than 6 dimensions
- Alternative to truth-tables to help visualize adjacencies
  - guide to applying the uniting theorem
  - on-set elements with only one variable changing value are adjacent unlike the situation in a linear truth-table

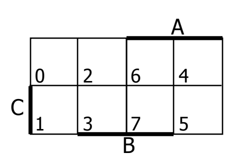


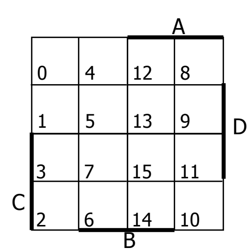
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

Karnaugh maps (cont'd)

- Numbering scheme based on Gray–code
  - e.g., 00, 01, 11, 10
  - only a single bit changes in code for adjacent map cells



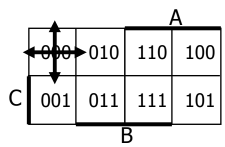


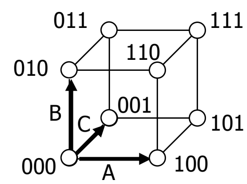


13 = 1101= ABC'D

Adjacencies in Karnaugh maps

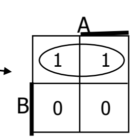
- Wrap from first to last column
- Wrap top row to bottom row



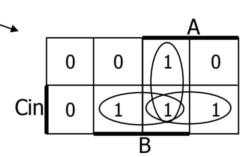


Karnaugh map examples

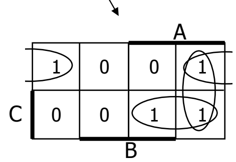
- F =
- Cout =
- f(A,B,C) = Σm(0,4,5,7)



B′



AB+ ACin + BCin



$AC + B'C' + \cancel{AB'}$

obtain the complement of the function by covering 0s with subcubes

More Karnaugh map examples

0	0	1	1
0	0	1	1

G(A,B,C) = A

1	0	0	1
0	0	1	1

F(A,B,C) = Σm(0,4,5,7) = AC + B'C'

0	1	1	0
1	1	0	0

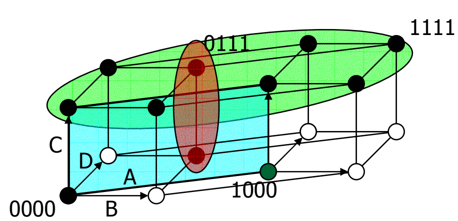
F' simply replace 1's with 0's and vice versa
F'(A,B,C) = Σ m(1,2,3,6)= BC' + A'C

Karnaugh map: 4-variable example

- F(A,B,C,D) = Σm(0,2,3,5,6,7,8,10,11,14,15)

F = C + A'BD + B'D'

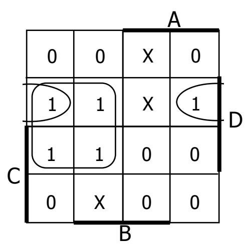
1	0	0	1
0	1	0	0
1	1	1	1
1	1	1	1



find the smallest number of the largest possible subcubes to cover the ON-set (fewer terms with fewer inputs per term)

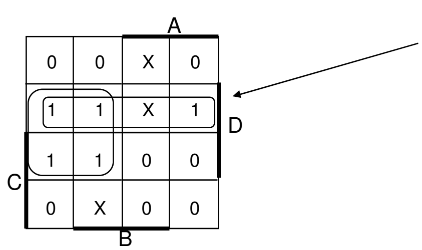
Karnaugh maps: don’t cares

- f(A,B,C,D) = Σ m(1,3,5,7,9) + d(6,12,13)
  - without don't cares
    - f = A′D + B′C′D



Karnaugh maps: don’t cares (cont’d)

- f(A,B,C,D) = Σ m(1,3,5,7,9) + d(6,12,13)
  - f = A'D + B'C'D without don't cares
  - f = A'D + C'D with don't cares



by using don't care as a "1"
a 2-cube can be formed
rather than a 1-cube to cover
this node

<u>don't cares</u> can be treated as
1s or 0s
depending on which is more
advantageous

Activity

- Minimize the function $F = \Sigma\, m(0, 2, 7, 8, 14, 15) + d(3, 6, 9, 12, 13)$

Combinational logic summary

- Logic functions, truth tables, and switches
  - NOT, AND, OR, NAND, NOR, XOR, . . ., minimal set
- Axioms and theorems of Boolean algebra
  - proofs by re-writing and perfect induction
- Gate logic
  - networks of Boolean functions and their time behavior
- Canonical forms
  - two-level and incompletely specified functions
- Simplification
  - a start at understanding two-level simplification
- Later
  - automation of simplification
  - multi-level logic
  - time behavior
  - hardware description languages
  - design case studies