Binary Arithmetic

Single Bit Binary Addition

Given two binary digits (X,Y), we get the following sum (S) and carry (C):

X	0	0	1	1
+Y	+0	+1	+0	+1
CS	00	01	01	10

Truth Table: Two Bit Adder

X	Y	Binary Sum (C)(S)
0	0	0 0
0	1	0 1
1	0	0 1
1	1	1 0

CARRY SUM

Truth Tables of Logical Operations

- Truth tables are used to show/define the relationships between the truth values of
  - the individual propositions and
  - the compound propositions based on them

p	q	p∧q	p∨q	p⊕q	p⇒q	p⇔q
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	0
1	1	1	1	0	1	1

Single Bit Binary Addition with Carry

Given two binary digits (X,Y), a carry in (Z) we get the following sum (S) and carry (C):

Carry in (Z) of 0:

Z	0	0	0	0
X	0	0	1	1
+Y	+0	+1	+0	+1
CS	00	01	01	10

Carry in (Z) of 1:

Z	1	1	1	1
X	0	0	1	1
+Y	+0	+1	+0	+1
CS	01	10	10	11

Full Adder: Include Carry Input

Z	Y	X	S = X + Y + Z	
			Decimal value	Binary value
0	0	0	0	0 0
0	0	1	1	0 1
0	1	0	1	0 1
0	1	1	2	1 0
1	0	0	1	0 1
1	0	1	2	1 0
1	1	0	2	1 0
1	1	1	3	1 1

CARRY SUM

Truth Table: Full Adder

Z	Y	X	Binary value (C)(S)
0	0	0	0 0
0	0	1	0 1
0	1	0	0 1
0	1	1	1 0
1	0	0	0 1
1	0	1	1 0
1	1	0	1 0
1	1	1	1 1

CARRY SUM

Multiple Bit Binary Addition

- **Extending this to a multiple bit examples:**

Carries	00000
Augend	01100
Addend	10001
Sum	11101

Single Bit Binary Subtraction

- **Given two binary digits (X,Y), we get the following difference (S) and borrow (B)**

X	0	0	1	1
-Y	-0	-1	-0	-1
BS	00	11	01	00

Borrow

Truth Table: Two Bit Subtractor

X	Y	Binary Difference (B)(D)
0	0	0 0
0	1	1 1
1	0	0 1
1	1	0 0

BORROW DIFFERENCE

Single Bit Binary Subtraction with Borrow

- **Given two binary digits (X,Y), a borrow in (Z) we get the following difference (D) and borrow (B):**
- **Borrow in (Z) of 0:**

Z	0	0	0	0
X	0	0	1	1
- Y	-0	-1	-0	-1
BD	0 0	1 1	0 1	0 0

- **Borrow in (Z) of 1:**

Z	1	1	1	1
X	0	0	1	1
- Y	-0	-1	-0	-1
BD	11	1 0	0 0	1 1

Truth Table: Full Subtractor

Z	Y	X	Binary value (C)(D)
0	0	0	0 0
0	0	1	1 1
0	1	0	0 1
0	1	1	0 0
1	0	0	1 1
1	0	1	1 0
1	1	0	0 0
1	1	1	1 1

CARRY DIFFERENCE

Multiple Bit Binary Subtraction

- **Extending this to a multiple bit example:**
- **Notes:**
  - **The 0 is a Borrow-In to the least significant bit.**
  - **If the Subtrahend > the Minuend, interchange and append a – to the result.**

Borrows	00000
Minuend	10110
Subtrahend	10010
Difference	00100

Multiple Bit Binary Subtraction

- Extending this to a multiple bit examples:
- Notes: The 0 is a Borrow-In to the least significant bit. If the Subtrahend > the Minuend, interchange and append a – to the result.

Borrows	00110
Minuend	10110
Subtrahend	10011
Difference	00011

Binary Multiplication

The binary multiplication table is simple:

0 * 0 = 0 | 1 * 0 = 0 | 0 * 1 = 0 | 1 * 1 = 1

Extending multiplication to multiple digits:

Multiplicand	**1011**
Multiplier	**x 101**
Partial Products	**1011**
	0000 -
	1011 - -
Product	**110111**

Signed Magnitude?

- Use fixed length binary representation
- Use left-most bit (called *most significant bit* or MSB) for sign:
  - 0 for positive
  - 1 for negative
- Example: $+18_{ten} = 00010010_{two}$
  $-18_{ten} = 10010010_{two}$

Difficulties with Signed Magnitude

- Sign and magnitude bits should be differently treated in arithmetic operations.
- Addition and subtraction require different logic circuits.
- Overflow is difficult to detect.
- "Zero" has two representations:
  - $+0_{ten} = 00000000_{two}$
  - $-0_{ten} = 10000000_{two}$
- *Signed-integers are not used in modern computers.*

Thank You

Logic:Expression

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

CS-230: Digital Logic Design & Computer Architecture

Lecture 3 (10 January 2022)

CADSL

Signed Number System

Signed Magnitude?

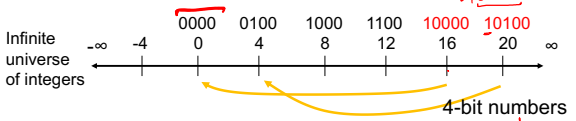
- Use fixed length binary representation
- Use left-most bit (called *most significant bit* or MSB) for sign:
  - 0 for positive
  - 1 for negative
- Example: $+18_{ten} = 00010010_{two}$
  $-18_{ten} = 10010010_{two}$

Difficulties with Signed Magnitude

- Sign and magnitude bits should be differently treated in arithmetic operations.
- Addition and subtraction require different logic circuits.
- Overflow is difficult to detect.
- "Zero" has two representations:
  $+0_{ten} = 00000000_{two}$
  $-0_{ten} = 10000000_{two}$
- *Signed-integers are not used in modern computers.*

Problems with Finite Math

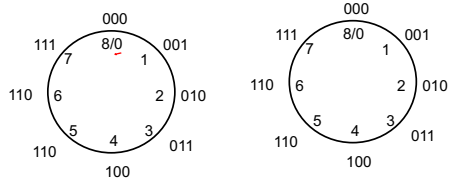
- Finite size of representation:
  - Digital circuit cannot be arbitrarily large.
  - Overflow detection – easy to determine when the number becomes too large.



- Represent negative numbers:
  - Unique representation of 0.

3-bit Universe

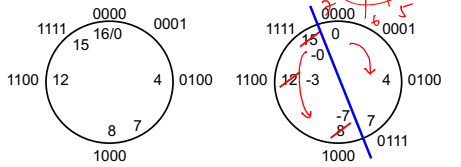
Modulo-8 (3-bit) universe



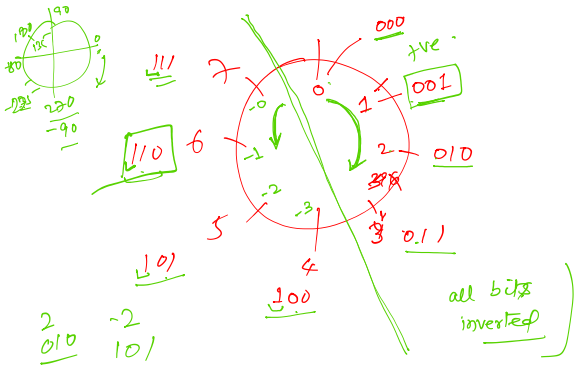
Only 8 integers: 0 through 7, or – 3 through 3

4-bit Universe

Modulo-16 (4-bit) universe

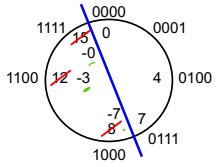


Only 16 integers: 0 through 15, or – 7 through 7



0 → 1
1 → 0

One Way to Divide Universe
1's Complement Numbers

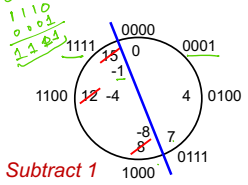


Decimal magnitude	Binary number	
	Positive	Negative
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000

Negation rule: invert bits.

Problem: 0 ≠ – 0

Another Way to Divide Universe
2's Complement Numbers



Subtract 1 on this side

Decimal magnitude	Binary number	
	Positive	Negative
0	0000	
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8		1000

Negation rule: invert bits and add 1

Integers With Sign – Two Ways

- Use fixed-length representation, but no explicit sign bit:
  - 1's complement: To form a negative number, complement each bit in the given number.
  - 2's complement: To form a negative number, start with the given number, subtract one, and then complement each bit, *or*

    first complement each bit, and then add 1.
- 2's complement is the preferred representation.

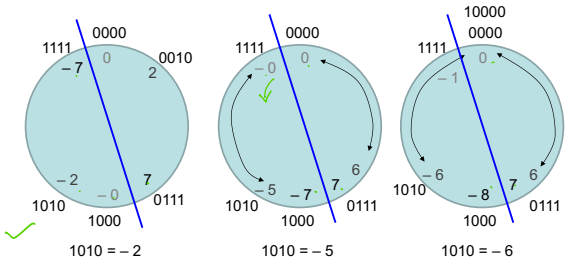
2's-Complement Integers

- Why not 1's-complement? *Don't like two zeros.*
- Negation rule:
  - Subtract 1 and then invert bits, or
  - Invert bits and add 1
- Some properties:
  - Only one representation for 0
  - Exactly as many positive numbers as negative numbers
  - Slight asymmetry – there is one negative number with no positive counterpart

General Method for Binary Integers with Sign

- Select number (n) of bits in representation.
- Partition 2^n integers into two sets:
  - 00…0 through 01…1 are $2^n/2$ positive integers.
  - 10…0 through 11…1 are $2^n/2$ negative integers.
- Negation rule transforms negative to positive, and vice-versa:
  - Signed magnitude: invert MSB (most significant bit)
  - 1's complement: Subtract from $2^n - 1$ or 1…1 (same as "inverting all bits")
  - 2's complement: Subtract from 2^n or 10…0 (same as 1's complement + 1)

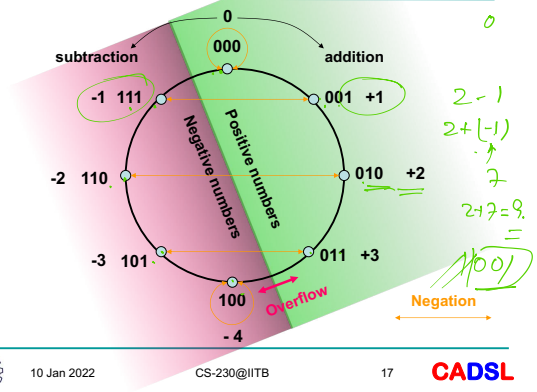
Three Systems (n = 4)



Three Representations

Sign-magnitude	1's complement	2's complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = - 0	100 = - 3	100 = - 4
101 = - 1	101 = - 2	101 = - 3
110 = - 2	110 = - 1	110 = - 2
111 = - 3	111 = - 0	111 = - 1
		(Preferred)

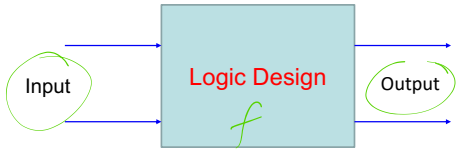
2's Complement Numbers (n = 3)



Summary

- For a given number (n) of digits we have a finite set of integers. For example, there are 10^3 = 1,000 decimal integers and 2^3 = 8 binary integers in 3-digit representations.
- We divide the finite set of integers [0, $r^n - 1$], where radix r = 10 or 2, into two equal parts representing positive and negative numbers.
- Positive and negative numbers of equal magnitudes are complements of each other: x + complement (x) = 0.

Digital System



output = f(input) {0,1}

I/O Behaviour: Automobile Ignition

- Engine turns on when
- Ignition key is applied AND
  - either Car is in parking gear OR
  - Brake pedal is on
- AND
  - either Seat belt is fastened OR
  - Car is in parking gear

Truth Table: Half Adder

X	Y	Binary Sum (C)(S)
0	0	0 0
0	1	0 1
1	0	0 1
1	1	1 0

CARRY SUM

Truth Table: Full Adder

Z	Y	X	Binary value (C)(S)
0	0	0	0 0
0	0	1	0 1
0	1	0	0 1
0	1	1	1 0
1	0	0	0 1
1	0	1	1 0
1	1	0	1 0
1	1	1	1 1

CARRY SUM

Truth Table: Full Adder

Z	Y	X	Carry C	Sum 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

Truth Tables of logical functions

- Truth tables are used to show/define the relationships between the truth values of
  - the individual propositions and
  - the compound propositions based on them

p	q	p.q	P+q	p⊕q	p⇒q	p⇔q
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	0
1	1	1	1	0	1	1

Logic Expressions

Truth Table

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

Logic Expression

$F = \bar{X}.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.Y.\bar{Z}$

$X.Y.\bar{Z} + X.Y.Z$

- Logic expressions, truth tables describe the same function!
- Truth tables are unique; expressions are not. This gives flexibility in implementing functions.

Thank You

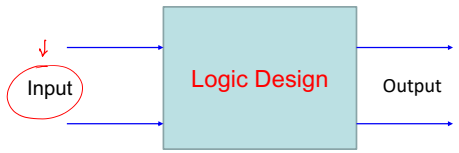
Logic: Implementation

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

CS-230: Digital Logic Design & Computer Architecture

Lecture 4 (11 January 2022)

Digital System



Logic Expressions

Truth Table	
X Y Z	F
0 0 0	1
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	1

Logic Expression

$F = \overline{X}.\overline{Y}.Z + X.\overline{Y}.\overline{Z} + X.\overline{Y}.Z$

$X.Y.\overline{Z} + X.Y.Z$

- Logic expressions, truth tables describe the same function!
- Truth tables are unique; expressions are not. This gives flexibility in implementing functions.

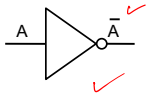
How Many Logic Functions?

- Output column of truth table has length 2^n for n input variables.
- It can be arranged in 2^{2^n} ways for n variables.
- Example: n = 1, single variable.

Input	Output functions			
A	F1(A)	F2(A)	F3(A)	F4(A)
0	0	0	1	1
1	0	1	0	1

NOT Gate

Truth Table	
A	$\overline{A}$
0	1
1	0

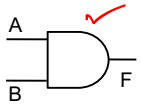


Symbol

Logic Function

AND Gate

Symbol

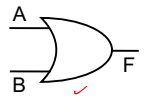


Logic Function

Truth Table		
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate

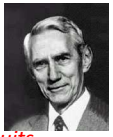
Symbol



Logic Function

Truth Table		
A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Shannon's Legacy

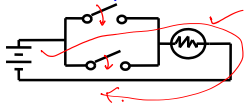


- *A Symbolic Analysis of Relay and Switching Circuits*, Master's Thesis, MIT, 1940. Perhaps the most influential master's thesis of the 20th century.
- *An Algebra for Theoretical Genetics,* PhD Thesis, MIT, 1940.
- Founded the field of Information Theory.
- C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication,* University of Illinois Press, 1949. A "must read."

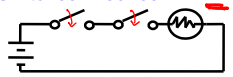
Logic Function Implementation

- Using Switches
  - For inputs:
    - logic 1 is switch closed
    - logic 0 is switch open
  - For outputs:
    - logic 1 is light on
    - logic 0 is light off.
  - NOT uses a switch such that:
    - logic 1 is switch open
    - logic 0 is switch closed

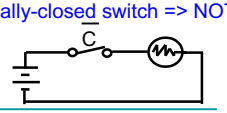
Switches in parallel => OR



Switches in series => AND



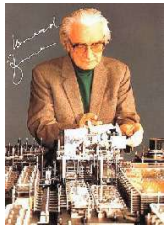
Normally-closed switch => NOT



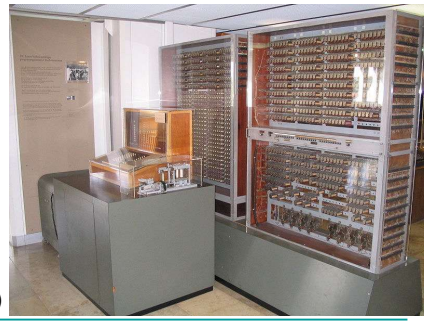
Switching Devices

- Electromechanical relays (1940s)
- Vacuum tubes (1950s)
- Bipolar transistors (1960 - 1980)
- Field effect transistors (1980 -)
- Integrated circuits (1970 -)

Relay Computers
Conrad Zuse (1910-1995)



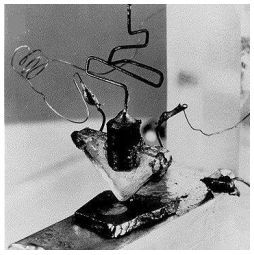
Z1 (1938)



Z3 (1941)

Transistor, 1948

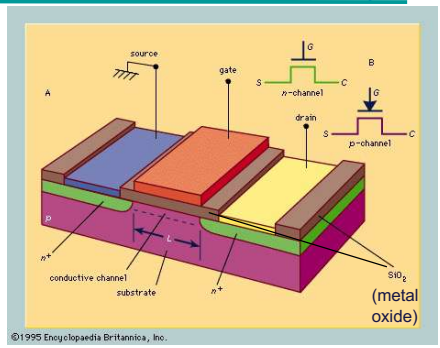
The thinker, the tinkerer, the visionary and the transistor
John Bardeen, Walter Brattain, William Shockley
Nobel Prize, 1956



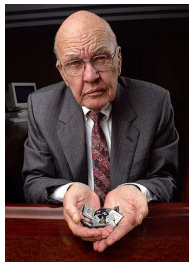
Bell Laboratories, Murray Hill, New Jersey

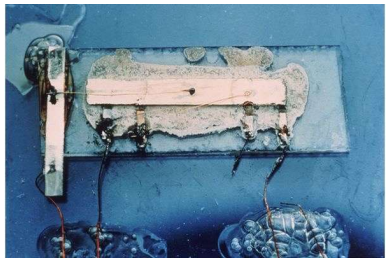
Field Effect Transistor (FET)

a.k.a.
metal oxide
semiconductor
(MOS) FET.



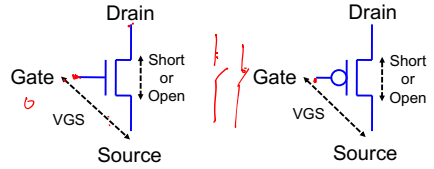
Integrated Circuit (1958)





Jack Kilby (1923-2005), Nobel Prize, 2000

MOSFET (Metal Oxide Semiconductor Field Effect Transistor)



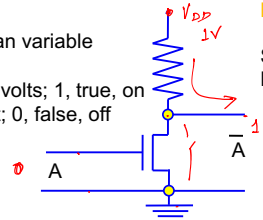
Reference:
R. C. Jaeger and T. N. Blalock, *Microelectronic Circuit Design, Third Edition*, McGraw Hill.

NMOSFET NOT Gate (Early Design)

Power supply
VDD volts
w.r.t. ground

A: Boolean variable

A = VDD volts; 1, true, on
A = 0 volt; 0, false, off



Problem: When A = 1, current leakage causes power dissipation.

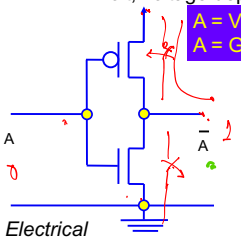
Solution: Complementary MOS design proposed by

F. M. Wanlass and C.-T. Sah, "Nanowatt Logic Using Field-Effect Metal-Oxide Semiconductor Triodes," *International Solid State Circuits Conference Digest of Technical Papers*, Feb 20, 1963, pp. 32-33.

CMOS NOT Gate (Modern Design)

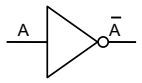
Power supply
VDD = 1 volt; voltage depends on technology.

A = VDD = 1 volt is state "1"
A = GND = 0 volt is state "0"

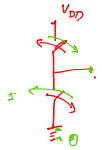


Truth Table	
A	Ā
0	1
1	0

Boolean Function



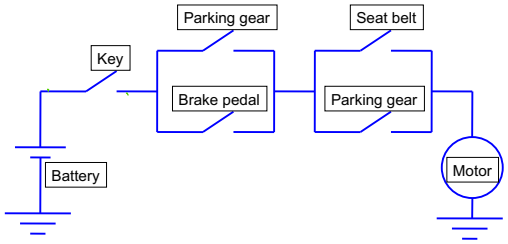
Symbol



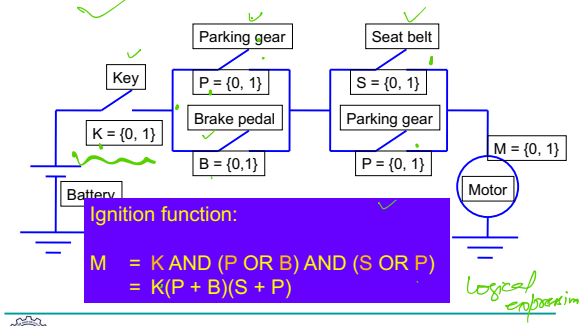
Example: Automobile Ignition

- Engine turns on when
- Ignition key is applied AND
  - Car is in parking gear OR
  - Brake pedal is on
- AND
  - Seat belt is fastened OR
  - Car is in parking gear

Switching logic

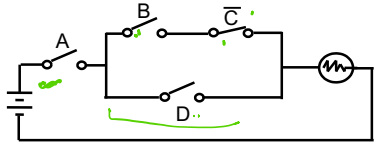


Define Variables



Logic Function Implementation

- Example: Logic Using Switches



- Light is on (L = 1) for
  L(A, B, C, D) = A ((B C') + D) = A B C' + A D
  and off (L = 0), otherwise.
- Useful model for relay circuits and for CMOS gate circuits, the foundation of current digital logic technology

Logic Expression

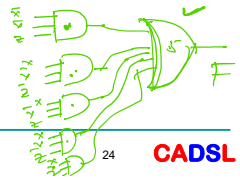
Truth Table

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

Logic Expression

$F = \bar{X}.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.Y.Z$

$X.Y.\bar{Z} + X.Y.Z$



Logic Expressions

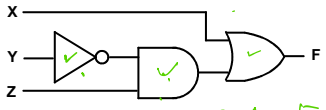
Truth Table

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

Equation

$F = X + \bar{Y}.Z$

Logic Diagram



Digital Logic Design

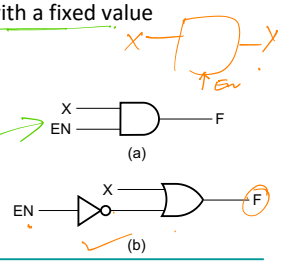
- Express input output relationship using Truth table
- Generate the logical expression by disjunction (OR) terms (conjunction of variables – AND) where system evaluates to true
- Replace all operators by the logic gates
- Replace logic gates by its transistor level circuit

Common Functions

Enabling Function

- *Enabling* permits an input signal to pass through to an output
- *Disabling* blocks an input signal from passing through to an output, replacing it with a fixed value
- When disabled, 0 output
- When disabled, 1 output

EN	X	Y
0	0	0
0	1	0
1	0	0
1	1	1



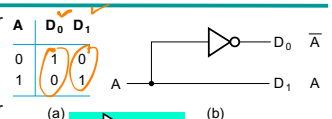
Decoding Function

- Decoding - the
  - Conversion of *n*-bit input to *m*-bit output
  - Given $n \le m \le 2^n$
- Circuits that perform decoding are called *decoders*
  - Called *n*-to-*m* line decoders, where $m \le 2^n$, and
  - Generate 2^n (or fewer) 1's in output for the *n* input variables

Decoder

- 1-to-2-Line Decoder

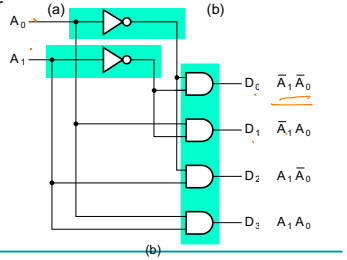
A	D_0	D_1
0	1	0
1	0	1



- 2-to-4-Line Decoder

A_1	A_0	D_0	D_1	D_2	D_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

(a)



Encoding Function

- Encoding - the opposite of decoding
  - Conversion of *m*-bit input to *n*-bit output
- Circuits that perform encoding are called *encoders*
  - An encoder has 2^n (or fewer) input lines and *n* output lines which generate the binary code corresponding to the input values
  - Typically, an encoder converts a code containing exactly one bit that is 1 to a binary code corres-ponding to the position in which the 1 appears.

Encoder

- A decimal-to-BCD encoder
  - Inputs: 10 bits corresponding to decimal digits 0 through 9, (D_0, …, D_9)
  - Outputs: 4 bits with BCD codes
  - Function: If input bit D_i is a 1, then the output (A_3, A_2, A_1, A_0) is the BCD code for i,
- The truth table could be formed, but alternatively, the equations for each of the four outputs can be obtained directly.

Encoder

- Input D_i is a term in equation A_j if bit A_j is 1 in the binary value for i.
- Equations:

$A_3 = D_8 + D_9$

$A_2 = D_4 + D_5 + D_6 + D_7$

$A_1 = D_2 + D_3 + D_6 + D_7$

$A_0 = D_1 + D_3 + D_5 + D_7 + D_9$

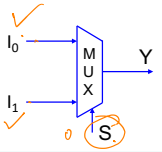
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001

Selection Function

- Selecting of data or information is a critical function in digital systems and computers
- Circuits that perform selecting have:
  - A set of information inputs from which the selection is made
  - A single output
  - A set of control lines for making the selection
- Logic circuits that perform selecting are called multiplexers

Multiplexers

- A multiplexer selects one input line and transfers it to output
  - *n* control inputs (S_{n-1}, … S_0) called *selection inputs*
  - $m \leq 2^n$ information inputs (I_{2^n-1}, … I_0)
  - output Y



2-to-1-Line Multiplexer

- Since $2 = 2^1$, n = 1
- The single selection variable S has two values:
  - S = 0 selects input I_0
  - S = 1 selects input I_1
- Truth Table
- Symbolic equation:

$Y = I_0 . \bar{S} + S . I_1$

- Logic expression

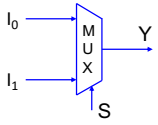
$Y = \bar{S}.I_0.\bar{I}_1 + \bar{S}.I_0.I_1 + S.\bar{I}_0.I_1 + S.I_0.I_1$

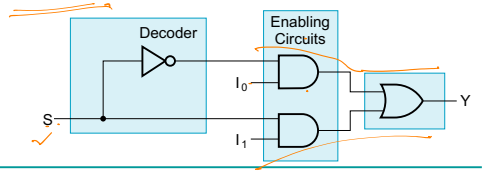
S	I_0	I_1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

2-to-1-Line Multiplexer

- The single selection variable S has two values:
  - S = 0 selects input I_0
  - S = 1 selects input I_1
- The logic equation:

$Y = I_0\bar{S} + S.I_1$





Thank You

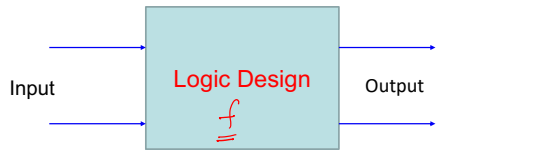
Logic: Implementation

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

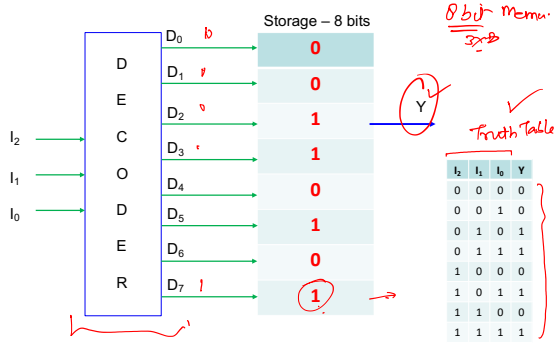
CS-230: Digital Logic Design & Computer Architecture

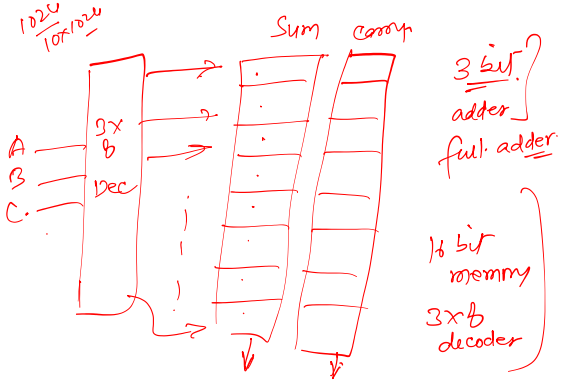
Lecture 4 (13 January 2022)

Digital System



Using Storage Elements





Optimization

Specification: Logic Function

Truth Table

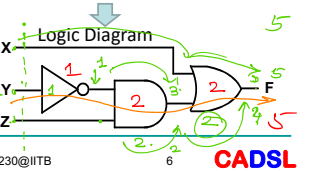
X Y Z	F
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	0
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	1

Logic Expression

$F = \bar{X}.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.Y.Z + X.\bar{Y}.Z + X.Y.Z$

$F = X + \bar{Y}.Z$

Logic Diagram



Implementation

Logical expression ⇒ not unique.

minimize logical expression

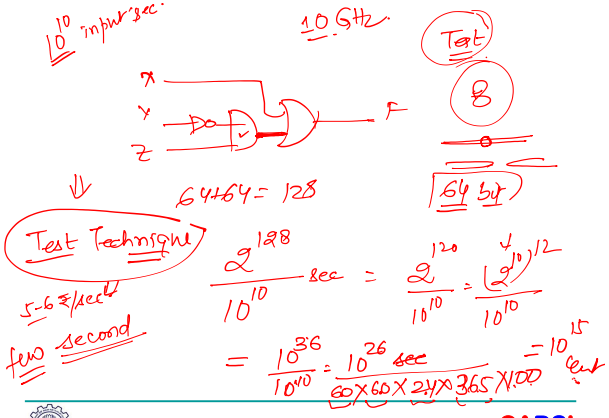
Parameters.

cost

Optimization Parameters

- Area: # Switches (Gates)
- Performance (Delay): # Switches in series
- Power: # Switches
- Testability: Interconnect network
- Security
- Intelligence

PPTASI



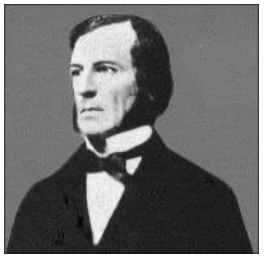
ALGEBRA

Algebra

- Algebra is defined as
  1. Set of elements
  2. Set of operators
  3. Number of postulates
- A set of elements is any collection of objects having common properties

  $S = \{a,b,c,d\};\ a \in S, e \notin S$
- A binary operator $*$ defined on a set S of elements is a rule that assigns each pair from S to a unique pair from S. $a * b = c$

George Boole (1815-1864)



- Born, Lincoln, England
- Professor of Math., Queen's College, Cork, Ireland
- Book, *The Laws of Thought*, 1853

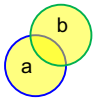
Boolean Algebra

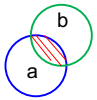
- Boolean Algebra is defined as
  1. Set of elements {0, 1}
  2. Set of operators {+, ·, ~}
  3. Number of postulates
- Boolean Algebra: 5-tuple

  $\{B, +, ., \sim, 0, 1\}$
- Closure: If a and b are Boolean then $(a.b)$ and $(a + b)$ are also Boolean

Postulate 1: Commutativity

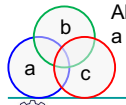
- Binary operators + and · are commutative.
- That is, for any elements *a* and *b* in *B*:
  - $a + b = b + a$
  - $a \cdot b = b \cdot a$





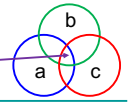
Postulate 2: Associativity

- Binary operators + and · are associative.
- That is, for any elements a, b and c in B:
  - a + (b + c) = (a + b) + c
  - a · (b · c) = (a · b) · c
- Example: EE department has three courses with student groups a, b and c



All EE students:
a + (b + c)

EE students in all EE courses: a · (b · c)



Postulate 3: Identity Elements

- There exist 0 and 1 elements in B, such that for every element a in B
  - a + 0 = a
  - a · 1 = a
- Definitions:
  - 0 is the identity element for + operation
  - 1 is the identity element for · operation
- Remember, 0 and 1 here should not be misinterpreted as 0 and 1 of ordinary algebra.

Postulate 5: Distributivity

- Binary operator + is distributive over · and · is distributive over +.
- That is, for any elements a, b and c in K:
  - a + (b · c) = (a + b) · (a + c)
  - a · (b + c) = (a · b) + (a · c)
- Remember dot (·) operation is performed before + operation:

  a + b · c = a + (b · c) ≠ (a + b) · c

Postulate 6: Complement

- A unary operation, *complementation*, exists for every element of B.
- That is, for any element a in B:

$$a + \bar{a} = 1$$
$$a \cdot \bar{a} = 0$$

- Where, 1 is identity element for ·

  0 is identity element for +

The Duality Principle

- Each postulate of Boolean algebra contains a pair of expressions or equations such that one is transformed into the other and vice-versa by interchanging the operators, + ↔ ·, and identity elements, 0 ↔ 1.
- The two expressions are called the duals of each other.

Theorems 1

Theorem 1: Idempotency

For all elements a in B: $a + a = a$; $a \cdot a = a$.

Theorem 2: Existance of Null Element

- $a + 1 = 1$, for + operator.
- $a \cdot 0 = 0$, for · operator.

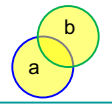
Theorem 3: Involution holds

- $\bar{\bar{a}} = a$

Theorem 4: Absorption

- $a + a.b = a$
- $a.(a + b) = a$
- Proof: $a + a\,b = a.1 + a.b$ (identity element)
  $= a.(1 + b)$ (distributivity)
  $= a.1$ (Theorem 2)
  $= a$ (identity element)

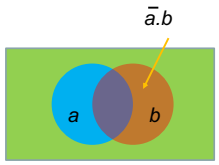
Similar proof for a (a + b) = a.



Theorems: Adsorption & Uniting

- Theorem 5: Adsorption

$$a + \bar{a}b = a + b$$
$$a(\bar{a} + b) = ab$$



- Theorem 6: Uniting

$$ab + a\bar{b} = a$$
$$(a + b)(a + \bar{b}) = a$$

Theorem 7: DeMorgan's Theorem

- $\overline{(a + b)} = \bar{a} \cdot \bar{b}, \quad \forall a, b \in B$
- $\overline{(a \cdot b)} = \bar{a} + \bar{b}, \quad \forall a, b \in B$

1806 - 1871

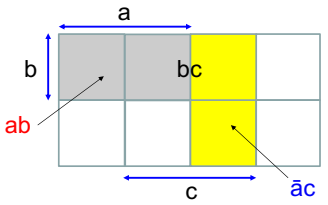
Generalization of DeMorgan's Theorem:

$$\overline{a + b + \cdots + z} = \bar{a} \cdot \bar{b} \cdots \bar{z}$$
$$\overline{a \cdot b \cdots z} = \bar{a} + \bar{b} + \cdots + \bar{z}$$

Theorem 8: Consensus

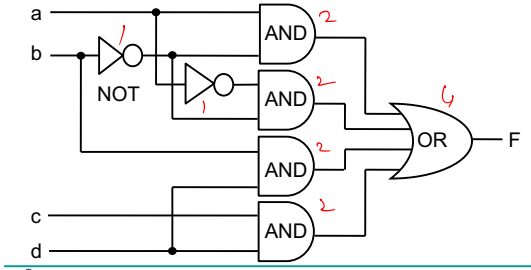
$$ab + \bar{a}c + bc = ab + \bar{a}c$$
$$(a + b)(\bar{a} + c)(b + c) = (a + b)(\bar{a} + c)$$

Lecture 5&6



Understanding Minimization

- Logic function: $F = a\bar{b} + \bar{a}\bar{b} + bd + cd$



Logic Minimization

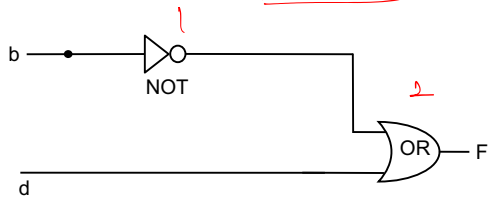
- Reducing products:

$F = a\bar{b} + \bar{a}\bar{b} + bd + cd$	
$= \bar{b}(a + \bar{a}) + bd + cd$	Distributivity
$= \bar{b}1 + bd + cd$	Complementation
$= \bar{b}(c + \bar{c}) + bd + cd$	Identity
$= bd + \bar{b}c + cd + \bar{b}\bar{c}$	Distributivity
$= bd + \bar{b}c + \bar{b}\bar{c}$	Consensus theorem
$= bd + \bar{b}(c + \bar{c})$	Distributivity
$= bd + \bar{b} \cdot 1$	Complement, identity
	Adsorption

$F = \bar{b} + d$

Logic Minimization

- Minimized expression: $F = \bar{b} + d$



Thank You

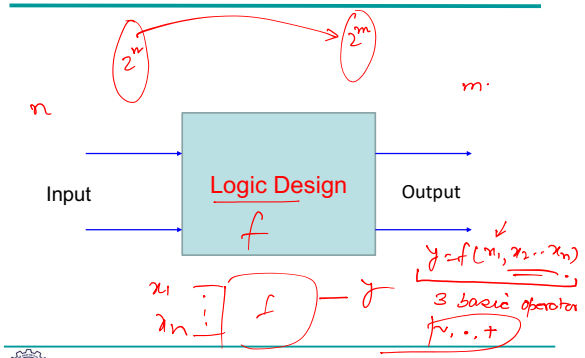
Logic: Implementation

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

CS-230: Digital Logic Design & Computer Architecture

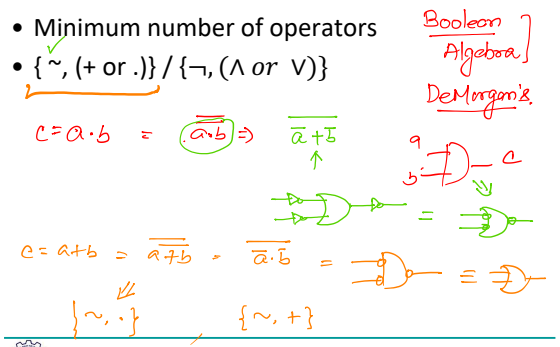
Lecture 6 (17 January 2022)

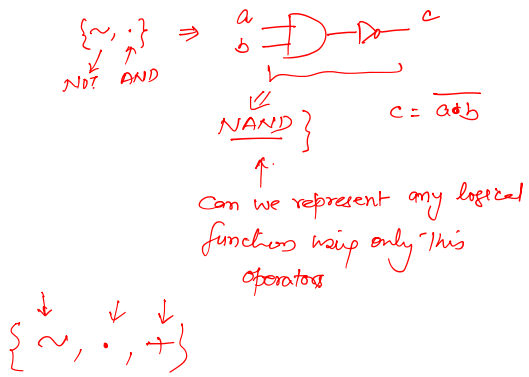
Digital System



Minimum Operator Set

- Minimum number of operators
- {~, (+ or .)} / {¬, (∧ *or* ∨)}

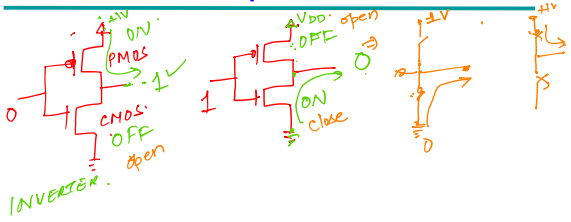


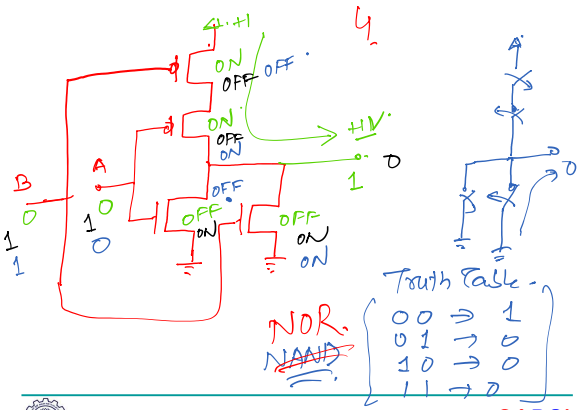


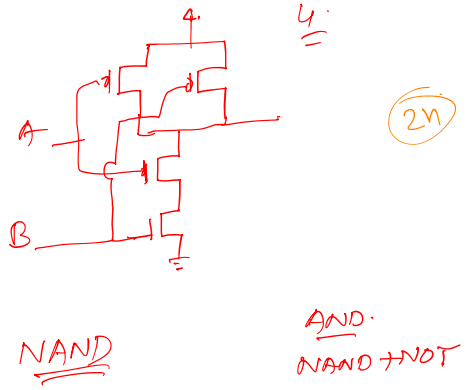
Universal Operator: NAND

- NAND: Composite operator (AND and NOT)
- $\overline{a} = \overline{a \,.\, a}$
- $a \,.\, b = \overline{\overline{a \,.\, b}} = \overline{(a \,.\, b)}\,\overline{(a \,.\, b)}$
- $a + b = \overline{\overline{a} \,.\, \overline{b}} = \overline{(a \,.\, a)}\,\overline{(b \,.\, b)}$

Universal Operator: NAND



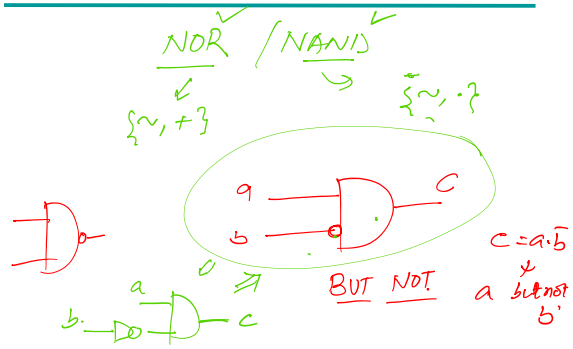




Universal Operator: NOR

- NOR: Composite operator (OR and NOT)
- $\overline{a} = \overline{a + a}$
- $a + b = \overline{\overline{a + b}} = \overline{(a + b)} + \overline{(a + b)}$
- $a \,.\, b = \overline{\overline{a} + \overline{b}} = \overline{(a + a)} + \overline{(b + b)}$

Universal Operator: NOR



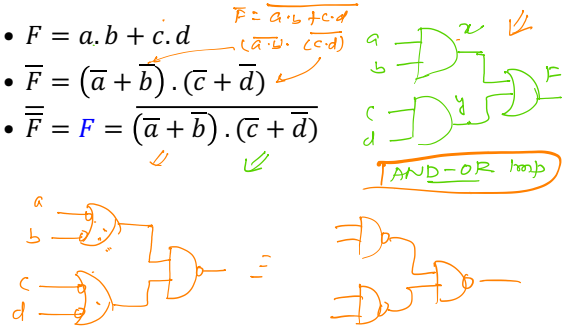
Complementing Functions

- Use DeMorgan's Theorem to complement a function:
  1. Interchange AND and OR operators
  2. Complement each constant value and literal
- Example: Complement $F = \bar{x}.y.\bar{z} + x.\bar{y}.\bar{z}$

  $\bar{F} = (x + \bar{y} + z)(\bar{x} + y + z)$

Logic Expression (SOP)

- $F = a.b + c.d$
- $\overline{F} = (\overline{a} + \overline{b}) \,.\, (\overline{c} + \overline{d})$
- $\overline{\overline{F}} = F = \overline{(\overline{a} + \overline{b}) \,.\, (\overline{c} + \overline{d})}$

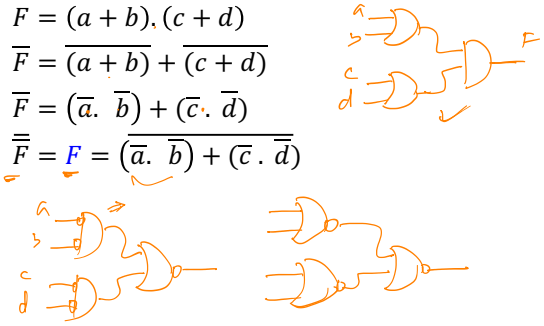


Logic Expression (SOP)

- $F = a.b + c.d$
- $F = \overline{(\overline{a} + \overline{b}) \,.\, (\overline{c} + \overline{d})}$

Logic Expression (POS)

- $F = (a + b).(c + d)$
- $\overline{F} = \overline{(a + b)} + \overline{(c + d)}$
- $\overline{F} = (\overline{a}.\ \overline{b}) + (\overline{c}\,.\,\overline{d})$
- $\overline{\overline{F}} = F = \overline{(\overline{a}.\ \overline{b}) + (\overline{c}\,.\,\overline{d})}$



Logic Expression (POS)

- $F = (a + b).(c + d)$
- $F = \overline{(\overline{a}.\ \overline{b}) + (\overline{c}\,.\,\overline{d})}$

Truth Table

Truth Table

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

Logic Expression

$F = \bar{X}.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.\bar{Y}.Z$

$X.Y.\bar{Z} + X.Y.Z$

Min Terms

Truth Table

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

$F = m_1 + m_4 + m_5 + m_6 + m_7$

$= \sum 1, 4, 5, 6, 7$

Max Terms

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

Thank You

Logic: Representation

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

CS-230: Digital Logic Design & Computer Architecture

Lecture 7 (18 January 2022)

Truth Table/ Min Term/Max Term

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

Truth Table/ Min Term/Max Term

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

Truth Table/ Min Term/Max Term

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

Logic Expression

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

Logic Expression

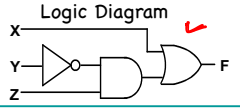
$F = \overline{X}.\overline{Y}.Z + X.\overline{Y}.\overline{Z} + X.\overline{Y}.Z$

$X.\overline{Y}.\overline{Z} + X.Y.Z$

Logic Expression Boolean Algebra

$F = X + \overline{Y}\,Z$

Logic Diagram



What to Minimize?

Logic Expression

$F = \overline{X}.\overline{Y}.Z + X.\overline{Y}.\overline{Z} + X.\overline{Y}.Z + X.Y.\overline{Z} + X.Y.Z$

Boolean Algebra

Logic Expression

$F = X + \overline{Y}\,Z$

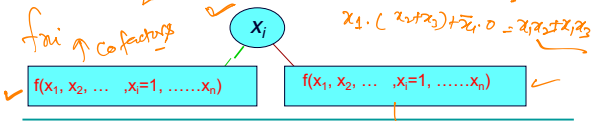
Graphical Method: Binary Decision Diagram

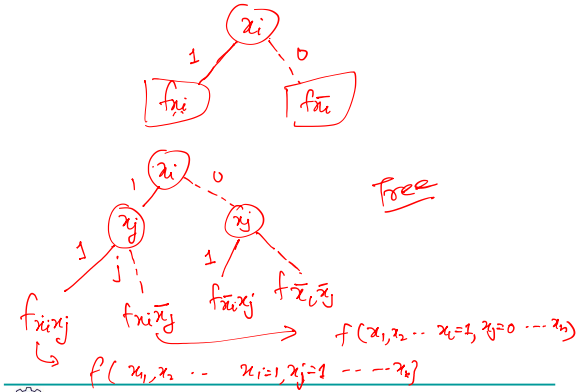
Binary Decision Diagram

- BDD is canonical form of representation
- Shanon's expansion theorem
- $f(x_1, x_2, ...x_i, ...x_n) =$

  $x_i . f(x_1, x_2, \ldots, x_i=1, ...x_n) +$

  $x_i' . f(x_1, x_2, \ldots, x_i=0, ...x_n)$



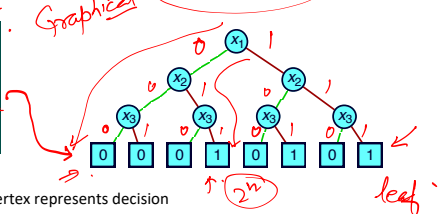


Decision Structures

Truth Table

x_1	x_2	x_3	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	0
1	1	1	1

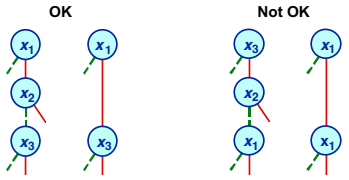
Decision Tree



- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value.

Variable Ordering

- Assign arbitrary total ordering to variables
  - e.g., $x_1 < x_2 < x_3$
- Variables must appear in ascending order along all paths

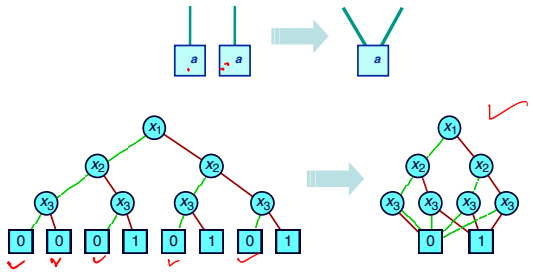


Properties

- No conflicting variable assignments along path
- Simplifies manipulation

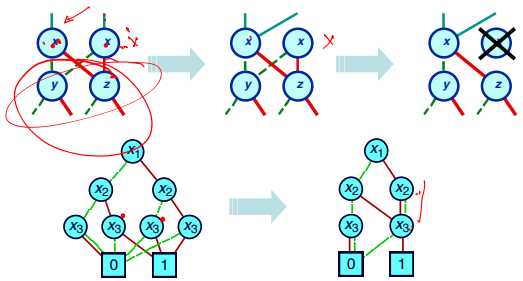
Reduction Rule #1

Merge equivalent leaves



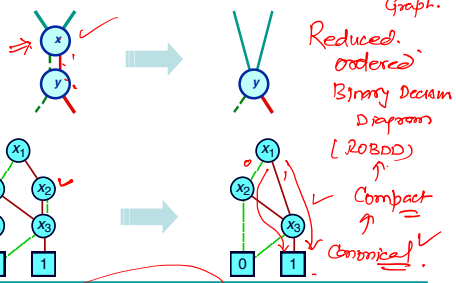
Reduction Rule #2

Merge isomorphic nodes

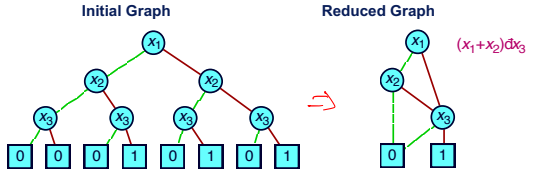


Reduction Rule #3

Eliminate Redundant Tests



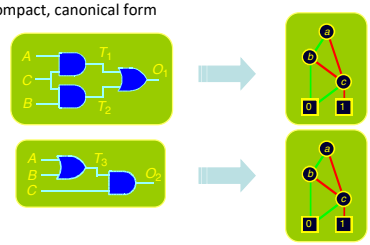
Example OBDD



- Canonical representation of Boolean function
  - For given variable ordering
  - Two functions equivalent if and only if graphs isomorphic
    - Can be tested in linear time
  - Desirable property: *simplest form is canonical.*

Binary Decision Diagram

- Generate Complete Representation of Circuit Function
  - Compact, canonical form



- Functions equal if and only if representations identical
- Never enumerate explicit function values
- Exploit structure & regularity of circuit functions

Example Functions

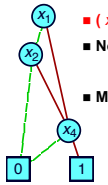
Constants

- 0 Unique unsatisfiable function
- 1 Unique tautology

Variable

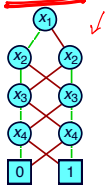
Treat variable as function

Typical Function

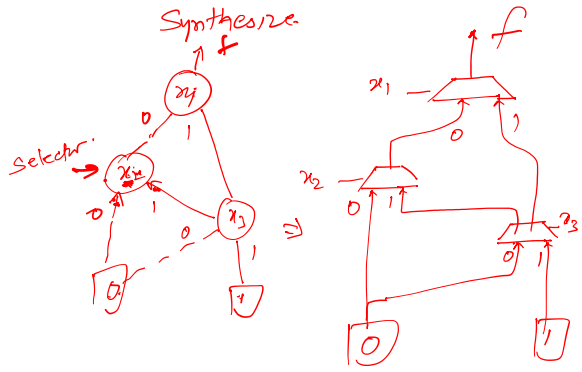


- $(x_1 + x_2) \cdot x_4$
- No vertex labeled x_3
  - independent of x_3
- Many subgraphs shared

Odd Parity



Linear representation



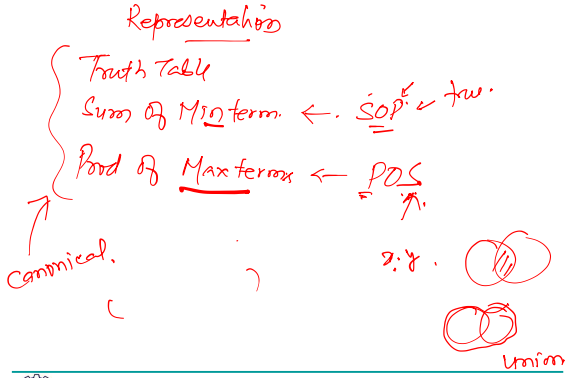
Thank You

Logic: Representation

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

CS-230: Digital Logic Design & Computer Architecture

Lecture 8 (20 January 2022)



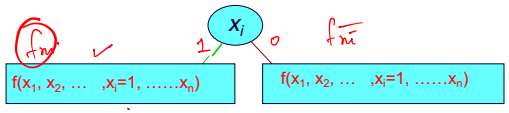
Graphical Method: Binary Decision Diagram

Binary Decision Diagram

- BDD is canonical form of representation
- Shanon's expansion theorem
- $f(x_1, x_2, ...x_i, ...x_n) =$

  $x_i.f(x_1, x_2, \ldots, x_i=1, ...x_n) +$

  $x_i'.f(x_1, x_2, \ldots, x_i=0, ...x_n)$

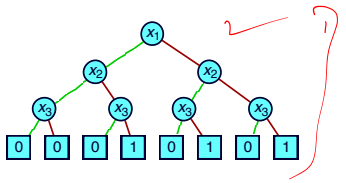


Decision Structures

Truth Table

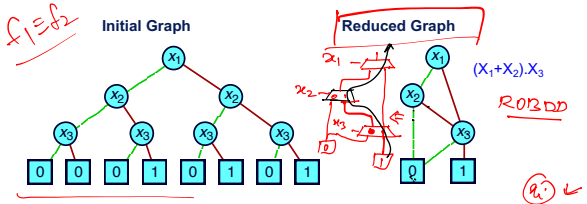
x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Decision Tree



- Vertex represents decision
- Follow green (dashed) line for value 0
- Follow red (solid) line for value 1
- Function value determined by leaf value.

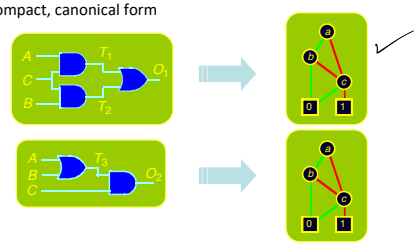
Example OBDD



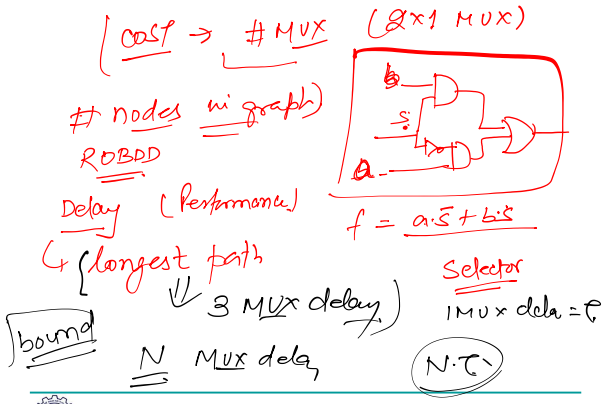
- Canonical representation of Boolean function
  - ❖ For given variable ordering
  - ➢ Two functions equivalent if and only if graphs isomorphic
    - o Can be tested in linear time
  - ➢ Desirable property: *simplest form is canonical*.

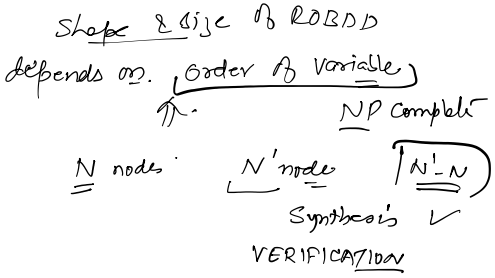
Binary Decision Diagram

- Generate Complete Representation of Circuit Function
  - Compact, canonical form

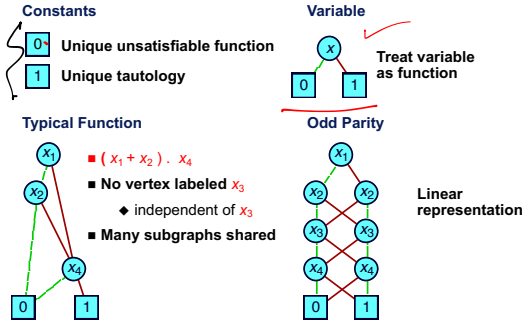


- ➢ Functions equal if and only if representations identical
- ➢ Never enumerate explicit function values
- ➢ Exploit structure & regularity of circuit functions



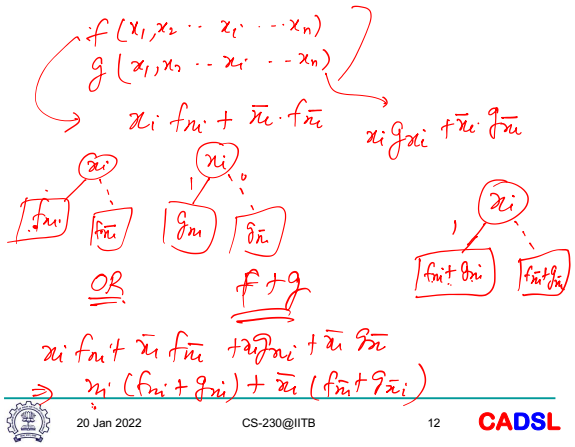


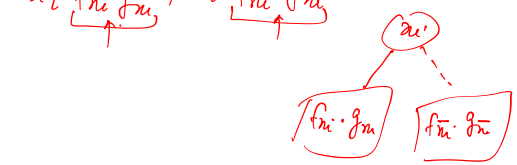
Example Functions



Operations with BDD

- ❖ Let v_1, v_2 denote root nodes of f_1, f_2 respectively , with $var(v_1) = x_1$ and $var(v_2) = x_2$
- ❖ If v_1 and v_2 are leafs, f_1 OP f_2 is a leaf node with value $val(v_1)$ OP $val(v_2)$

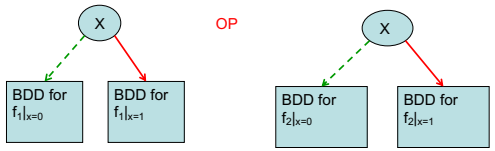




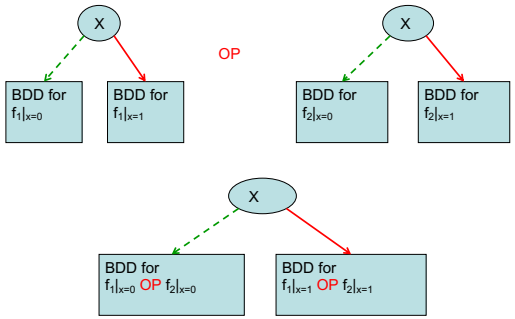
Operations with BDD

- ❖ If $x_1 = x_2 = x$, apply Shanon's expansion

f_1 OP $f_2 = x' . (f_1|_{x=0}$ OP $f_2|_{x=0}) + x . (f_1|_{x=1}$ OP $f_2|_{x=1})$



Operations with BDD



$f(x_1, x_2 \cdots x_n)$ $g(x_2 x_3 \cdots x_n)$

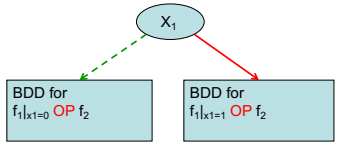
$(x_1 \cdot f_{x_1} + \bar{x_1} f_{\bar{x_1}}) \cdot g$

$x_1 \cdot (f_{x_1} \cdot g) + \bar{x_1} (f_{\bar{x_1}} \cdot g)$

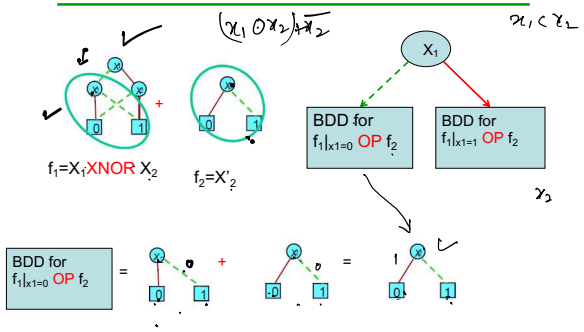
Operations with BDD

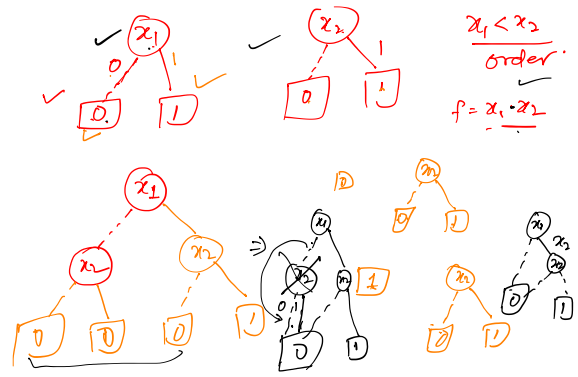
❖ Else suppose $x_1 < x_2 = x$, in variable order

f_1 OP $f_2 = x'_1 (f_1|_{x1=0}$ OP $f_2) + x_1 (f_1|_{x1=1}$ OP $f_2)$

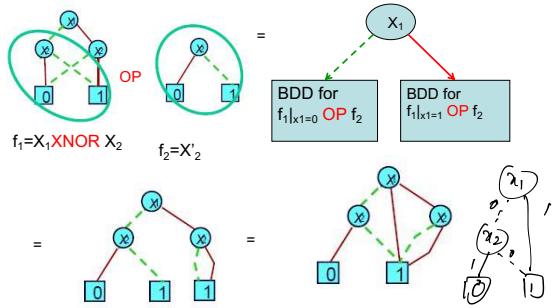


Operations with BDD: Example

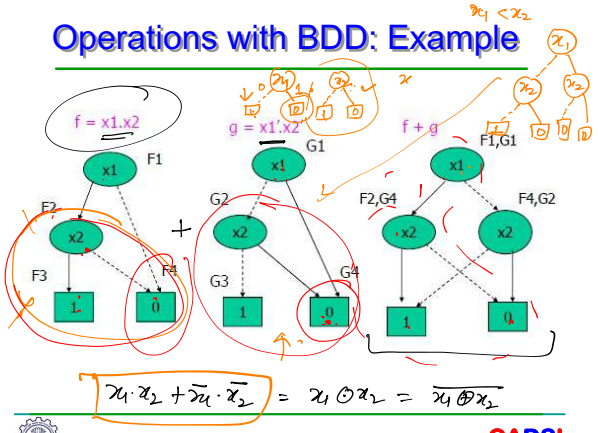




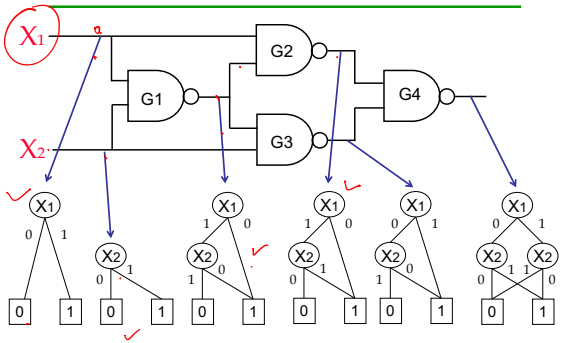
Operations with BDD: Example



Operations with BDD: Example



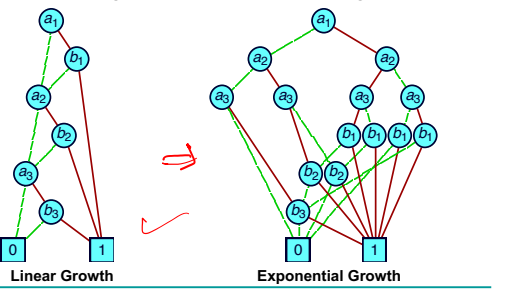
From Circuits to BDD



Effect of Variable Ordering

$(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$

$a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$



Selecting Good Variable Ordering

- Intractable Problem
  - Even when problem represented as OBDD
    - i.e., to find optimum improvement to current ordering
- Application-Based Heuristics
  - Exploit characteristics of application
  - e.g., Ordering for functions of combinational circuit
    - Traverse circuit graph depth-first from outputs to inputs
    - Assign variables to primary inputs in order encountered

Selecting Good Variable Ordering

- Static Ordering
  - Fan In Heuristic
  - Weight Heuristic
- Dynamic Ordering
  - Variable Swap
  - Window Permutation
  - Sifting

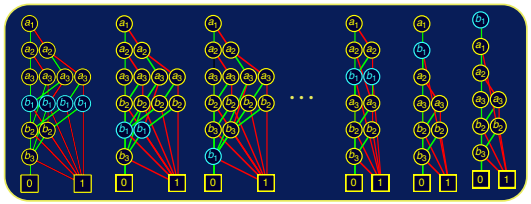
Dynamic Variable Reordering

- Richard Rudell, Synopsys
- Periodically Attempt to Improve Ordering for All BDDs
  - ❖Move each variable through ordering to find its best location
- Has Proved Very Successful
  - ❖Time consuming but effective

Dynamic Reordering By Sifting

- Choose candidate variable
- Try all positions in variable ordering
  - Repeatedly swap with adjacent variable
- Move to best position found

Best Choices



ROBDD Sizes & Variable Ordering

- Bad News
  - Finding optimal variable ordering NP-Hard
  - Some functions have exponential BDD size for all orders *e.g.* multiplier
- Good News ☺
  - Many functions/tasks have reasonable size ROBDDs
  - Algorithms remain practical up to 1,000,000 node OBDDs
  - Heuristic ordering methods generally satisfactory
- What works in Practice
  - Application-specific heuristics e.g. DFS-based ordering for combinational circuits
  - Dynamic ordering based on variable sifting *(R. Rudell)*

WXFRHU

Thank You

Logic Optimization

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

CS-230: Digital Logic Design & Computer Architecture

Lecture 9 (24 January 2022)

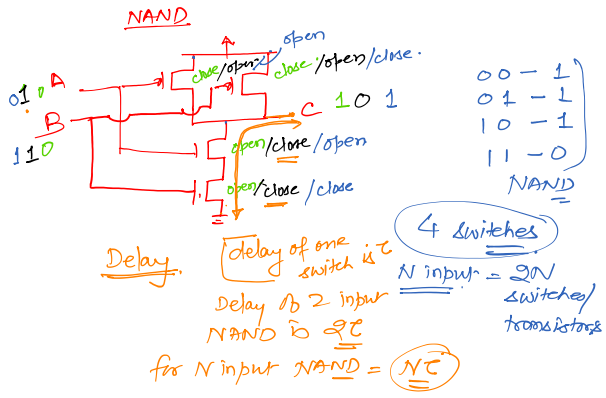
Representation
① Truth Table ② Sum of min terms ③ Product of Max terms — UNIQUE, Canonical
Graphical.
④ Binary Decision Diagram (ROBDD) — Canonical
⑤ Reed Muller Representation - canonical. (AND and XOR)
⑥ And Inverter Graph (AIG) — not canonical
ABC ⑦ SOP ⑧ POS — non canonical
⑨ Hybrid

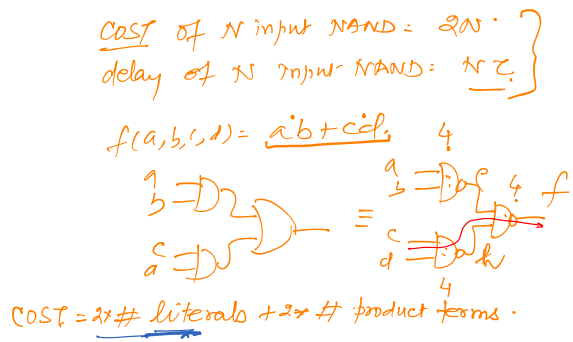
Function Minimization

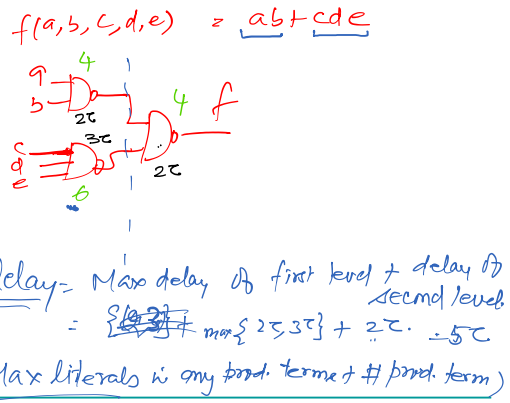
SOP → 2 level. ⊕ AND → OR ⊖ NAND-NAND
POS → 2 level OR → AND ⊖ NOR-NOR.

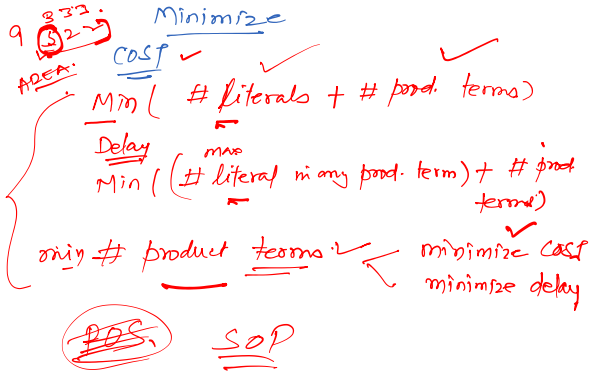
minimize → Boolean Algebra
⇒ Not scalable approach

Automatic
Algorithmic approach.
① Graphical Method. (K-Map)
② Tabular Method. (Q.M.)



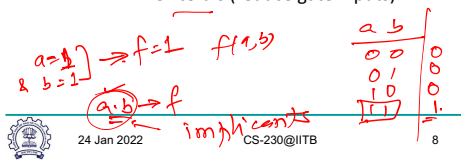






Logic Minimization

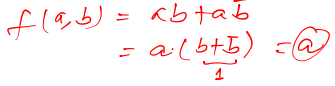
- Generally means
  - In SOP form:
    - Minimize number of products (reduce gates) and
    - Minimize literals (reduce gate inputs)
  - In POS form:
    - Minimize number of sums (reduce gates) and
    - Minimize literals (reduce gate inputs)



Product or Implicant or Cube

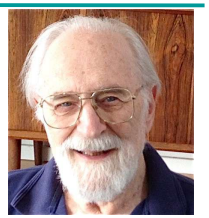
- Any set of literals ANDed together.
- Minterm is a special case where all variables are present. It is the largest product.
- A minterm is also called a 0-implicant of 0-cube.
- A 1-implicant or 1-cube is a product with one variable eliminated:
  - Obtained by combining two adjacent 0-cubes
  - $A\bar{B}CD + ABC\cdot\bar{D} = ABC(D + \bar{D}) = ABC$

Function Minimization



Graphical Method: Karnaugh Map

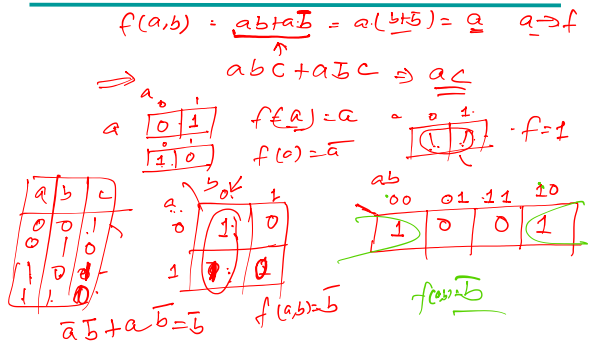
Maurice Karnaugh
- American Physicist
- Bell Lab (1952 – 66)
- Developed K-Map in 1954



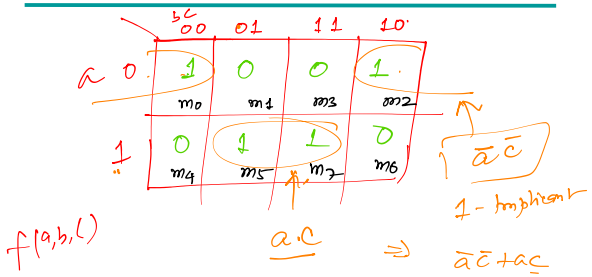
Maurice Karnaugh
Born: 4 October 1924

Karnaugh, Maurice (November 1953), "The Map Method for Synthesis of Combinational Logic Circuits" . *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*. **72**(5): 593–599

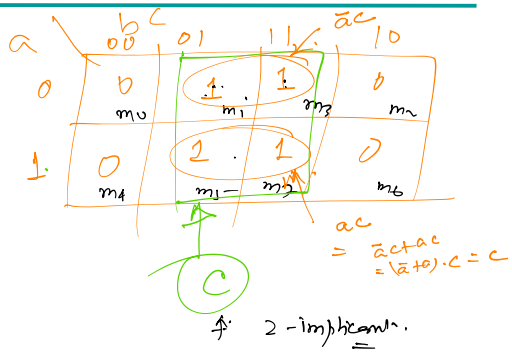
Function Minimization: K-Map



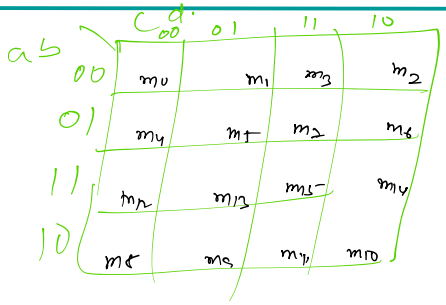
Function Minimization: K-Map

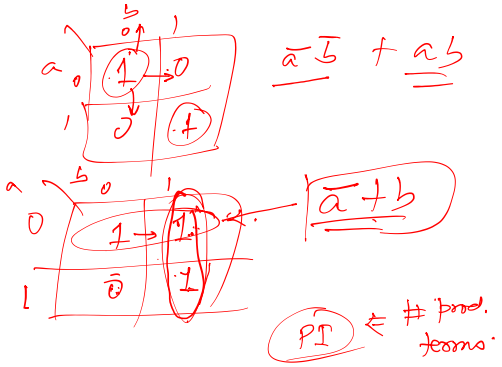


Function Minimization: K-Map

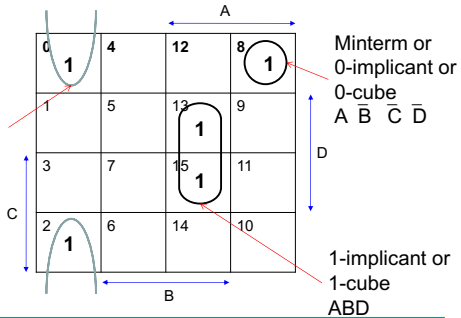


Function Minimization: K-Map





Cubes (Implicants) of 4 Variables



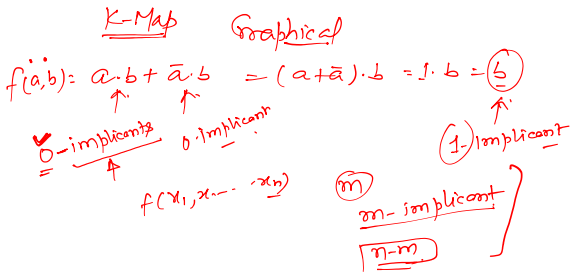
Thank You

Logic Optimization

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

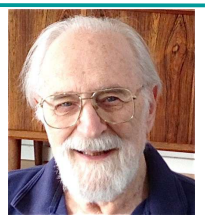
CS-230: Digital Logic Design & Computer Architecture

Lecture 9 (24 January 2022)



Graphical Method: Karnaugh Map

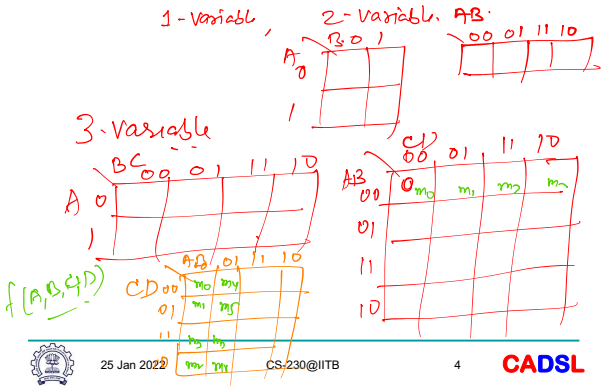
Maurice Karnaugh
- American Physicist
- Bell Lab (1952 – 66)
- Developed K-Map in 1954



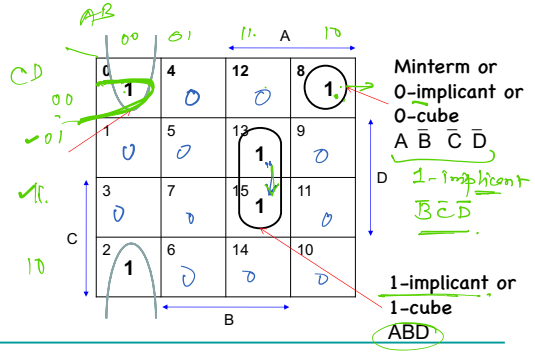
Maurice Karnaugh
Born: 4 October 1924

Karnaugh, Maurice (November 1953), "The Map Method for Synthesis of Combinational Logic Circuits" . *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*. **72**(5): 593–599

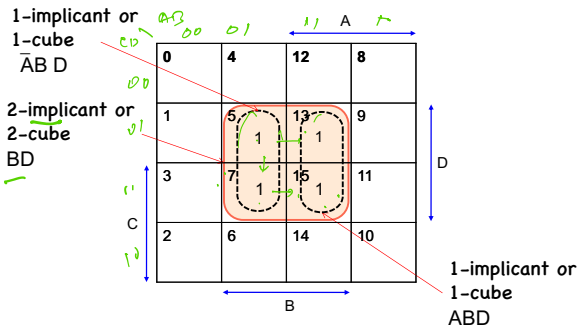
Function Minimization: K-Map



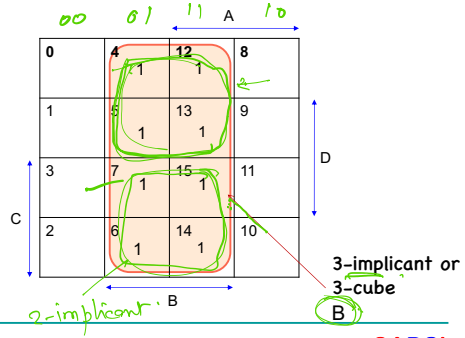
Cubes (Implicants) of 4 Variables



Growing Cubes, Reducing Products



Largest Cubes or Smallest Products

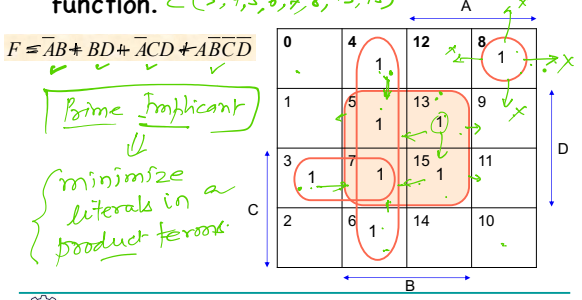


Implication and Covering

- A larger cube covers a smaller cube if all minterms of the smaller cube are included in the larger cube.
- A smaller cube implies (or subsumes) a larger cube if all minterms of the smaller cube are included in the larger cube.

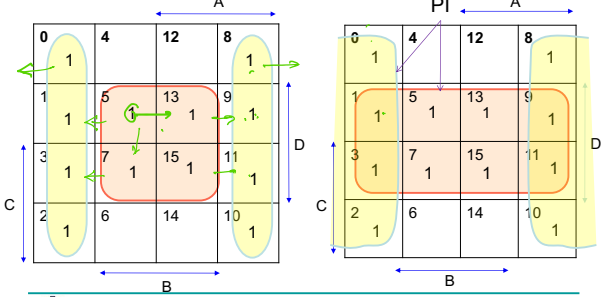
Implicants of a Function

- Minterms, products, cubes that imply the function.



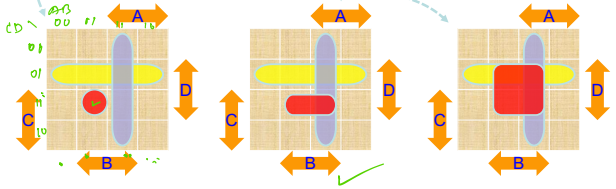
Prime Implicant (PI)

- A cube or implicant of a function that cannot grow larger by expanding into other cubes.



Growing Implicants to PI

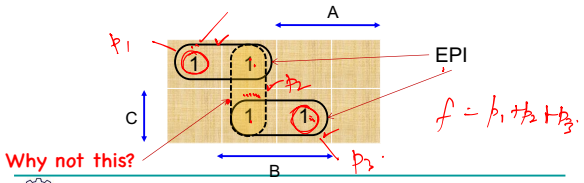
- F = $\bar{A}B + \bar{C}D + \bar{A}BCD$ = $A\bar{B} + \bar{A}BCD$ = initial implicants
  = $AB + \bar{A}BCD + BCD + \bar{C}D$ consensus th.
  = $AB + BCD + \bar{C}D$ absorption th.
  = $AB + BCD + \bar{C}D + B\bar{D}$ consensus th.
  = $AB + \bar{C}D + B\bar{D}$ absorption th.



PI ← product terms
Sum
f = (
Will it be minimum?

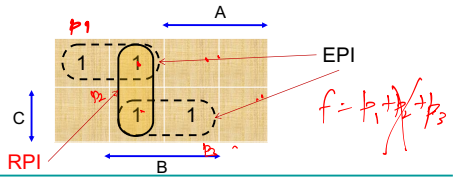
Essential Prime Implicant (EPI)

- If among the minterms subsuming a prime implicant (PI), there is at least one minterm that is covered by this and only this PI, then the PI is called an essential prime implicant (EPI).
- Also called essential prime cube (EPC).



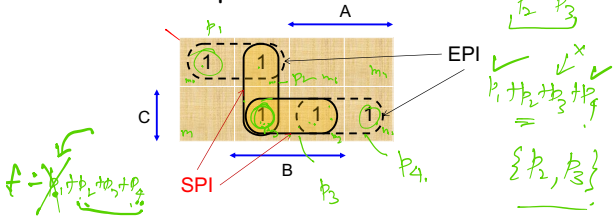
Redundant Prime Implicant (RPI)

- If each minterm subsuming a prime implicant (PI) is also covered by other essential prime implicants, then that PI is called a redundant prime implicant (RPI).
- Also called redundant prime cube (RPC).



Selective Prime Implicant (SPI)

- A prime implicant (PI) that is neither EPI nor RPI is called a selective prime implicant (SPI).
- Also called selective prime cube (SPC).
- SPIs occur in pairs.



Minimum Sum of Products (MSOP)

- Identify all prime implicants (PI) by letting minterms and implicants grow.
- Construct MSOP with PI only :
  - Cover all minterms
  - Use only essential prime implicants (EPI)
  - Use no redundant prime implicant (RPI)
  - Use cheaper selective prime implicants (SPI)

Selection of Selective PI (SPI)
↓
min number with min no of literals

Identifying EPI

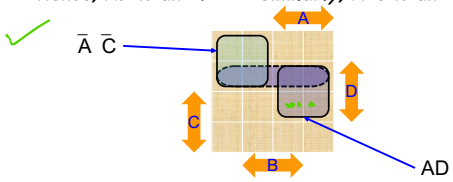
- Find all prime implicants.
- From prime implicant SOP, remove a PI.
- Apply consensus theorem to the remaining SOP.
- If the removed PI is generated, then it is either an RPI or an SPI.
- If the removed PI is not generated, then it is an EPI

Example

- PI SOP: F = $A\,D + \bar{A}\,\bar{C} + \bar{C}\bar{D}$
- Is AD an EPI?

  F – {AD} = $\bar{A}\,\bar{C} + \bar{C}\bar{D}$, no new PI can be generated

Hence, AD *is an EPI.* *Similarly,* $A\,\bar{C}$ *is an EPI.*



Example (Cont.)

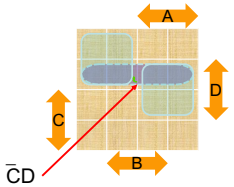
- PI SOP: F = $A\,D + \bar{A}\,\bar{C} + \bar{C}\bar{D}$
- Is $\bar{C}\bar{D}$ an EPI?

  $F - \{\bar{C}\bar{D}\} = A\,D + \bar{A}\,\bar{C}$

  $= A\,D + \bar{A}\,\bar{C} + \bar{C}\,D$ (Consensus theorem)

Hence $\bar{C}\,D$ *is not an EPI (it is an RPI)*

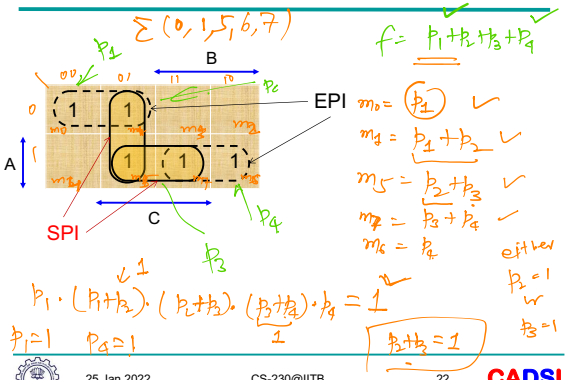
Minimum SOP:
F = $A\,D + \bar{A}\,\bar{C}$



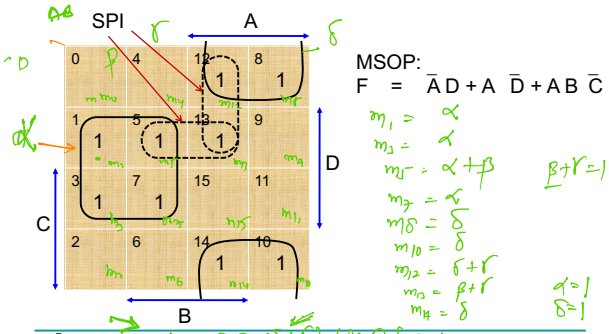
Finding MSOP

1. Start with minterm or cube SOP representation of Boolean function.
2. Find all prime implicants (PI).
3. Include all EPI's in MSOP.
4. Find the set of uncovered minterms, {UC}.
5. MSOP is minimum if {UC} is empty. *DONE.*
6. For a minterm in {UC}, include the largest PI from remaining PI's (non-EPI's) in MSOP.
7. Go to step 4.

Selection of SPI: Patrick's Method



Example: F=∑m(1,3,5,7,8,10,12,13,14)



Thank You

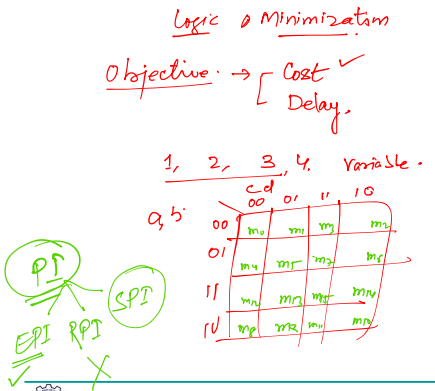
Logic Optimization

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

CS-230: Digital Logic Design & Computer Architecture

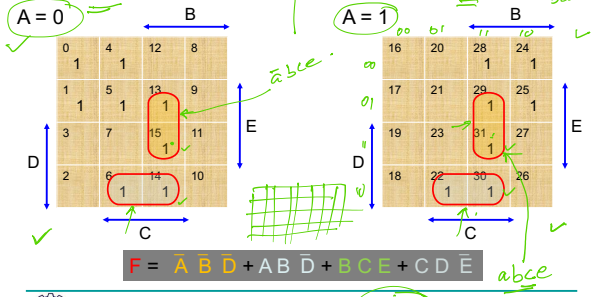
Lecture 10 (31 January 2022)

CADSL



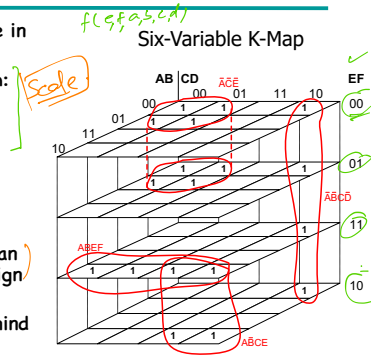
Five-Variable Function

- F(A,B,C,D,E)
  = ∑m(0,1,4,5,6,13,14,15,22,24,25,28,29,30,31)



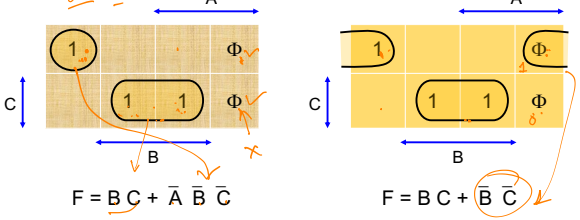
Why Not More Than Five Variables?

- Too hard to visualize in 3+ dimensions
- Systematic approach: "Tabular Methods"
  - Used in computer programs
- Why K-Maps at all?
  - Faster for quick optimizations
  - Understand Boolean logic and HW design
  - Design with simplification in mind



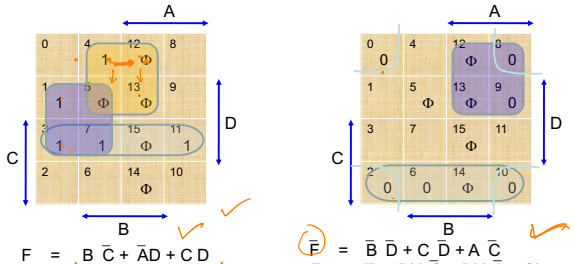
Functions with Don't Care Minterms

- F(A,B,C) = ∑m(0,3,7) + d(4,5)
- Include don't care minterms when beneficial.



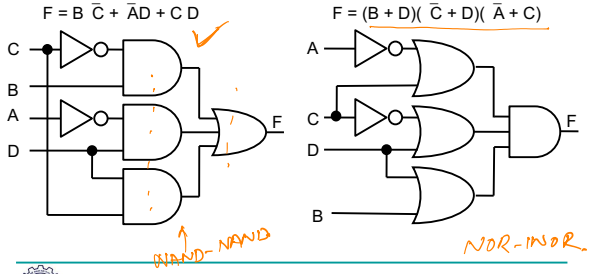
Minimized SOP and POS

- F(A,B,C,D) = ∑ m(1,3,4,7,11) + d(5,12,13,14,15)
  = ∏ M(0,2,6,8,9,10) D(5,12,13,14,15)



SOP and POS Circuits

- F(A,B,C,D) = $\sum$ m(1,3,4,7,11)+d(5,12,13,14,15)
  = $\prod$ M(0,2,6,8,9,10) D(5,12,13,14,15)



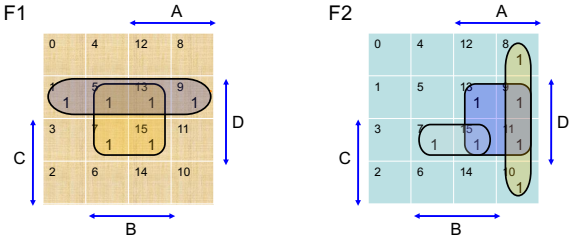
Multiple-Output Minimization

Multiple-Output Minimization

Inputs				Outputs	
A	B	C	D	F1	F2
0	0	0	0	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	1	1
1	0	0	0	0	1
1	0	0	1	1	1
1	0	1	0	0	1
1	0	1	1	0	1
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	1

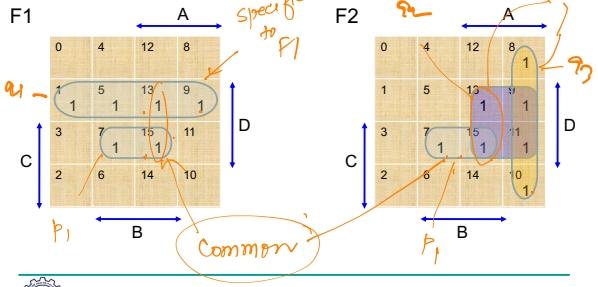
Individual Output Minimization

Need five products.

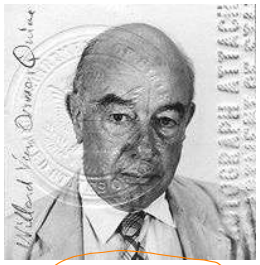


Global Minimization

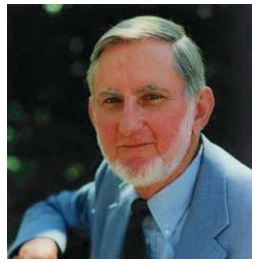
Need four products.



Quine-McCluskey



Willard V. O. Quine
1908 – 2016



Edward J. McCluskey
1929 -- 2016

Quine-McCluskey Tabular Minimization Method

- W. V. Quine, "The Problem of Simplifying Truth Functions," *American Mathematical Monthly*, vol. 59, no. 10, pp. 521-531, October 1952.
- E. J. McCluskey, "Minimization of Boolean Functions," *Bell System Technical Journal*, vol. 35, no. 11, pp. 1417-1444, November 1956.

Q-M Tabular Minimization

- Minimizes functions with many variables.
- Begin with minterms:
  - Step 1: Tabulate minterms in groups of increasing number of true variables.
  - Step 2: Conduct linear searches to identify all prime implicants (PI).
  - Step 3: Tabulate PI's vs. minterms to identify EPI's.
  - Step 4: Tabulate non-essential PI's vs. minterms not covered by EPI's. *Select* minimum number of PI's to cover all minterms.
- MSOP contains all EPI's and *selected* non-EPI's.

F(A,B,C,D) = $\sum$ m(2,4,6,8,9,10,12,13,15)

- Q-M Step 1: Group minterms with 1 true variable, 2 true variables, etc.

Minterm	ABCD	Groups
2	0010	1: single 1
4	0100	
8	1000	
6	0110	2: two 1's
9	1001	
10	1010	
12	1100	
13	1101	3: three 1's
15	1111	4: four 1's

Q-M Step 2

- Find all implicants by combining minterms, and then combining products that differ in a single variable: For example,
  - 2 and 6, or $\bar{A}\,\bar{B}\,C\,\bar{D}$ and $\bar{A}\,B\,C\,\bar{D} \rightarrow \bar{A}\,C\,\bar{D}$, written as 0 – 1 0.
- Try combining a minterm (or product) with all minterms (or products) listed below in the table.
- Include resulting products in the next list.
- If minterm (or product) does not combine with any other, mark it as PI.
- Check the minterm (or product) and repeat for all other minterms (or products).

Step 2 Executed on Example

List 1			List 2			List 3		
Minterm	ABCD	PI?	Minterms	ABCD	PI?	Minterms	ABCD	PI?
2	0010	X	2, 6	0-10	PI_2	8,9,12,13	1-0-	PI_1
4	0100	X	2,10	-010	PI_3			
8	1000	X	4,6	01-0	PI_4			
6	0110	X	4,12	-100	PI_5			
9	1001	X	8,9	100-	X			
10	1010	X	8,10	10-0	PI_6			
12	1100	X	8,12	1-00	X			
13	1101	X	9,13	1-01	X			
15	1111	X	12,13	110-	X			
			13,15	11-1	PI_7			

EPI / SPI

Step 3: Identify EPI's

Covered by EPI →				x	x		x	x	x
Minterms →	2	4	6	8	9	10	12	13	15
PI_1 is EPI				x	x		x	x	
PI_2	x		x						
PI_3	x					x			
PI_4		x	x						
PI_5		x					x		
PI_6				x		x			
PI_7 is EPI								x	x

PI_1 & PI_7 EPI

Step 4: Cover Remaining Minterms

Remaining minterms →	2	4	6	10
PI_2	x		x	
PI_3	x			x
PI_4		x	x	
PI_5		x		
PI_6				x

Integer linear program (ILP), available from MATLAB and other sources: Define integer {0,1} variables, $x_k = 1$, select PI_k; $x_k = 0$, do not select PI_k.

Minimize $\sum_k x_k$, subject to constraints:

x2 + x3 ≥ 1
x4 + x5 ≥ 1
x2 + x4 ≥ 1
x3 + x6 ≥ 1

A solution is x3 = x4 = 1, x2 = x5 = x6 = 0, or select PI_3, PI_4

Linear Programming (LP)

- A mathematical optimization method for problems where some "cost" depends on a large number of variables.
- An easy to understand introduction is:
  - S. I. Gass, *An Illustrated Guide to Linear Programming*, New York: Dover Publications, 1970.
- Very useful tool for a variety of engineering design problems.
- Available in software packages like MATLAB.

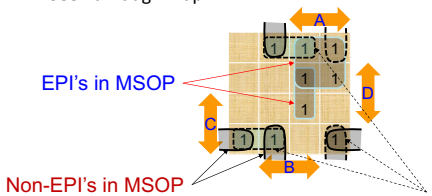
Step 4: Cover Remaining Minterms

Remaining minterms →	2	4	6	10
PI_2	x		x	
PI_3	x			x
PI_4		x	x	
PI_5		x		
PI_6				x

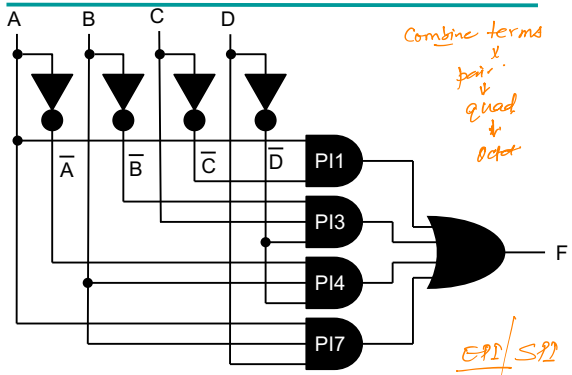
Patrick's Method

Q-M MSOP Solution and Verification

- F(A,B,C,D) = PI_1 + PI_3 + PI_4 + PI_7
  = 1-0- + -010 + 01-0 + 11-1
  = $A\,\bar{C} + \bar{B}\,C\,\bar{D} + \bar{A}\,B\,\bar{D} + A\,B\,D$
- See Karnaugh map.



Minimized Circuit



Combine terms: pair, quad, octet

EPI / SPI

Function with Don't Cares

$F(A,B,C,D) = \sum m(4,6,8,9,10,12,13) + \sum d(2, 15)$

- Q-M Step 1: Group "all" minterms with 1 true variable, 2 true variables, etc.

Minterm	ABCD	Groups
2	0010	1: single 1
4	0100	
8	1000	
6	0110	2: two 1's
9	1001	
10	1010	
12	1100	
13	1101	3: three 1's
15	1111	4: four 1's

Step 2: Same As Before on "All" Minterms

List 1			List 2			List 3		
Minterm	ABCD	PI?	Minterms	ABCD	PI?	Minterms	ABCD	PI?
2	0010	X	2, 6	0-10	PI2	8,9,12,13	1-0-	PI1
4	0100	X	2,10	-010	PI3			
8	1000	X	4,6	01-0	PI4			
6	0110	X	4,12	-100	PI5			
9	1001	X	8,9	100-	X			
10	1010	X	8,10	10-0	PI6			
12	1100	X	8,12	1-00	X			
13	1101	X	9,13	1-01	X			
15	1111	X	12,13	110-	X			
			13,15	11-1	PI7			

Step 3: Identify EPI's Ignoring Don't Cares

Covered by EPI →			x	x		x	x
Minterms →	4	6	8	9	10	12	13
PI1 is EPI			x	x		x	x
PI2		x					
PI3					x		
PI4	x	x					
PI5	x					x	
PI6			x		x		
PI7							x

Step 4: Cover Remaining Minterms

Remaining minterms →	4	6	10
PI_2		x	
PI_3			x
PI_4	x	x	
PI_5	x		
PI_6			x

Integer linear program (ILP), available from Matlab and other sources: Define integer {0,1} variables, xk = 1, select PI_k; xk = 0, do not select PI_k.

Minimize $\sum_k$ xk, subject to constraints: x4 + x5 ≥ 1

x2 + x4 ≥ 1

x3 + x6 ≥ 1

A solution is x3 = x4 = 1, x2 = x5 = x6 = 0, or select PI_3, PI_4

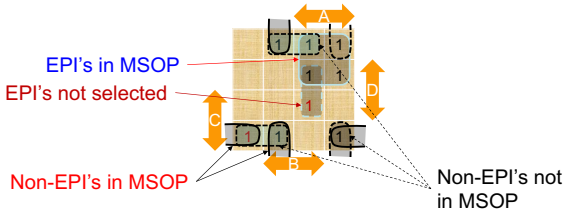
Step 4: Cover Remaining Minterms

Remaining minterms →	4	6	10
PI_2		x	
PI_3			x
PI_4	x	x	
PI_5	x		
PI_6			x

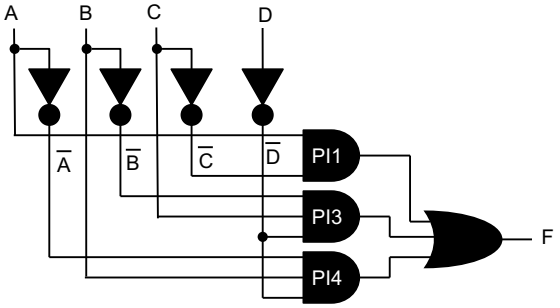
Patrick's Method

Q-M MSOP Solution and Verification

- F(A,B,C,D) = PI_1 + PI_3 + PI_4
  = 1-0- + -010 + 01-0
  = $A\bar{C} + \bar{B}C\bar{D} + \bar{A}B\bar{D}$
- See Karnaugh map.



Minimized Circuit



QM Minimizer on the Web

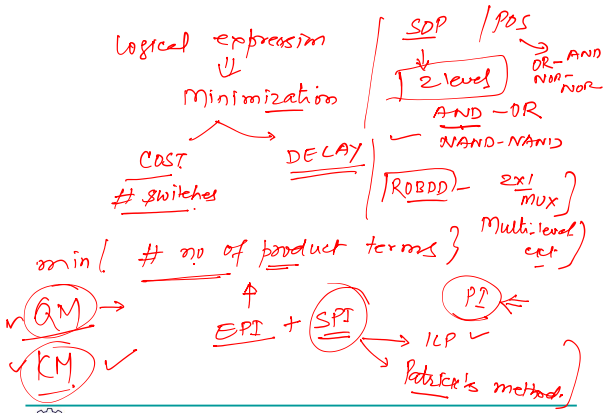
- http://quinemccluskey.com/

Thank You

Arithmetic Circuits

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

CS-230: Digital Logic Design & Computer Architecture



Multiple-Level Optimization

- Multiple-level circuits are circuits that have more than two level (plus input and/or output inverters)
- For a given function, multiple-level circuits can have reduced gate input cost compared to two-level (SOP and POS) circuits
- Multiple-level optimization is performed by applying transformations to circuits represented by equations while evaluating cost

Transformations

- Factoring - finding a factored form from SOP or POS expression
  - Algebraic - No use of axioms specific to Boolean algebra such as complements or idempotence
  - Boolean - Uses axioms unique to Boolean algebra
- Decomposition - expression of a function as a set of new functions

Transformations (continued)

- Substitution of G into F - expression function F as a function of G and some or all of its original variables
- Extraction - decomposition applied to multiple functions simultaneously

Transformation Examples

- **Algebraic Factoring**

$F = \bar{A}\bar{C}\bar{D} + \bar{A}B\bar{C} + ABC + AC\bar{D}$ $G = 16$

– **Factoring:**

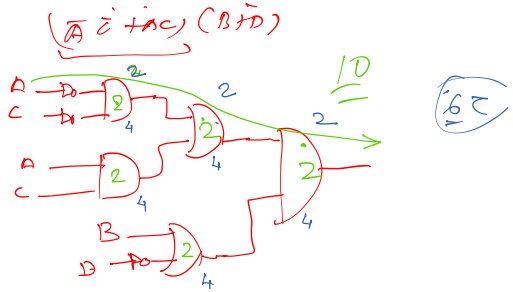
$F = \bar{A}(\bar{C}\bar{D} + B\bar{C}) + A(BC + C\bar{D})$ $G = 16$

– **Factoring again:**

$F = \bar{A}\bar{C}(B + \bar{D}) + AC(B + \bar{D})$ $G = 12$

– **Factoring again:**

$F = (\bar{A}\bar{C} + AC)(B + \bar{D})$ $G = 10$



Transformation Examples

- Decomposition
  - $F = A'C'D' + A'BC' + ABC + ACD'$ $G = 16$
  - The terms $A'C' + AC$ and $B + D'$ can be defined as new functions H and E respectively, decomposing $F = (A'C' + AC)(B + D')$:

$F = H E$, $H = A'C' + AC$, $E = B + D'$ $G = 10$

- This series of transformations has reduced G from 16 to 10, a substantial savings.
- The resulting circuit has three levels plus input inverters.

Transformation Examples

- **Substitution of E into F**
  - **Returning to F just before the final factoring step:**

$F = \bar{A}\bar{C}(B + \bar{D}) + AC(B + \bar{D})$ $G = 12$

  - **Defining $E = B + \bar{D}$, and substituting in F:**

$F = \bar{A}\bar{C}E + AC(E)$ $G = 10$

  - **This substitution has resulted in the same cost as the decomposition**

Transformation Examples

Summary

- Multi-level Optimization
- Transformations
  - Factoring - find a factored form from SOP or POS expression
  - Decomposition - express a function as a set of new functions
  - Substitution - express function F as a function of G and some or all of its original variables

Arithmetic Circuits:

Adders

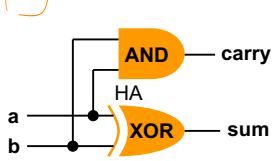
Half-Adder Adds two Bits

"half" because it has no carry input

- Adding two bits:

a	b	a + b
0	0	00
0	1	01
1	0	01
1	1	10

carry sum

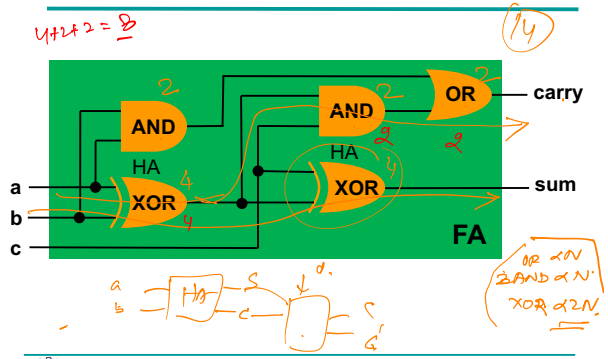


Full Adder: Include Carry Input

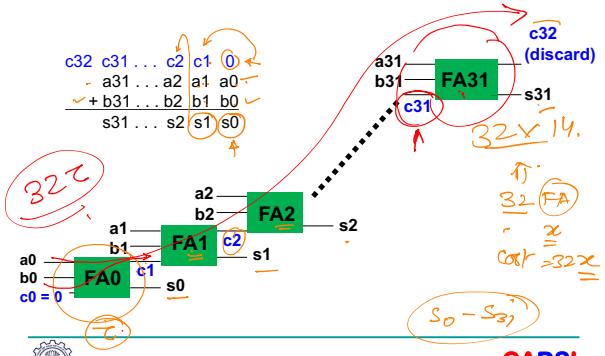
a	b	c	s = a + b + c	
			Decimal value	Binary value
0	0	0	0	00
0	0	1	1	01
0	1	0	1	01
0	1	1	2	10
1	0	0	1	01
1	0	1	2	10
1	1	0	2	10
1	1	1	3	11

CARRY SUM

Full-Adder Adds Three Bits

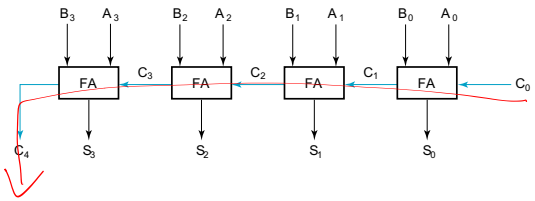


32-bit Ripple-Carry Adder



4-bit Ripple-Carry Binary Adder

- A four-bit Ripple Carry Adder made from four 1-bit Full Adders:

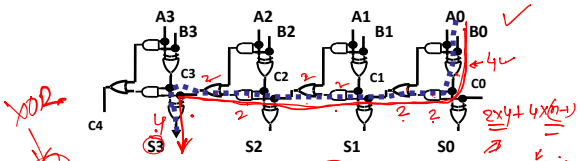


How Fast is Ripple-Carry Adder?

- Longest delay path (critical path) runs from (a0, b0) to sum31/C32
- Suppose delay of full-adder is 100ps
- Critical path delay = 3,200ps
- Must use more efficient ways to handle carry

Carry Propagation & Delay

- Propagation delay:
  - Carry must ripple from LSB to MSB.
- The gate-level propagation path for a 4-bit ripple carry adder of the last example:



- Too slow for many bits (e.g. 32 or 64)

Carry Propagation & Delay

Carry Propagation & Delay

Ripple Carry Adder

- Easy to create
  - Good hierarchy
  - Tileable structures
- Small hardware
- Slow!
  - Design is limited by the delays in propagating carry through all of the bitwise additions
  - The output bit at position m is not valid until after the carry out of the m-1 position is ready

Thank You

Arithmetic Circuits

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

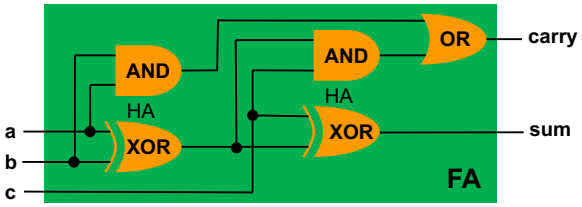
CS-230: Digital Logic Design & Computer Architecture

Lecture 13 (03 February 2022)

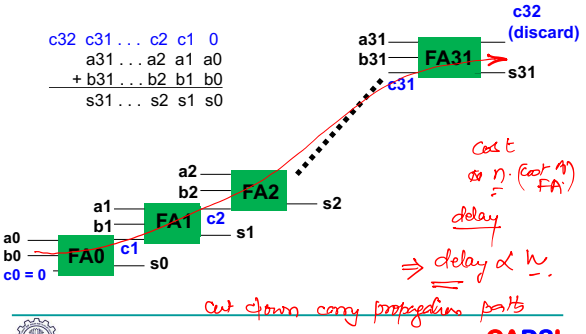
CADSL

Arithmetic Circuits: (Adders)

Full-Adder Adds Three Bits



32-bit Ripple-Carry Adder



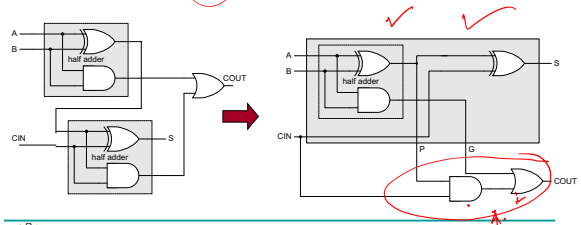
Ripple Carry Adder

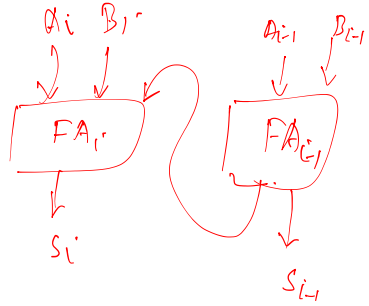
- Easy to create
  - Good hierarchy
  - Tileable structures
- Small hardware
- Slow!
  - Design is limited by the delays in propagating carry through all of the bitwise additions
  - The output bit at position m is not valid until after the carry out of the m-1 position is ready

FAST ADDERS

Faster Addition

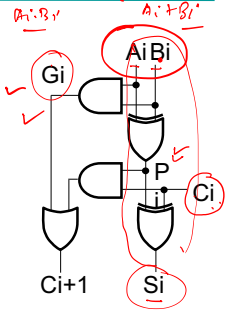
- For big adders, the carry-chain is very long
- Separate the carry chain and sum logic
- Partial Full Adders
  - Contain only the sum part of a FA





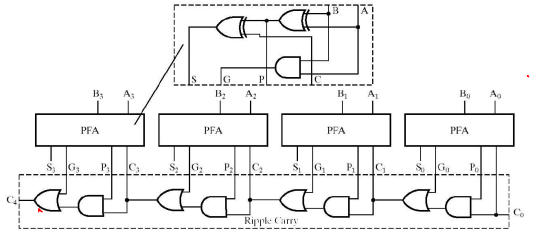
Carry Lookahead

- Given Stage i from a Full Adder, we know that there will be a carry generated when $A_i = B_i =$ "1", whether or not there is a carry-in.
- Alternately, there will be a carry propagated if the "half-sum" is "1" and a carry-in, C_i occurs.
- These two signal conditions are called *generate*, denoted as G_i, and *propagate*, denoted as P_i respectively and are identified in the circuit:



Reassembling RCA Using PFAs

- Can create ripple-carry adder with PFAs
  - G: **Generates** a carry at this position
  - P: **Propagates** a carry through this position



Carry Lookahead (continued)

- In the ripple carry adder:
  - G_i, P_i, and S_i are local to each cell of the adder
  - C_i is also local each cell
- In the carry lookahead adder, in order to reduce the length of the carry chain, C_i is changed to a more global function spanning multiple cells
- Defining the equations for the Full Adder in term of the P_i and G_i:

$$P_i = A_i \oplus B_i \qquad G_i = A_i B_i$$
$$S_i = P_i \oplus C_i \qquad C_{i+1} = G_i + P_i C_i$$

Carry Lookahead Development

- Flatten equations for carry using G_i and P_i terms for less significant bits
- Beginning at the cell 0 with carry in C_0:

$$C1 = G0 + P0\ C0$$

$$C2 = G1 + P1\ C1 = G1 + P1(G0 + P0\ C0)$$
$$= G1 + P1G0 + P1P0\ C0$$

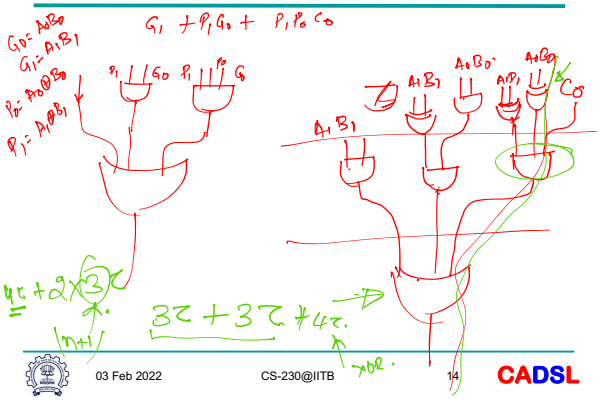
Carry Lookahead Development

$$C3 = G2 + P2\ C2 = G2 + P2(G1 + P1G0 + P1P0\ C0)$$

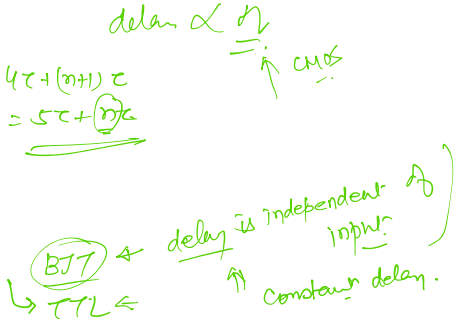
$$= G2 + P2G1 + P2P1G0 + P2P1P0\ C0$$

$$C4 = G3 + P3\ C3 = G3 + P3G2 + P3P2G1 + P3P2P1G0 + P3P2P1P0\ C0$$

Carry Lookahead Adder



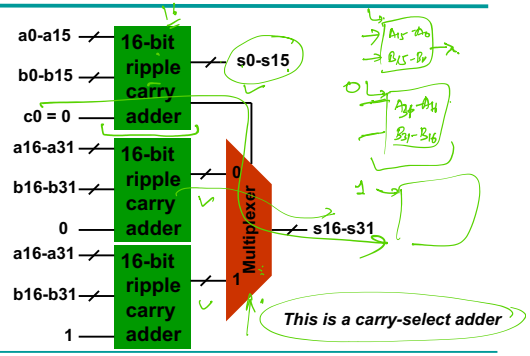
Carry Lookahead Adder



Carry Look-Ahead Adder

- As usual, can trade area/power for speed
  - Multi-level logic (ripple carry) reduces area
  - Flattening carry logic increases speed
- Sometimes called CLA

Speeding Up the Adder



Fast Adders

- In general, any output of a 32-bit adder can be evaluated as a logic expression in terms of all 65 inputs.
- Number of levels of logic can be reduced to $\log_2 N$ for N-bit adder. Ripple-carry has N levels.
- More gates are needed, about $\log_2 N$ times that of ripple-carry design.

Prefix Computation

Key Architectures for Carry Calculation

- 1960: J. Sklansky adder
- 1973: Kogge-Stone adder
- 1980: Ladner-Fisher adder
- 1982: Brent-Kung adder
- 1987: Han Carlson adder
- 1999: S. Knowles adder

Other parallel adder architectures:

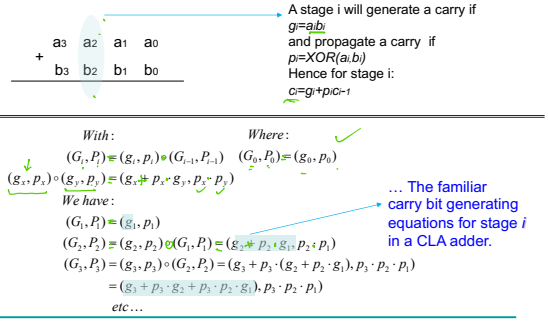
- 1981: H. Ling adder
- 2001: Beaumont-Smith

Binary Addition

- **Input:** two n-bit binary numbers $a_{n-1}\ldots a_1 a_0$ and $b_{n-1}\ldots b_1 b_0$, one bit carry-in c_0
- **Output:** n-bit sum $s_{n-1}\ldots s_1 s_0$ and one bit carry out c_n
- Prefix Addition: Carry generation & propagation

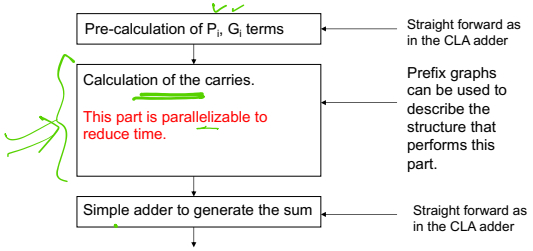
$$\text{Generate}: \quad g_i = a_i b_i$$
$$\text{Propagate}: \quad p_i = a_i \oplus b_i$$
$$c_{i+1} = g_i + p_i \cdot c_i$$
$$s_i = c_i \oplus (a_i \oplus b_i)$$

Binary Addition as a prefix sum problem.



Parallel Prefix Adders

- The parallel prefix adder employs the 3-stage structure of the CLA adder. The improvement is in the carry generation stage which is the most intensive one:



Prefix Addition – Formulation

Pre-processing: $g_i = a_i b_i \qquad p_i = a_i \oplus b_i$

Prefix Computation:

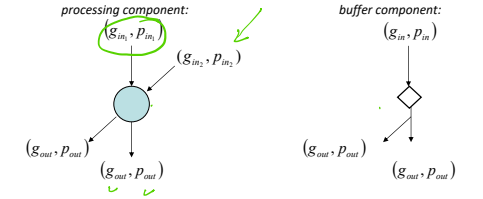
$$G_{[i:k]} = G_{[i:j]} + P_{[i:j]} G_{[j-1:k]}$$
$$P_{[i:k]} = P_{[i:j]} P_{[j-1:k]}$$

Post-processing:

$$c_{i+1} = G_{[i:0]} + P_{[i:0]} \cdot c_0$$
$$s_i = p_i \oplus c_i$$

Computation of Carries – Prefix Graphs

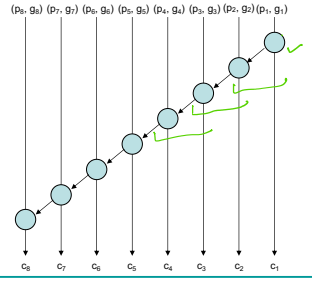
The components usually seen in a prefix graph are the following:



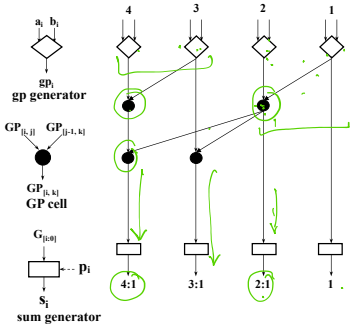
$$(g_{out}, p_{out}) = (g_{in_1} + p_{in_1} \cdot g_{in_2}, p_{in_1} \cdot p_{in_2}) \qquad (g_{out}, p_{out}) = (g_{in}, p_{in})$$

Prefix graphs for representation of Prefix addition

- Serial adder carry generation represented by prefix graphs



Prefix Adder – Prefix Structure Graph

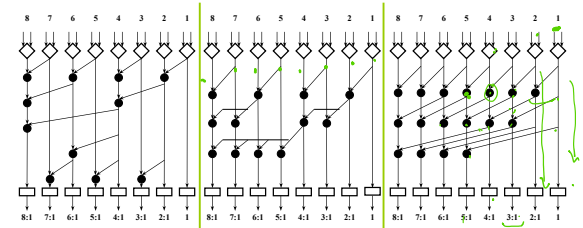


Pre-processing

Prefix Computation

Post-processing

Classical Prefix Adders

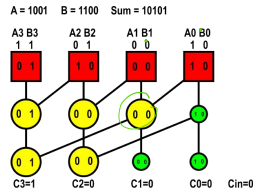


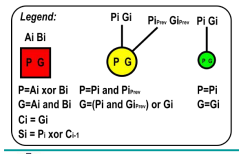
Brent-Kung:
Logical levels: $2\log_2 n - 1$
Max fanouts: 2

Sklansky:
Logical levels: $\log_2 n$
Max fanouts: n/2

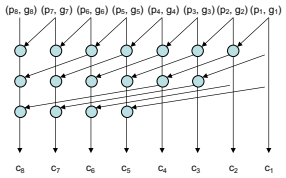
Kogge-Stone:
Logical levels: $\log_2 n$
Max fanouts: 2

Kogge Stone Adder



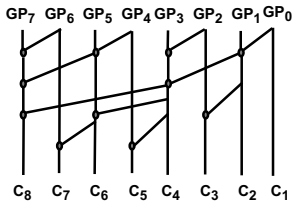


1973: Kogge-Stone adder



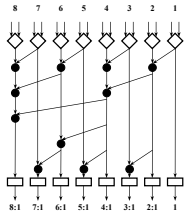
- The Kogge-Stone adder has:
  - Low depth
  - High node count (implies more area).
  - Minimal fan-out of 1 at each node (implies faster performance).

1982: Brent-Kung (BK) Adder



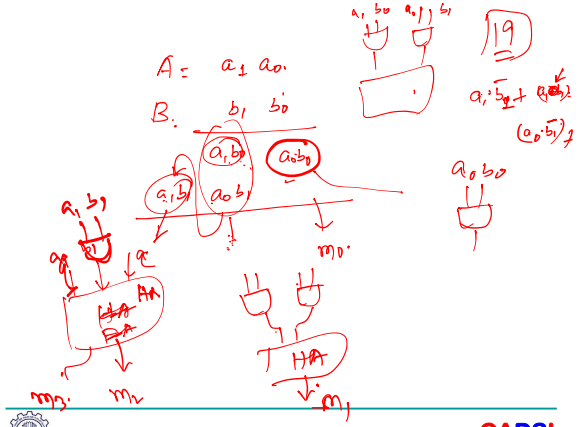
Brent and Kung, "A regular layout for parallel adders", In IEEE transaction for Computers, 1982

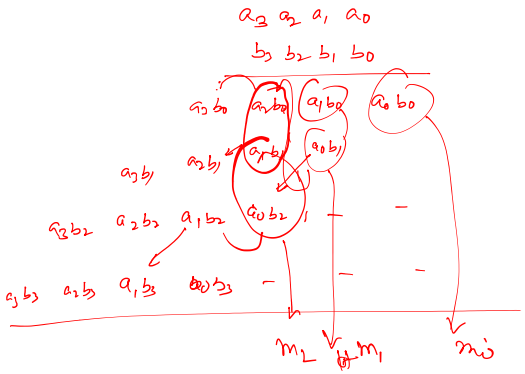
Brent-Kung (BK) Adder

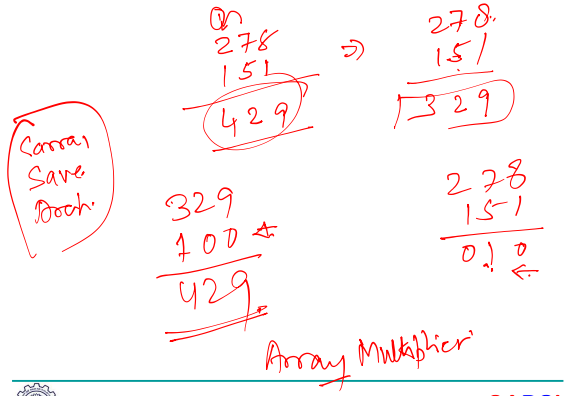


Multiplier

A_i







Thank You

Sequential Circuits

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

CS-230: Digital Logic Design & Computer Architecture

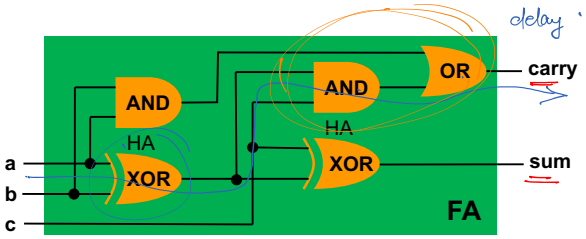
Lecture 14 (07 February 2022)

CADSL

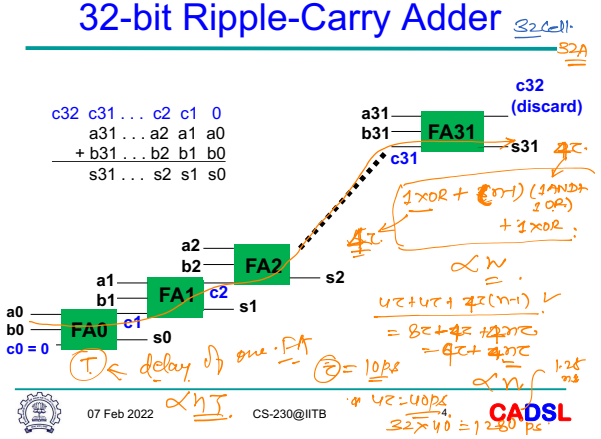
Arithmetic Circuits: Adders

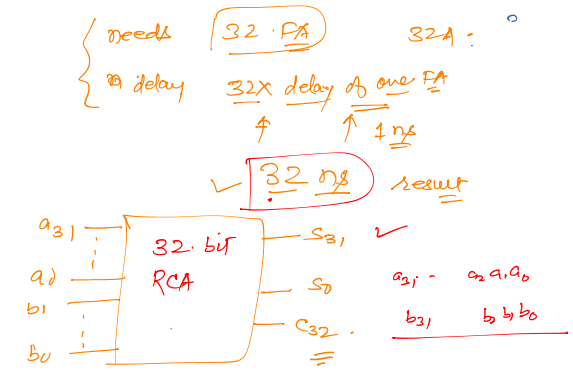
Adders
Area / Power
RCA
performance
CLA → αn
CSA – carry select
$\alpha k \log n$ ← Prefix – Adder
Multipliers

Full-Adder Adds Three Bits

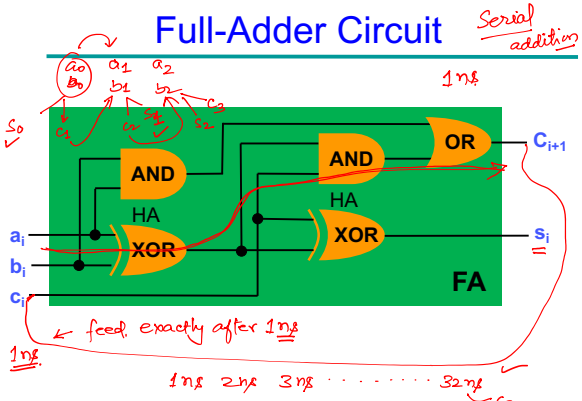


32-bit Ripple-Carry Adder

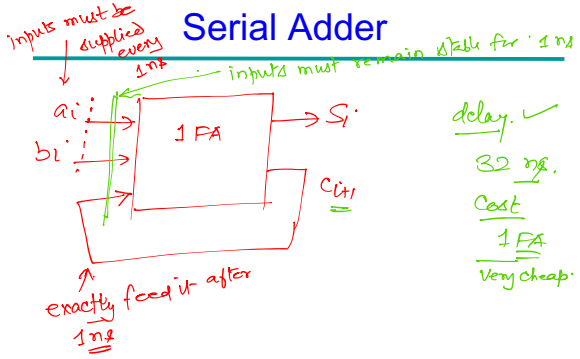




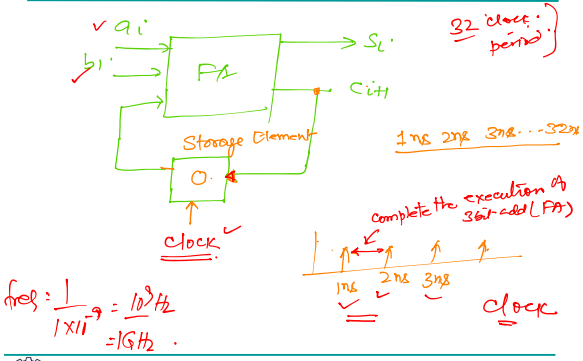
Full-Adder Circuit



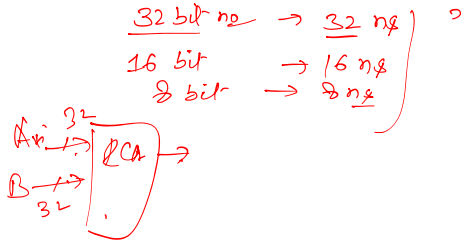
Serial Adder



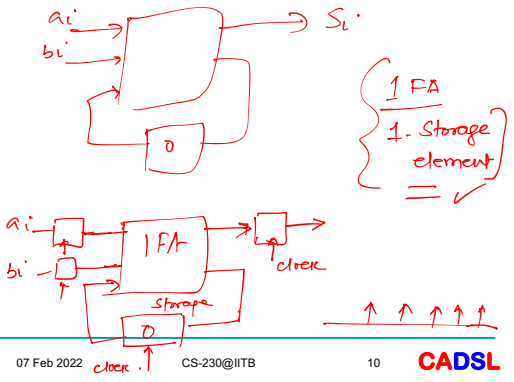
Serial Adder



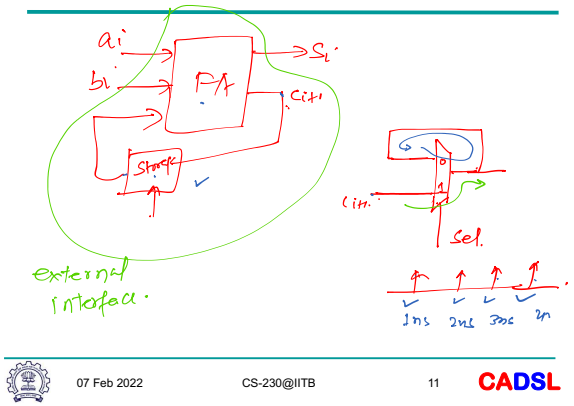
Serial Adder



Serial Adder



Serial Adder



Serial Adder

Input

a_i	b_i	c_i	S_i - output
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Output is not only dependent on the current inputs but also dependent on previous inputs

Sequential behaviour

Synchronous Sequential Circuits

Serial Adder

Temporal Behaviour → Sequential logic

(MEMORY)

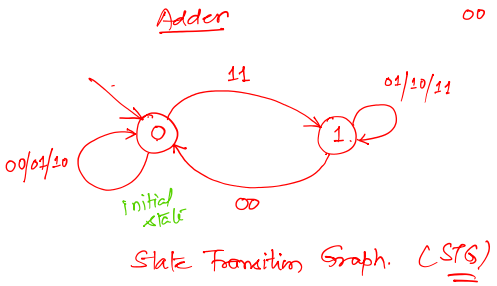
STATE OF THE SYSTEM

c_{31} of a_i - $a_1 a_0$ b_{31} b_0

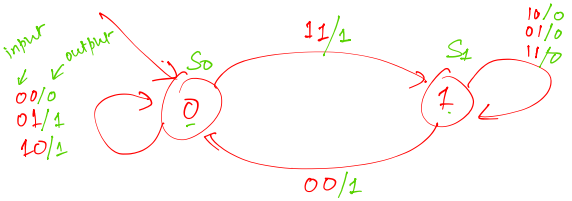
Memorize the carry.

Carry = 0

Carry = 1



Serial Adder



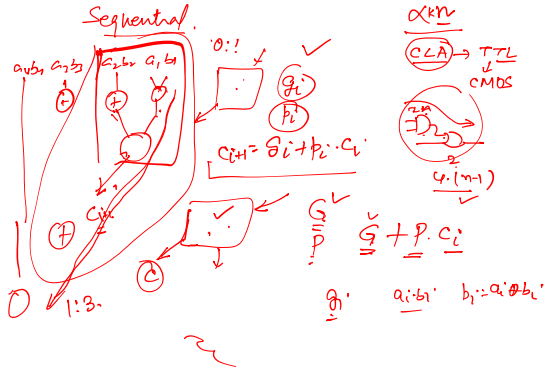
Thank You

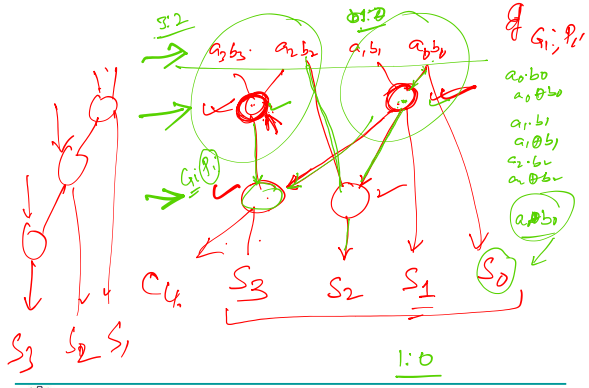
Sequential Circuits

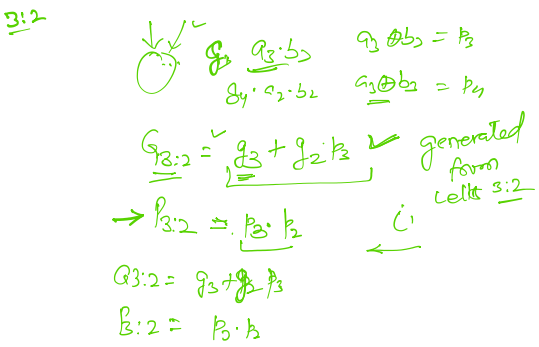
Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

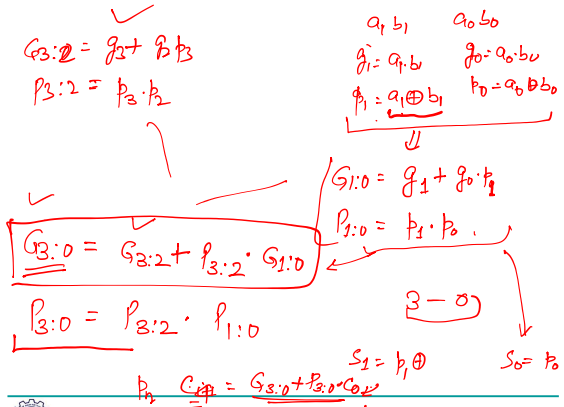
CS-230: Digital Logic Design & Computer Architecture

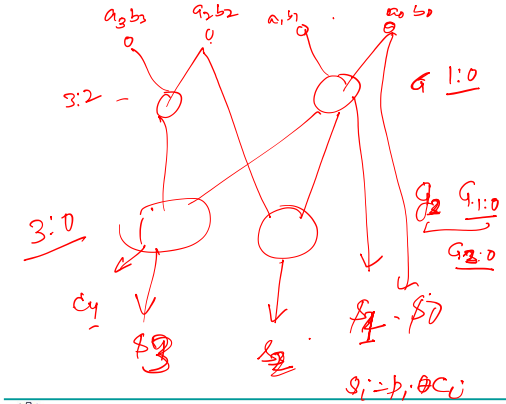
Lecture 15 (08 February 2022)

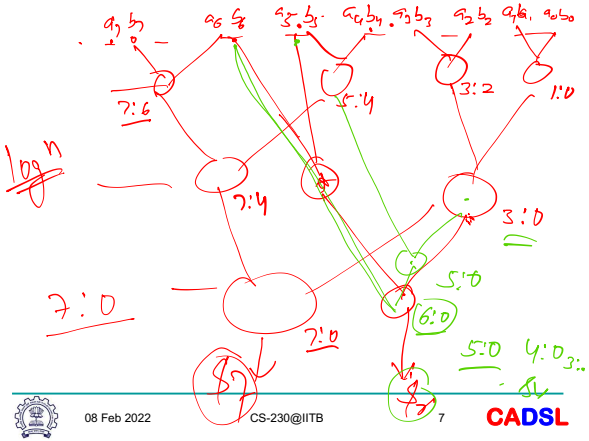


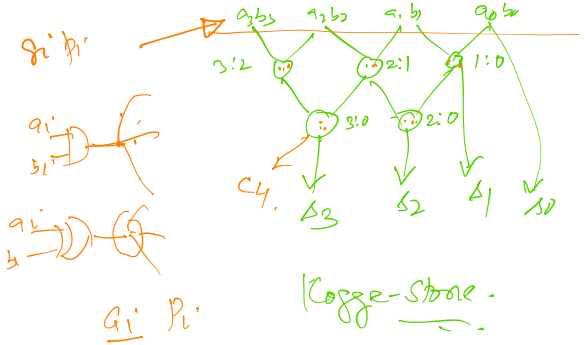




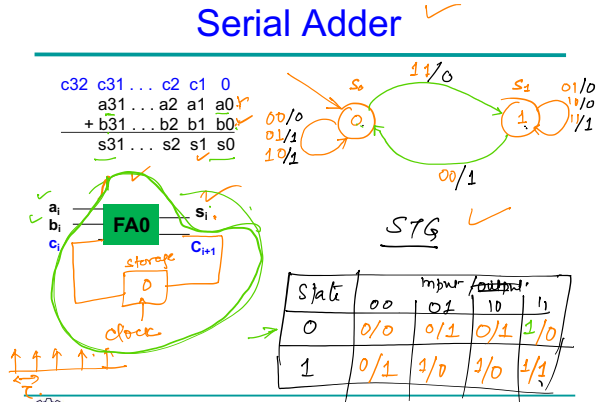




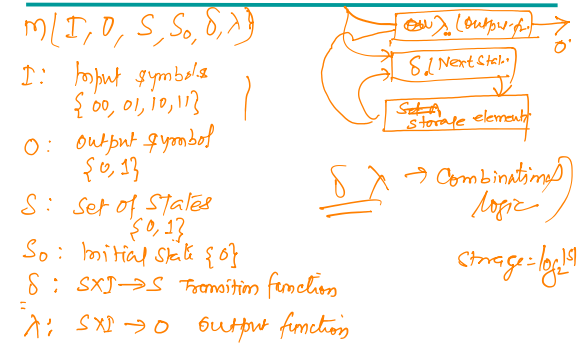




Serial Adder



State Machine



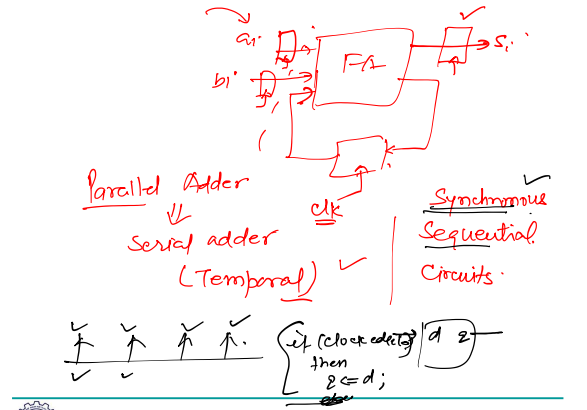
Thank You

Sequential Circuits

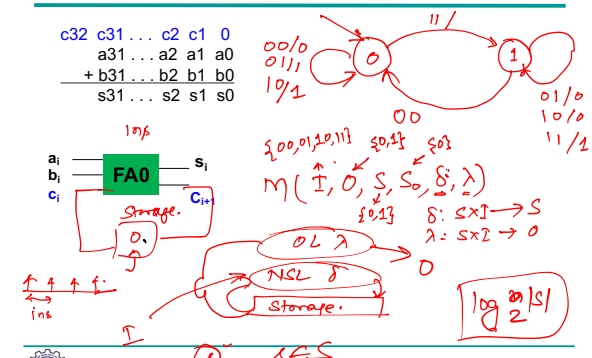
Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

CS-230: Digital Logic Design & Computer Architecture

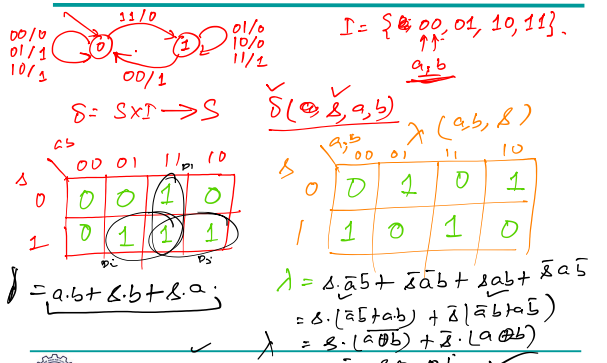
Lecture 16 (10 February 2022) CADSL



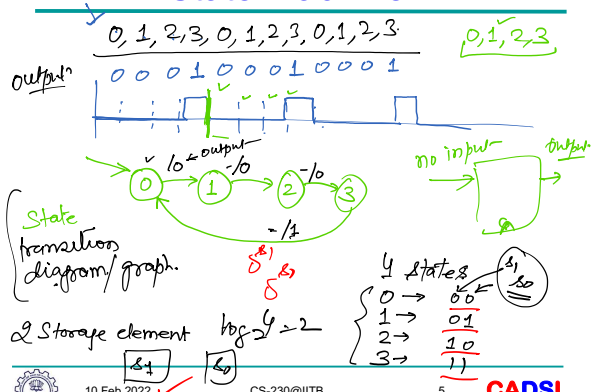
Serial Adder



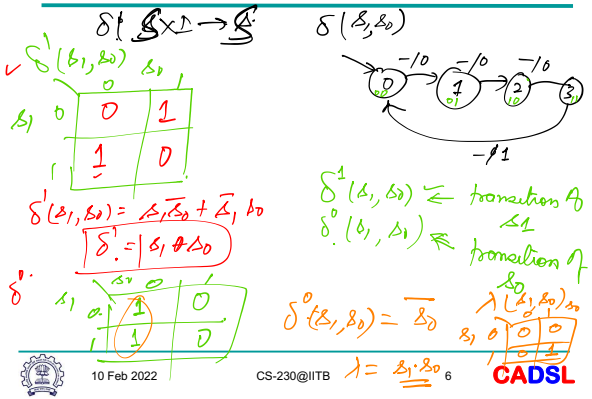
State Machine



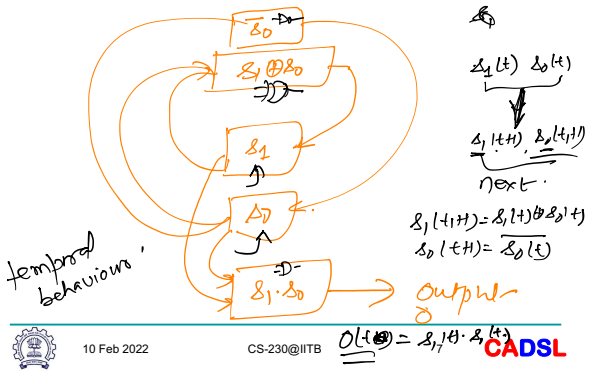
State Machine



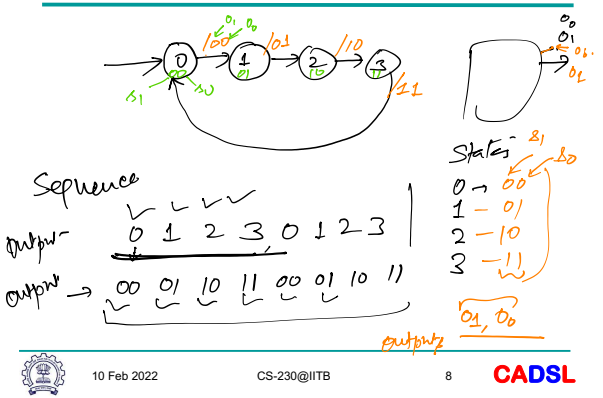
Finite State Machine



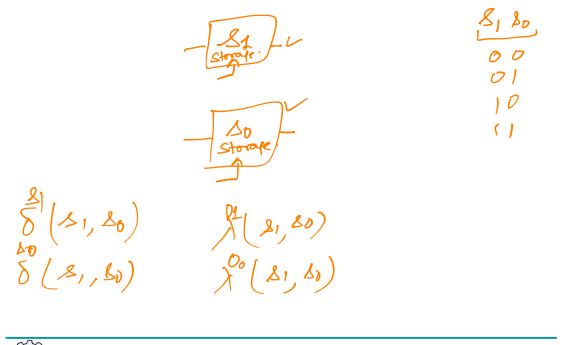
Finite State Machine



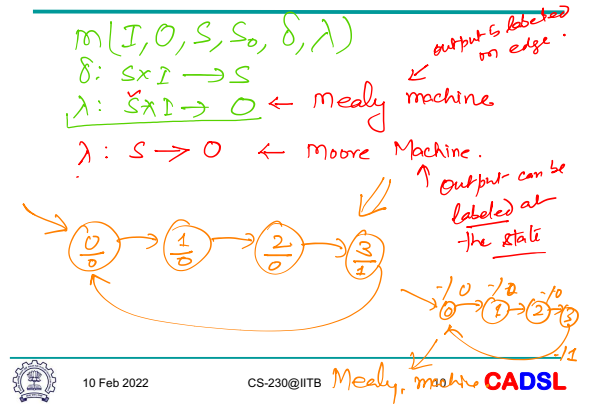
Finite State Machine



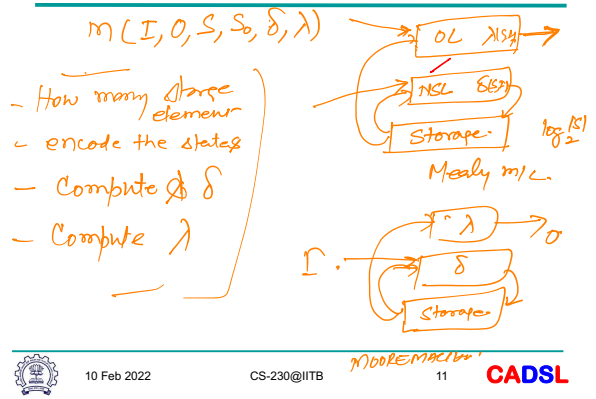
Finite State Machine



Finite State machine



Finite State machine



Thank You

Sequential Circuits

Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

CS-230: Digital Logic Design & Computer Architecture

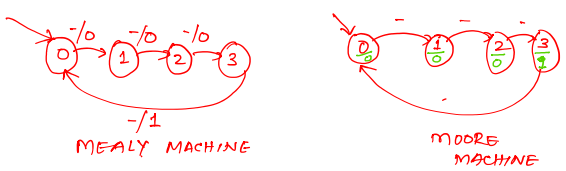
Sequential Circuits
- temporal behaviour
Clock
Synchronous Sequential Circuit
Asynchronous Sequential Circuits
(no clock)
order the event (handshaking)

State Machine

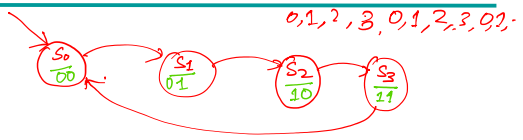
$M(I, O, S, S_0, \delta, \lambda)$

$\lambda : S \to O$ MOORE
$\lambda : S \times I \to O$ MEALY

0,0,0,1,0,0,0,1



State Machine



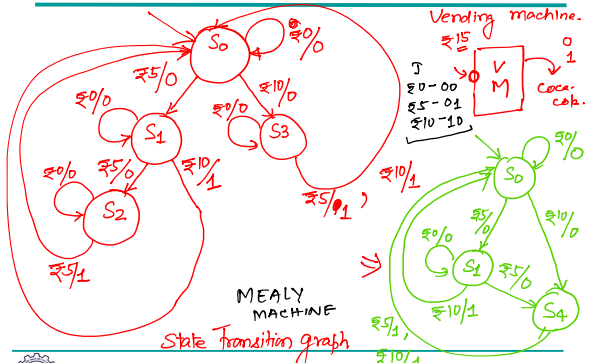
0,1,2,3,0,1,2,3,0,1,...

Free running counter.

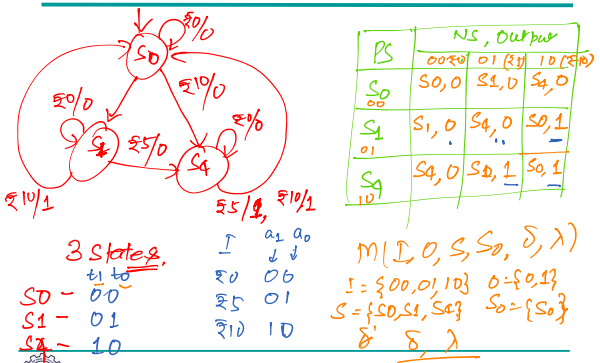
0,1,2,3,0,1,2,3, - - -

O: {0, 1, 2, 3}
00 01 10 11

Finite State Machine



Finite State Machine



PS	NS, Output		
	00 (₹0)	01 (₹5)	10 (₹10)
S_0 00	S_0, 0	S_1, 0	S_4, 0
S_1 01	S_1, 0	S_4, 0	S_0, 1
S_4 10	S_4, 0	S_0, 1	S_0, 1

3 States.
$t_1\ t_0$
S0 - 00
S1 - 01
S4 - 10

I	a_1	a_0
₹0	0	0
₹5	0	1
₹10	1	0

$M(I, O, S, S_0, \delta, \lambda)$
$I = \{00, 01, 10\}$ $O = \{0, 1\}$
$S = \{S_0, S_1, S_4\}$ $S_0 = \{S_0\}$
δ' δ_0 λ

Finite State Machine

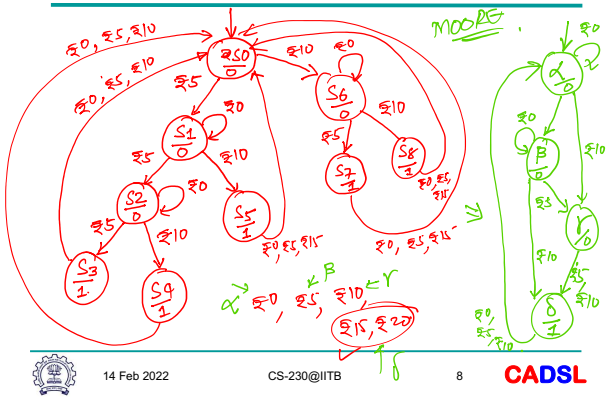
$\delta^{t_1}(t_1, t_0, a_1, a_0)$
$\delta^{t_0}(t_1, t_0, a_1, a_0)$
$\lambda(t_1, t_0, a_1, a_0)$

$t_1 t_0$ \ $a_1 a_0$	00	01	11	10
00	0	0	X	0
01	0	0	X	1
11	X	X	X	X
10	0	1	X	1

$\lambda = t_0 a_1 + t_1 a_1 + t_1 a_0$

$\delta^{t_1} =$
$\delta^{t_0} =$

Finite State Machine



Finite State machine

MOORE MACHINE

$\alpha, \beta, \Gamma, \delta$

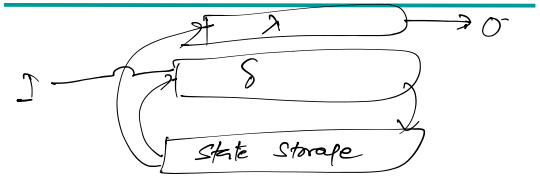
00 01 10 11

t_1, t_0

$I = \{$₹0 00, ₹5 01, ₹10 10$\}$

$\delta^{t_1}(t_1, t_0, a_1, a_0)$
$\delta^{t_0}(t_1, t_0, a_1, a_0)$
$\lambda(t_1, t_0)$

Finite State machine



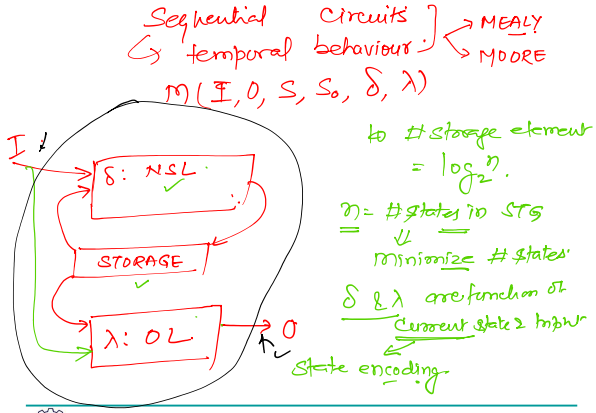
Thank You

Sequential Circuits

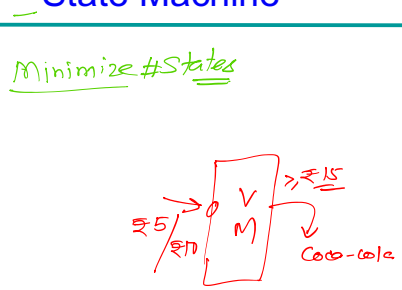
Virendra Singh
Professor
Computer Architecture and Dependable Systems Lab
Department of Computer Science & Engineering, and
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.cse.iitb.ac.in/~viren/
E-mail: viren@{cse, ee}.iitb.ac.in

CS-230: Digital Logic Design & Computer Architecture

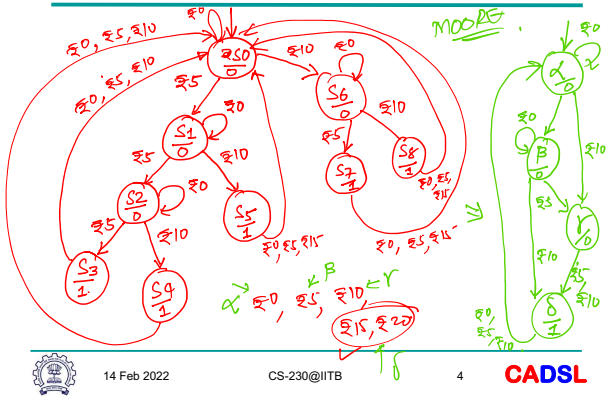
Lecture 18 (15 February 2022)



State Machine



Finite State Machine



State Minimization

X-Successor – If an input sequence X takes a machine from state S_i to state S_j, then S_j is said to be the X-successor of S_j

Strongly connected:- If for every pair of states (S_i, S_j) of a machine M there exists an input sequence which takes M from state S_i to S_j, then M is said to be strongly connected

State Equivalence

- Two states S_i and S_j of machine M are distinguishable if and only if there exists at least one finite input sequence which, when applied to M, causes different output sequences, depending on whether S_i or S_j is the initial state

State Equivalence

- Two states S_i and S_j of machine M are distinguishable if and only if there exists at least one finite input sequence which, when applied to M, causes different output sequences, depending on whether S_i or S_j is the initial state
- The sequence which distinguishes these states is called a distinguishing sequence of the pair (S_i, S_j)

State Equivalence

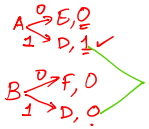
- Two states S_i and S_j of machine M are distinguishable if and only if there exists at least one finite input sequence which, when applied to M, causes different output sequences, depending on whether S_i or S_j is the initial state
- The sequence which distinguishes these states is called a distinguishing sequence of the pair (S_i, S_j)
- If there exists for pair (S_i, S_j) a distinguishing sequence of length k, the states in (S_i, S_j) are said to be k-distinguishable

State Equivalence

Machine M1

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

(A, B) – 1 Distinguishable



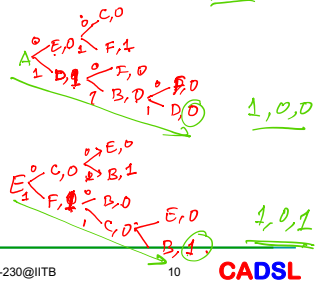
State Equivalence

Machine M1

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

(A, E) – 3 Distinguishable

Seq - 111



State Equivalence

Machine M1

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

(A, B) – 1 Distinguishable

(A, E) – 3 Distinguishable

Seq - 111

k-equivalent – The states that are not *k*-distinguishable are said to be k-equivalent

Also *r-equivalent* r<k

Distiguishable States

K- equivalent

K- indistinguishable.

State Equivalence

- States S_i and S_j of machine **M** are said to be equivalent if and only if, for every possible input sequence, the same output sequence will be produced regardless of whether S_i or S_j is the initial state
- States that are k-equivalent for all k < n-1, are equivalent
- $S_i \equiv S_j$, and $S_j \equiv S_k$, then $S_i \equiv S_k$

State Equivalence

- The set of states of a machine M can be partitioned into disjoint subsets, known as equivalence classes
- Two states are in the same equivalence class if and only if they are equivalent, and are in different classes if and only if they are distinguishable

Property: If S_i and S_j are equivalent states, their corresponding X-successors, for all X, are also equivalent

State Minimization Procedure

1. Partition the states of M into subsets s.t. all states in same subset are *1-equivalent*
2. Two states are 2-equivalent iff they are 1-equivalent and their I_i successors, for all possible I_i, are also 1-equivalent

MOORE MINIMIZATION

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

P0 = (ABCDEF)

State Minimization Procedure

1. Partition the states of M into subsets s.t. all states in same subset are *1-equivalent*
2. Two states are 2-equivalent iff they are 1-equivalent and their I_i successors, for all possible I_i, are also 1-equivalent

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

P0 = (ABCDEF)

P1 = (ACE), (BDF)

State Minimization Procedure

1. Partition the states of M into subsets s.t. all states in same subset are *1-equivalent*
2. Two states are 2-equivalent iff they are 1-equivalent and their I_i successors, for all possible I_i, are also 1-equivalent

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

P0 = (ABCDEF)

P1 = (ACE), (BDF)

P2 = (ACE), (BD), (F)

State Minimization Procedure

1. Partition the states of M into subsets s.t. all states in same subset are *1-equivalent*
2. Two states are 2-equivalent iff they are 1-equivalent and their I_i successors, for all possible I_i, are also 1-equivalent

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

P0 = (ABCDEF)

P1 = (ACE), (BDF)

P2 = (ACE), (BD), (F)

P3 = (AC), (E), (BD), (F)

State Minimization Procedure

1. Partition the states of M into subsets s.t. all states in same subset are *1-equivalent*
2. Two states are 2-equivalent iff they are 1-equivalent and their I_i successors, for all possible I_i, are also 1-equivalent

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

P0 = (ABCDEF)

P1 = (ACE), (BDF)

P2 = (ACE), (BD), (F)

P3 = (AC), (E), (BD), (F)

P4 = (AC), (E), (BD), (F)

State Minimization Procedure

1. Partition the states of M into subsets s.t. all states in same subset are *1-equivalent*
2. Two states are 2-equivalent iff they are 1-equivalent and their I_i successors, for all possible I_i, are also 1-equivalent

PS	NS, z	
	X = 0	X = 1
A	E, 0	D, 1
B	F, 0	D, 0
C	E, 0	B, 1
D	F, 0	B, 0
E	C, 0	F, 1
F	B, 0	C, 0

P0 = (ABCDEF)

P1 = (ACE), (BDF)

P2 = (ACE), (BD), (F)

P3 = (AC), (E), (BD), (F)

P4 = (AC), (E), (BD), (F)

Machine Equivalence

- Two machines M1, M2 are said to be equivalent if and only if, for every state in M1, there is corresponding equivalent state in M2
- If one machine can be obtained from the other by relabeling its states they are said to be isomorphic to each other

PS	NS, z	
	X = 0	X = 1
AC - α	β, 0	γ, 1
E - β	α, 0	δ, 1
BD - γ	δ, 0	γ, 0
F - δ	γ, 0	α, 0

State Equivalence - Example

Machine M2

PS	NS, z	
	X = 0	X = 1
A	E, 0	C, 0
B	C, 0	A, 0
C	B, 0	G, 0
D	G, 0	A, 0
E	F, 1	B, 0
F	E, 0	D, 0
G	D, 0	G, 0

P0 = (ABCDEFG)

P1 = (ABCDFG) (E)

P2 = (AF) (BCDG) (E)

P3 = (AF) (BD) (CG) (E)

P4 = (A) (F) (BD) (CG) (E)

P5 = (A) (F) (BD) (CG) (E)

Thank You