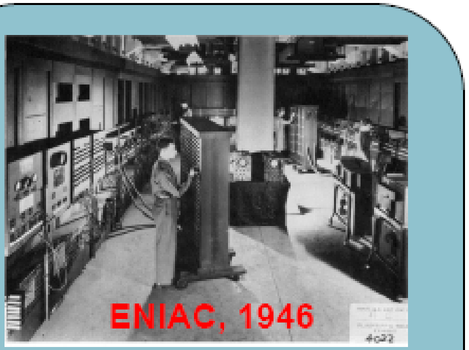
Ch 4. Combinational Logic Technologies

History
- From Switches to Integrated Circuits

- The underlying implementation technologies for digital systems

Vacuum Tubes

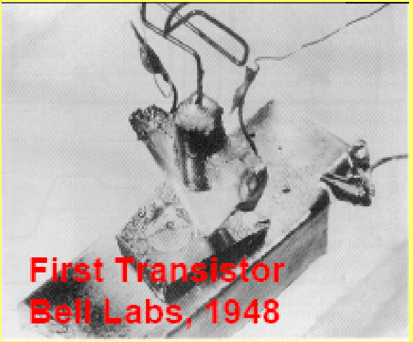


ENIAC, 1946

UNIVAC, 1951

1900 adds/sec

Transistors

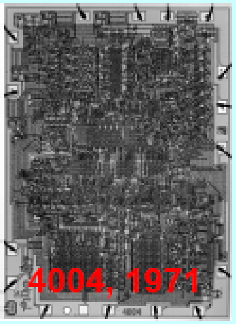


First Transistor
Bell Labs, 1948

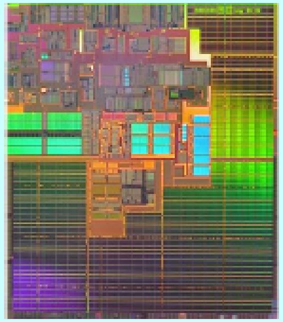
IBM System/360, 1964

500,000 adds/sec

VLSI Circuits



4004, 1971



Intel Itanium, 2003
2,000,000,000
adds/sec

Packaged Logic, Configurability, and Programmable Logic

- Standard parts
  - Integrated circuits that contain small number of simple logic gates widely used
- ROM and PLA/PAL
  - ROM
    - A fixed array of ones and zeros
  - Programmable logic array (PLA), Programmable array logic (PAL)
    - An array of fixed logic gates, arranged in a standard two-level form such as AND/OR
- Application-Specific Integrated Circuit (ASIC)
  - Gate array / Standard cells
- Field-Programmable Gate Array (FPGA)

Packaged Logic, Configurability, and Programmable Logic (cont'd)

- Historical development of components

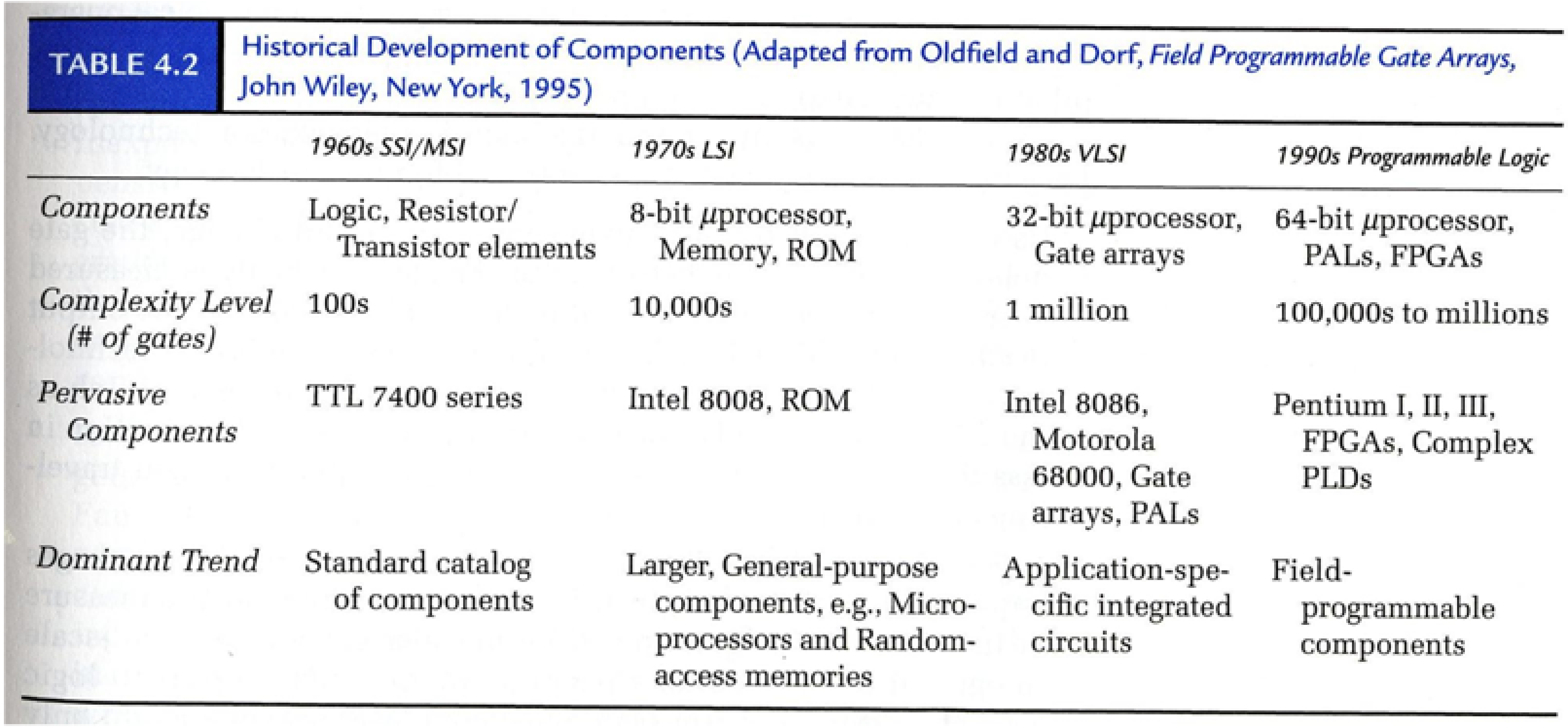


TABLE 4.2 Historical Development of Components (Adapted from Oldfield and Dorf, *Field Programmable Gate Arrays,* John Wiley, New York, 1995)

	1960s SSI/MSI	*1970s LSI*	*1980s VLSI*	*1990s Programmable Logic*
Components	Logic, Resistor/ Transistor elements	8-bit μprocessor, Memory, ROM	32-bit μprocessor, Gate arrays	64-bit μprocessor, PALs, FPGAs
Complexity Level (# of gates)	100s	10,000s	1 million	100,000s to millions
Pervasive Components	TTL 7400 series	Intel 8008, ROM	Intel 8086, Motorola 68000, Gate arrays, PALs	Pentium I, II, III, FPGAs, Complex PLDs
Dominant Trend	Standard catalog of components	Larger, General-purpose components, e.g., Microprocessors and Random-access memories	Application-specific integrated circuits	Field-programmable components

Technology Metrics

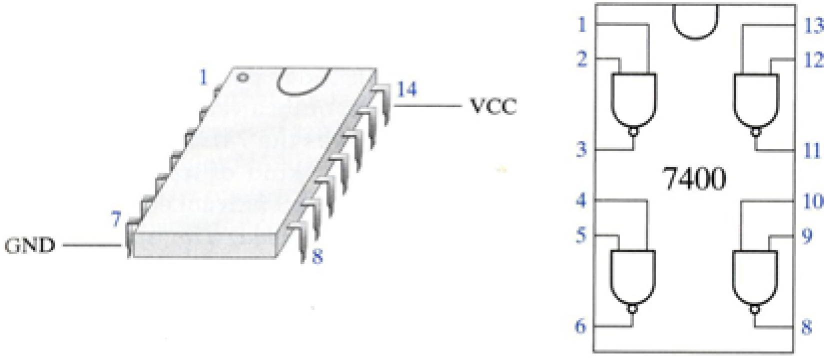
- Gate delay
  - The time delay from an input change to an output change
  - t(50% output final value) - t(50% of input final value)
- Degree of integration
  - The chip area and number of chip packages required to implement a given function in a technology
- Power dissipation
  - Power consumed as gates perform their logic functions
- Noise margin
  - The maximum voltage that can be added to or subtracted from the logic voltages carried on wires and still have the circuit interpret the voltage as the correct logic value
- Component cost

Technology Metrics (cont'd)

- Fan-out
  - The number of gate inputs to which a given gate output can be connected without exceeding electrical limitations
- Driving capability
  - The speed of communications between packaged components
- Configurability

Basic Logic Components
- Fixed Logic

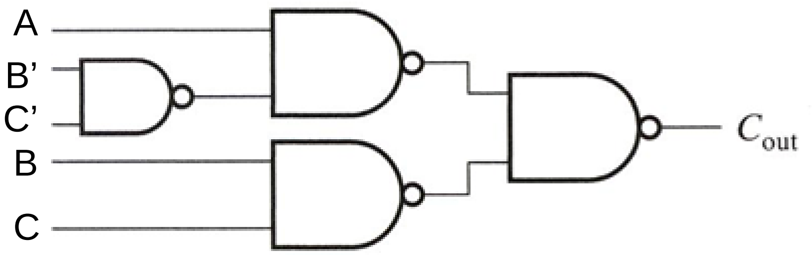
- Cell-based design
  - Designers pick and choose gates in a given standard cell library for logic functions they want to implement.
    - NAND, NOR, XOR, ADDER, MUX, …
  - SSI and MSI (Small and Medium Scale Integrated Circuit)
    - TTL: a family of packaged logic components
    - Example: TTL 7400

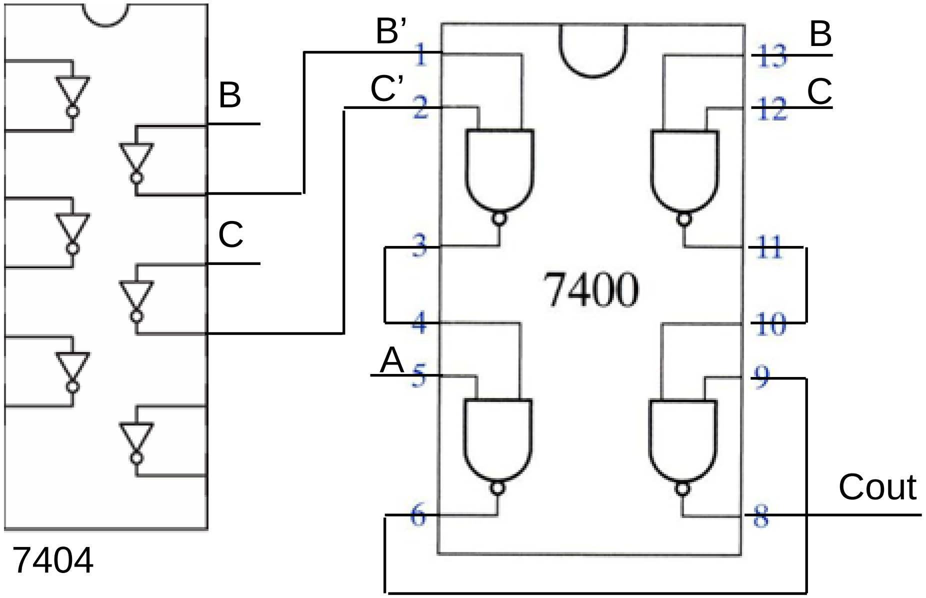


  - Standard cell library for modern ASIC
    - provides the same kinds of logic functions as those TTL standard catalog contains.

Fixed Logic (cont'd)

- Combinational logic implementation
  - Pick logic packages and wire them
  - Make Package/Cell Counts minimal





Basic Logic Components (cont'd) - Look-Up Tables

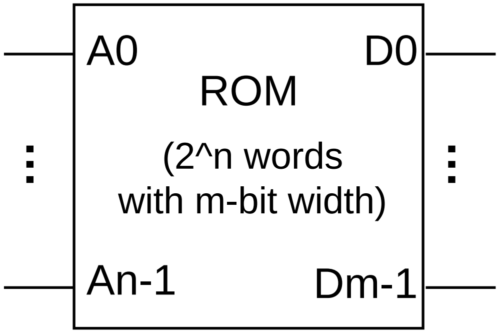
- Look-up table approach
  - Storing the output value of a function for each input combination in a table
  - Using the current input value as an index to look-up what the output should be
  - To change the function, simply change the values stored in the table, not change any wiring
  - Less sensitive to the complexity of the function
- Example
  - If the input A=0, B=1, and C=0,
    the corresponding index (address) of the table = 010
    and the value of the entry = 0100.
  - Thus, the output F0=0, F1=1, F2=0, F3=0

Index			Output			
A	B	C	F0	F1	F2	F3
0	0	0	0	0	1	0
0	0	1	1	1	1	0
0	1	0	0	1	0	0
0	1	1	0	0	0	1
1	0	0	1	0	1	1
1	0	1	1	0	0	0
1	1	0	0	0	0	1
1	1	1	0	1	0	0

Storing the outputs at each entry in a table device

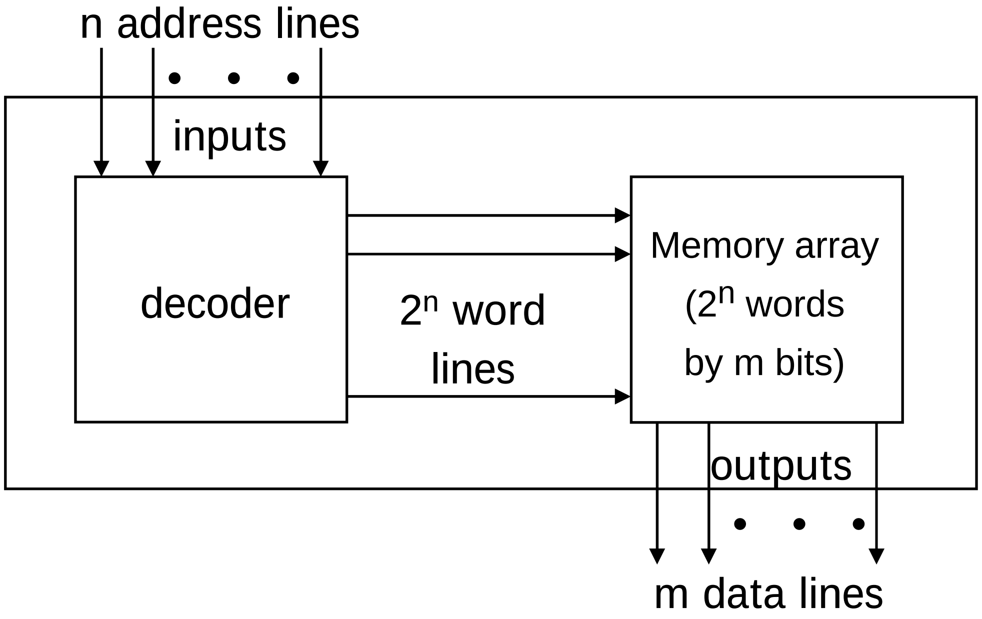
Look-Up Tables (cont'd)

- ROM (Read-Only Memory)
  - an array of values intended to be read many times but only written once
  - We can program the output of a truth table directly into a ROM
  - Address: index of the array
    - Inputs are used to index a row of the array
  - The value of the corresponding row are read out as the values of the outputs
  - Each additional input bit doubles the size of the memory array
    - Good to implement a complicated function that is difficult to simplify using the standard methods or change often the function.
  - Variants
    - Programmable ROM (PROM)
    - Erasable PROM (EPROM)
    - Electrically erasable PROM (EEPROM)



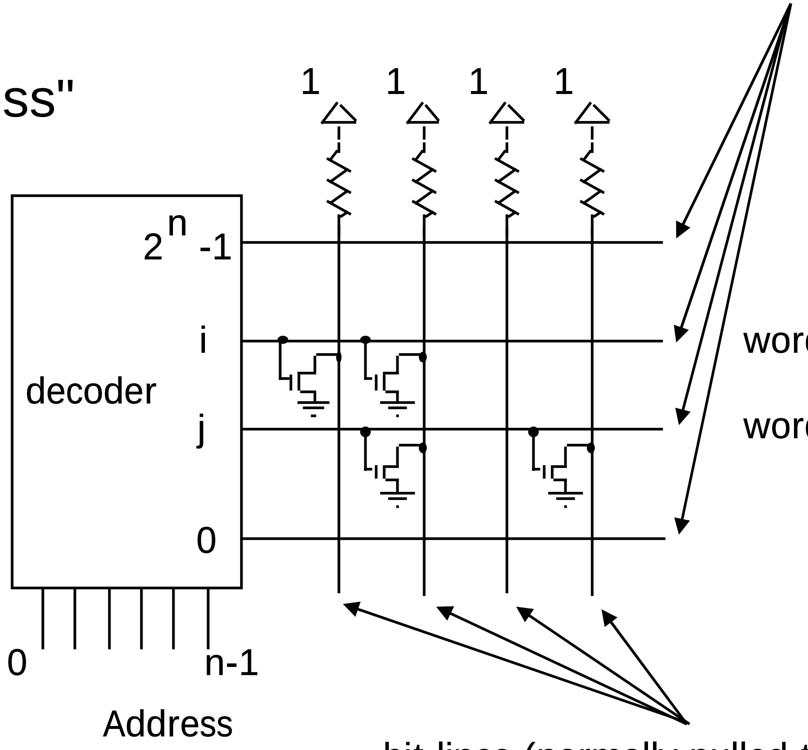
Look-Up Tables (cont'd)

- Internal organization of a ROM
  - Each row of the array is called a word and is selected by the control inputs, which are called the address.
  - The number of columns in the array is called the bit-width or word size.
  - Similar to a PLA structure but with a fully decoded AND array
    - completely flexible OR array (unlike PAL)



Look-Up Tables (cont'd)

- Two dimensional array of 1s and 0s
  - entry (row) is called a "word"
  - width of row = word-size
  - index is called an "address"
  - address is input
  - selected word is output



word lines (only one is active – decoder is just right for this)

word[i] = 0011

word[j] = 1010

internal organization

bit lines (normally pulled to 1 through resistor – selectively connected to 0 by word line controlled switches)

Look-Up Tables (cont'd)

- Combinational logic implementation (two-level canonical form) using a ROM

$$F0 = A'\,B'\,C \;+\; A\,B'\,C' \;+\; A\,B'\,C$$

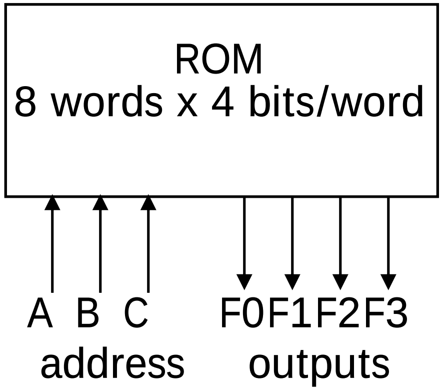
$$F1 = A'\,B'\,C \;+\; A'\,B\,C' \;+\; A\,B\,C$$

$$F2 = A'\,B'\,C' \;+\; A'\,B'\,C \;+\; A\,B'\,C'$$

$$F3 = A'\,B\,C \;+\; A\,B'\,C' \;+\; A\,B\,C'$$

A	B	C	F0	F1	F2	F3
0	0	0	0	0	1	0
0	0	1	1	1	1	0
0	1	0	0	1	0	0
0	1	1	0	0	0	1
1	0	0	1	0	1	1
1	0	1	1	0	0	0
1	1	0	0	0	0	1
1	1	1	0	1	0	0

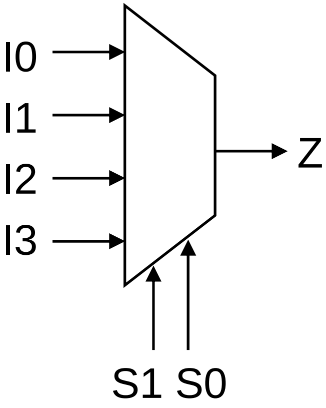
truth table



block diagram

Basic Logic Components (cont'd) - Multiplexer/Selector

- Multiplexer/Selector (MUX)
  - Sets its single output to the same value as one of its inputs under the direction of its control inputs.
  - 2^n data inputs, n control inputs (called "selects"), 1 output
  - used to connect 2^n input points to a single output point
  - control signal pattern forms binary index of input connected to output
- Functional description



$$Input: \quad I_j = \{0,1\} \quad where \ j = 0, 1, ..., 2^n - 1$$

$$S_i = \{0,1\} \quad where \ i = 0, 1, ..., n - 1$$

$$Output: \quad Z = \{0,1\}$$

$$Function: \ Z = I_j \quad where \ j = S = \sum_{i=0}^{n-1} S_i 2^i$$

Multiplexer/Selector (cont'd)

- Truth table of MUX
  - 2:1 MUX

$Z = A' I_0 + A I_1$

A	Z
0	I_0
1	I_1

I_1	I_0	A	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

functional form

logical form

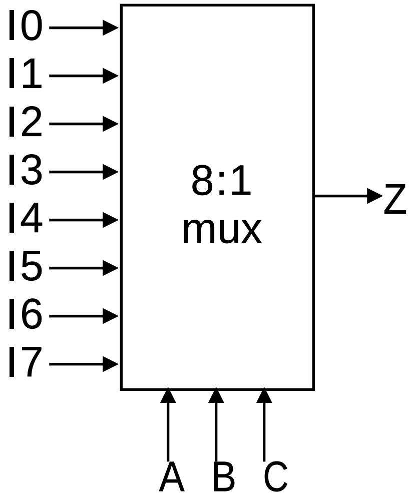
two alternative forms
for a 2:1 Mux truth table

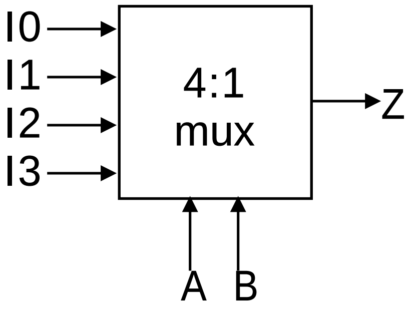
Multiplexer/Selector (cont'd)

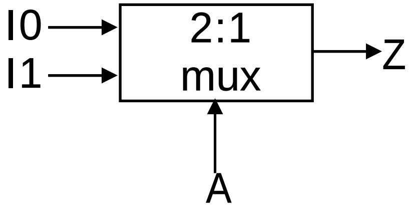
- Boolean equation of MUX
  - 2:1 mux: Z = A'I0 + AI1
  - 4:1 mux: Z = A'B'I0 + A'BI1 + AB'I2 + ABI3
  - 8:1 mux: Z = A'B'C'I0 + A'B'CI1 + A'BC'I2 + A'BCI3 + AB'C'I4 + AB'CI5 + ABC'I6 + ABCI7
  - In general:

$$Z = \sum_{k=0}^{2^n-1} m_k I_k$$

    - in minterm shorthand form for a 2^n:1 Mux

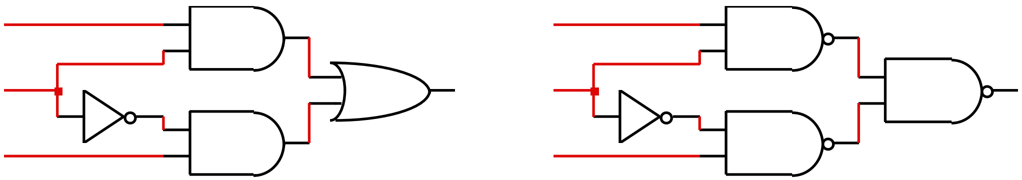




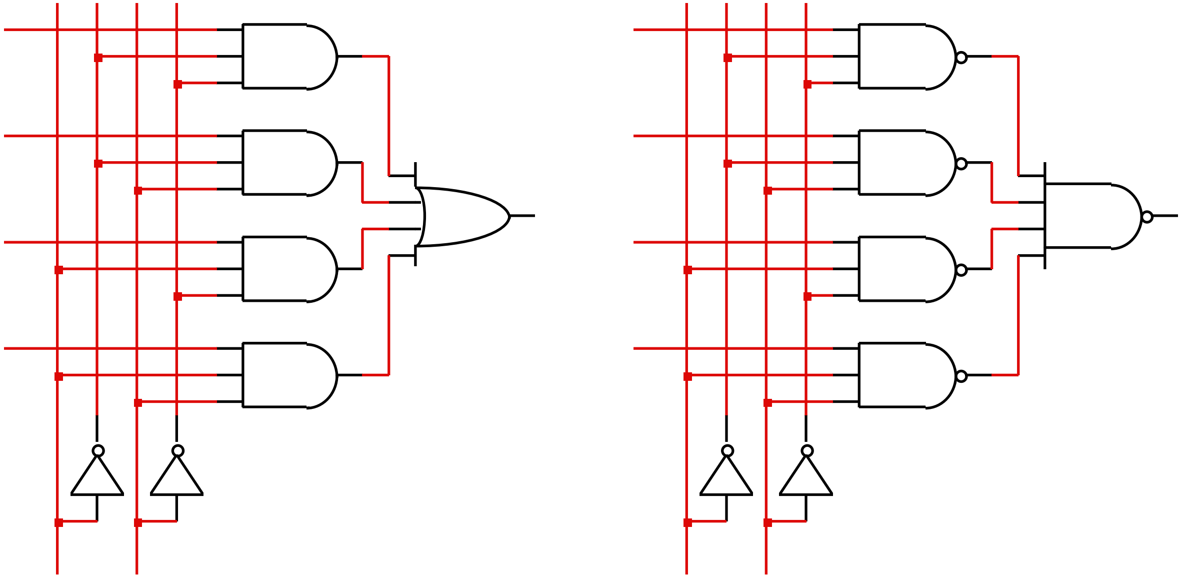


Multiplexer/Selector (cont'd)

- Gate Level Implementation of MUX
  - 2:1 mux

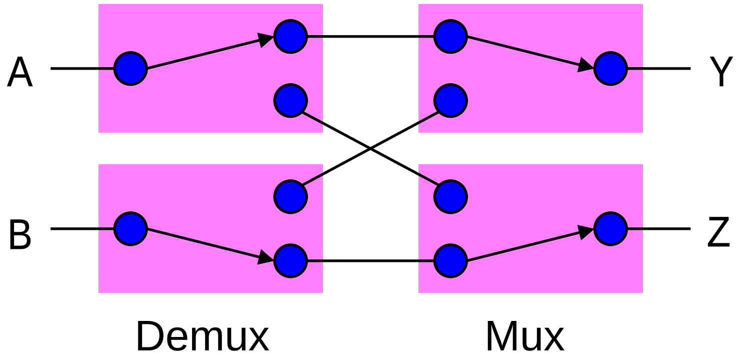


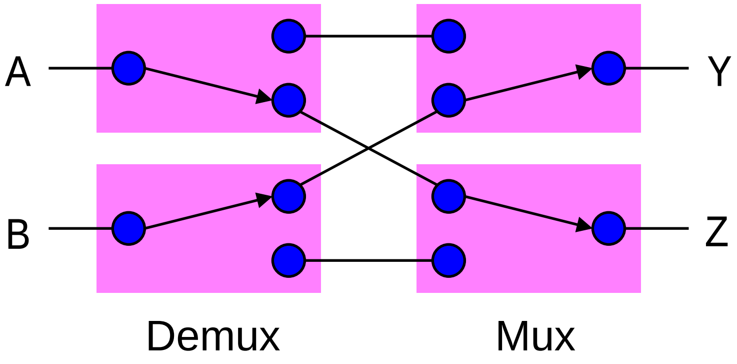
  - 4:1 mux



Multiplexer/Selector (cont'd) - Mux and Demux

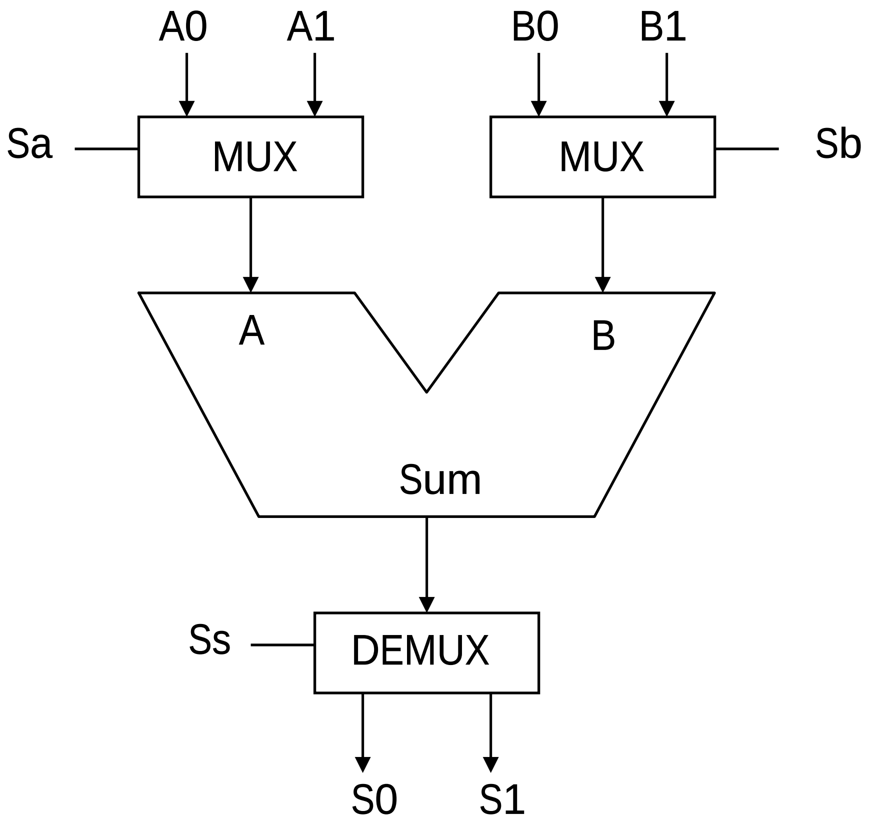
- Switch implementation of multiplexers and demultiplexers
  - can be composed to make arbitrary size switching networks
  - used to implement multiple-source/multiple-destination interconnections





Mux and Demux (cont'd)

- Uses of multiplexers/demultiplexers in multi-point connections

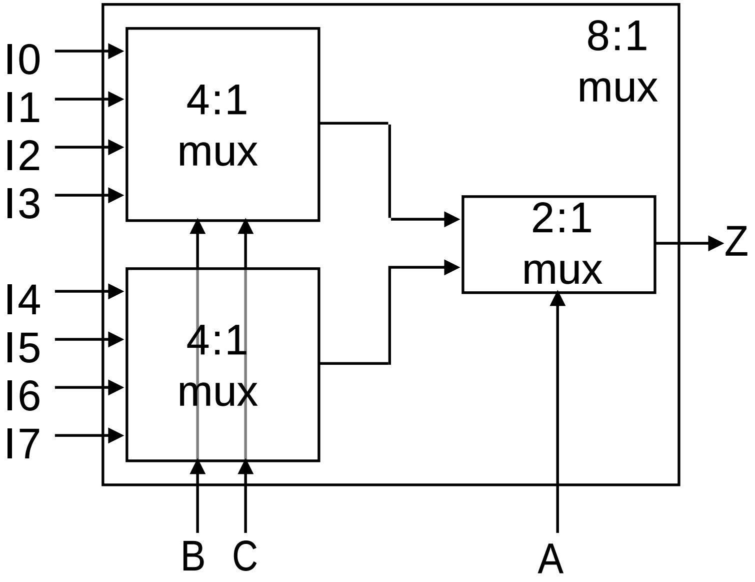


multiple input sources

multiple output destinations

Multiplexer/Selector (cont'd)
- Cascading Multiplexers

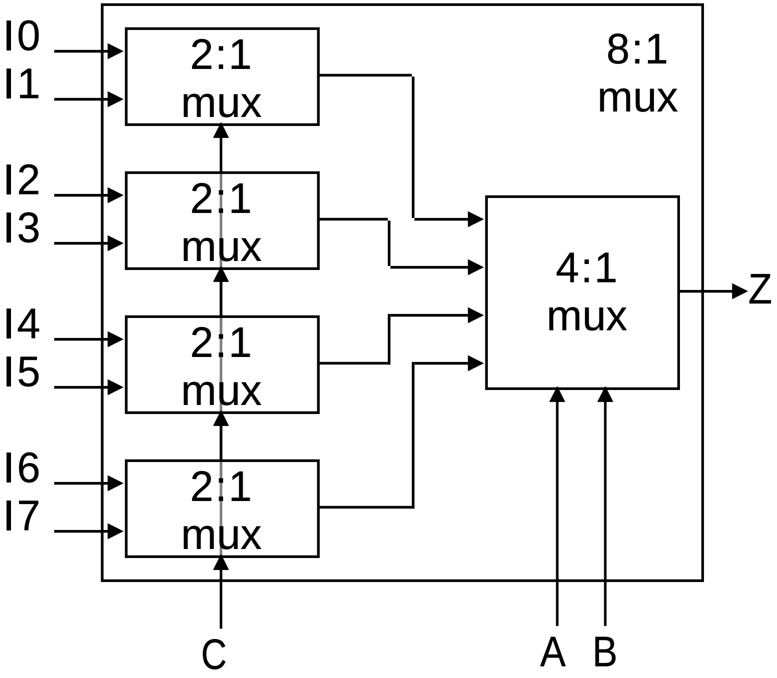
- Large multiplexers can be made by cascading smaller ones



control signals B and C simultaneously choose one of I0, I1, I2, I3 and one of I4, I5, I6, I7

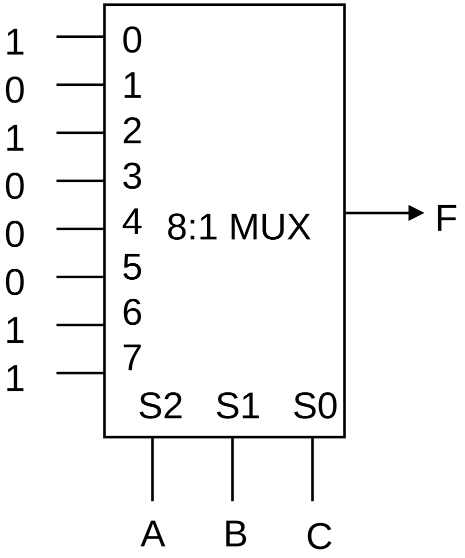
control signal A chooses which of the upper or lower mux's output to gate to Z

alternative implementation



Multiplexer/Selector (cont'd)
- Multiplexers as a Logic Building Block

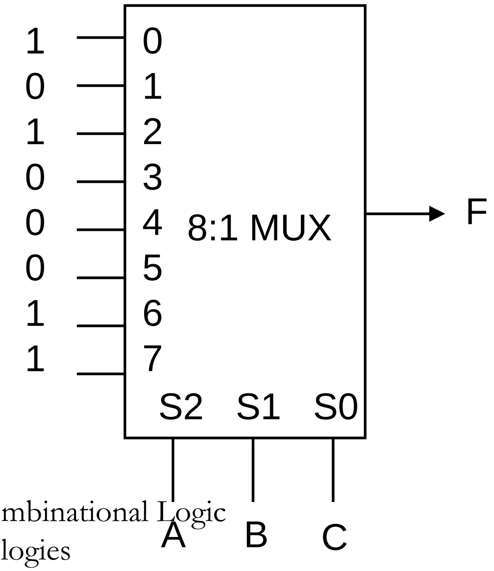
- A 2^n:1 multiplexer can implement any function of n variables
  - with the variables used as control inputs and
  - the data inputs tied to 0 or 1
  - in essence, a lookup table
- Example:
  - F(A,B,C) = m0 + m2 + m6 + m7
    = A'B'C' + A'BC' + ABC' + ABC
    = A'B'C'(1) + A'B'C(0)
    + A'BC'(1) + A'BC(0)
    + AB'C'(0) + AB'C(0)
    + ABC'(1) + ABC(1)



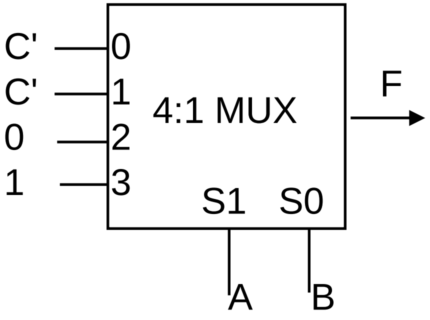
$$F = A'B'C'I_0 + A'B'CI_1 + A'BC'I_2 + A'BCI_3 + AB'C'I_4 + AB'CI_5 + ABC'I_6 + ABCI_7$$

Multiplexers as a Logic Building Block (cont'd)

- A 2^{n-1}:1 multiplexer can implement any function of n variables
  - with n-1 variables used as control inputs and
  - the data inputs tied to the last variable or its complement
- Example:
  - F(A,B,C) = m0 + m2 + m6 + m7
    = A'B'C' + A'BC' + ABC' + ABC
    = A'B'(C') + A'B(C') + AB'(0) + AB(1)



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



Multiplexers as a Logic Building Block (cont'd)

- Generalization

I_0	I_1	. . .	I_{n-1}	I_n	F			
.	.	.	.	0	0	0	1	1
.	.	.	.	1	0	1	0	1
					0	I_n	I_n'	1

n-1 mux control variables

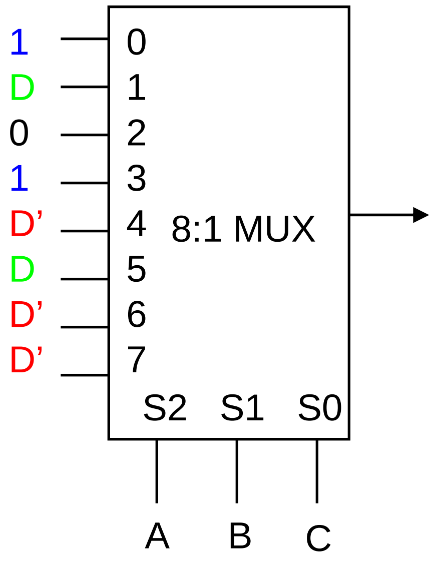
single mux data variable

four possible configurations of truth table rows can be expressed as a function of I_n

- Example: G(A,B,C,D) can be realized by an 8:1 MUX

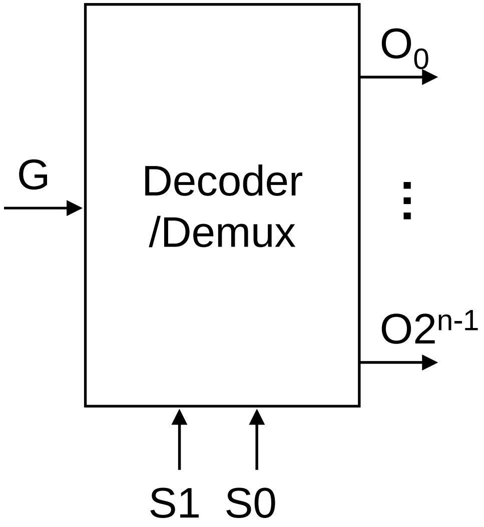
A	B	C	D	G	
0	0	0	0	1	1
0	0	0	1	1	
0	0	1	0	0	D
0	0	1	1	1	
0	1	0	0	0	0
0	1	0	1	0	
0	1	1	0	1	1
0	1	1	1	1	
1	0	0	0	1	D'
1	0	0	1	0	
1	0	1	0	0	D
1	0	1	1	1	
1	1	0	0	1	D'
1	1	0	1	0	
1	1	1	0	1	D'
1	1	1	1	0	

choose A,B,C as control variables



Basic Logic Components (cont'd) - Demultiplexer/Decoder

- Decoders/demultiplexers
  - single data input, n control inputs, 2^n outputs
  - control inputs (called “selects” (S)) represent binary index of output to which the input is connected
  - data input usually called “enable” (G)
- Functional description



$$\mathit{Input}: \quad G = \{0,1\}$$

$$S_i = \{0,1\} \ \ \mathit{where}\ i = 0, 1, \ldots, n\text{-}1$$

$$\mathit{Output}: \quad O_j = \{0,1\} \ \ \mathit{where}\ \ j = 0, 1, \ldots, 2^n\text{-}1$$

$$\mathit{Function}: \ O_j = \begin{cases} 1 \ \ \mathit{where}\ \ j = S = \sum_{i=0}^{n-1} S_i 2^i \\ 0 \ \ \mathit{otherwise} \end{cases}$$

Demultiplexer/Decoder (cont'd)

- Boolean equation of Decoder/Demux
  - General form for n-selects:
    - m_j refers to minterm

$$O_j = G \cdot m_j = \begin{cases} G \quad where \ j = S = \sum_{i=0}^{i=n-1} S_i 2^i \\ 0 \quad otherwise \end{cases}$$

- n = 1

1:2 Decoder:

O0 = G • S'
O1 = G • S

n = 2

2:4 Decoder:

O0 = G • S1' • S0'
O1 = G • S1' • S0
O2 = G • S1 • S0'
O3 = G • S1 • S0

- n = 3

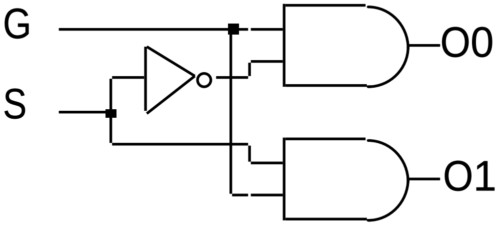
3:8 Decoder:

O0 = G • S2' • S1' • S0'
O1 = G • S2' • S1' • S0
O2 = G • S2' • S1 • S0'
O3 = G • S2' • S1 • S0
O4 = G • S2 • S1' • S0'
O5 = G • S2 • S1' • S0
O6 = G • S2 • S1 • S0'
O7 = G • S2 • S1 • S0

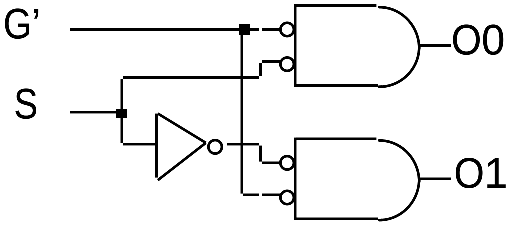
Demultiplexer/Decoder (cont'd)

- Gate level implementation of Demux
  - 1:2 decoders

active-high
enable

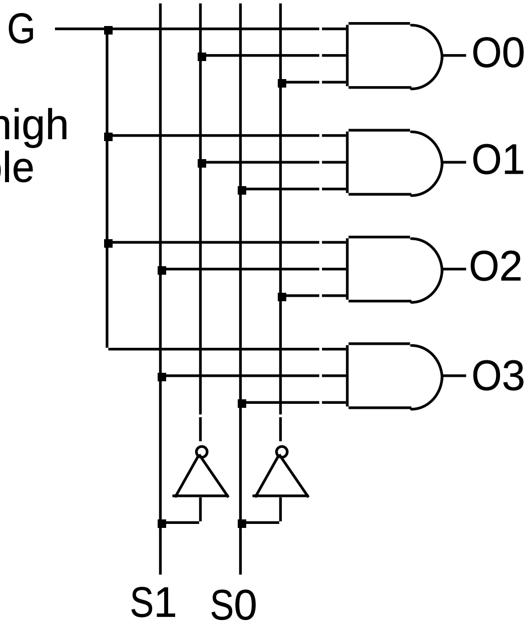


active-low
enable

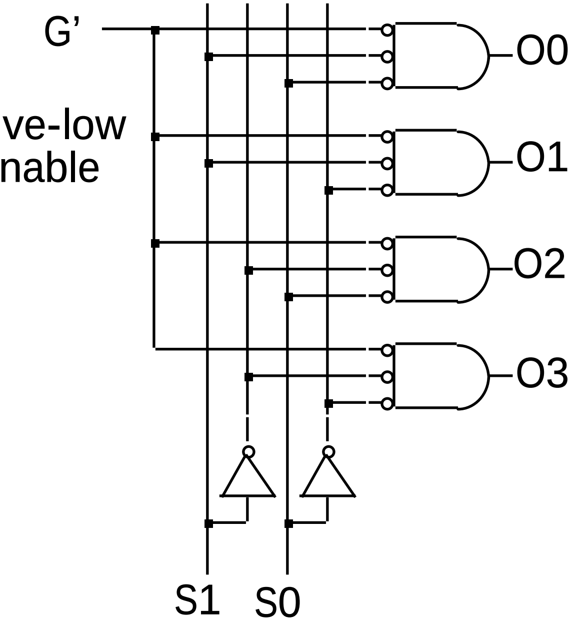


  - 2:4 decoders

active-high
enable

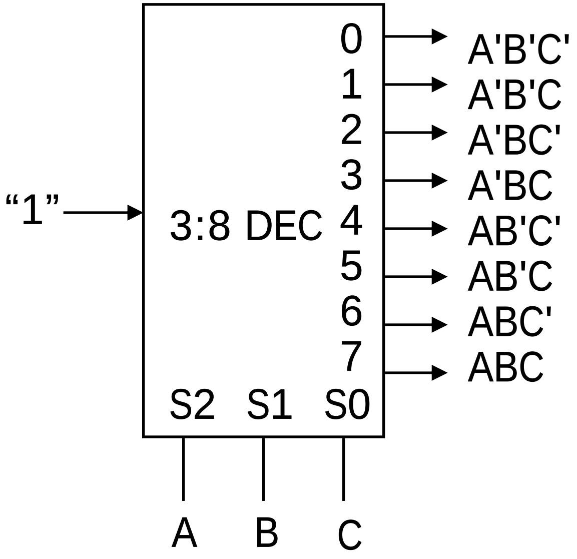


active-low
enable



Demultiplexer/Decoder (cont'd)

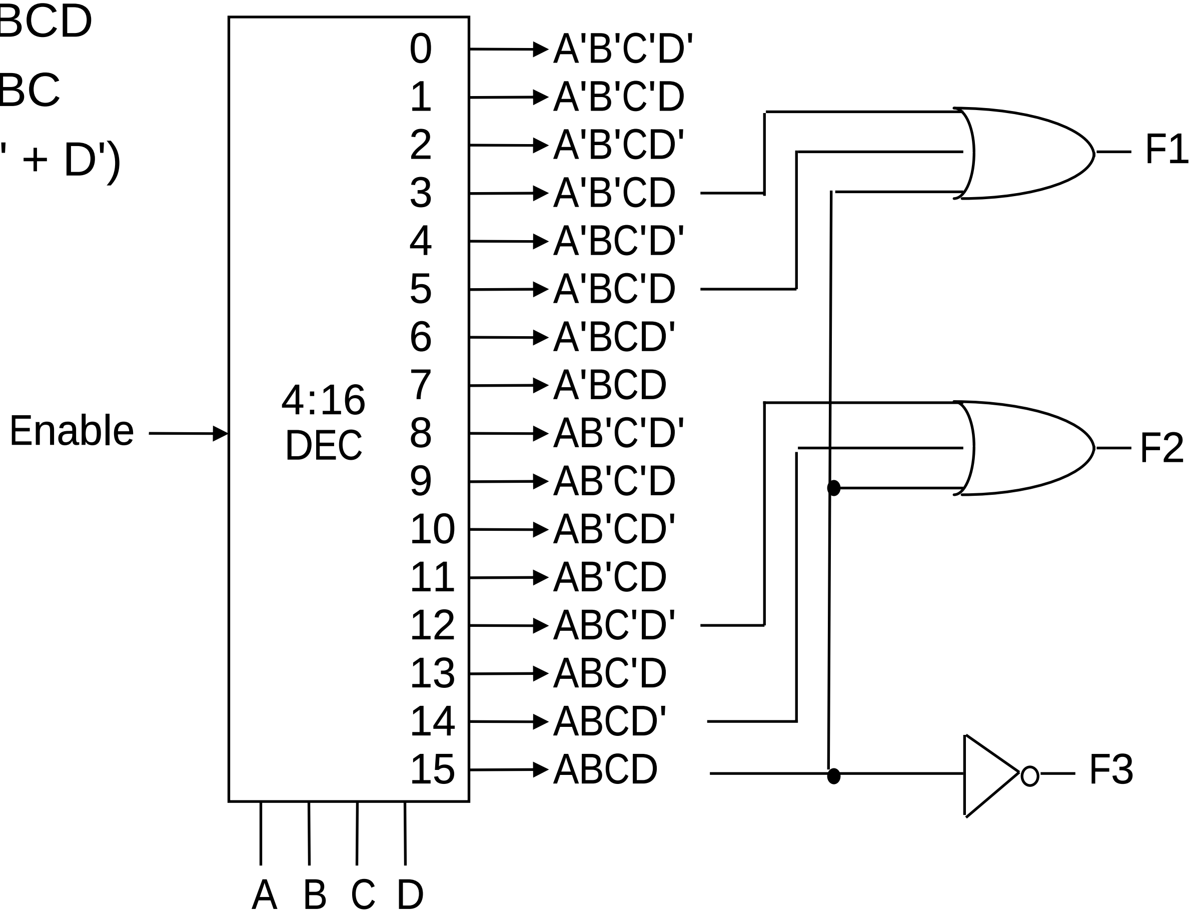
- Demux as a general-purpose building block
  - A n:2^n decoder can implement any function of n variables
    - with the variables used as control inputs
    - the enable inputs tied to 1 and
    - the appropriate minterms summed to form the function



demultiplexer generates appropriate minterm based on control signals (it "decodes" control signals)

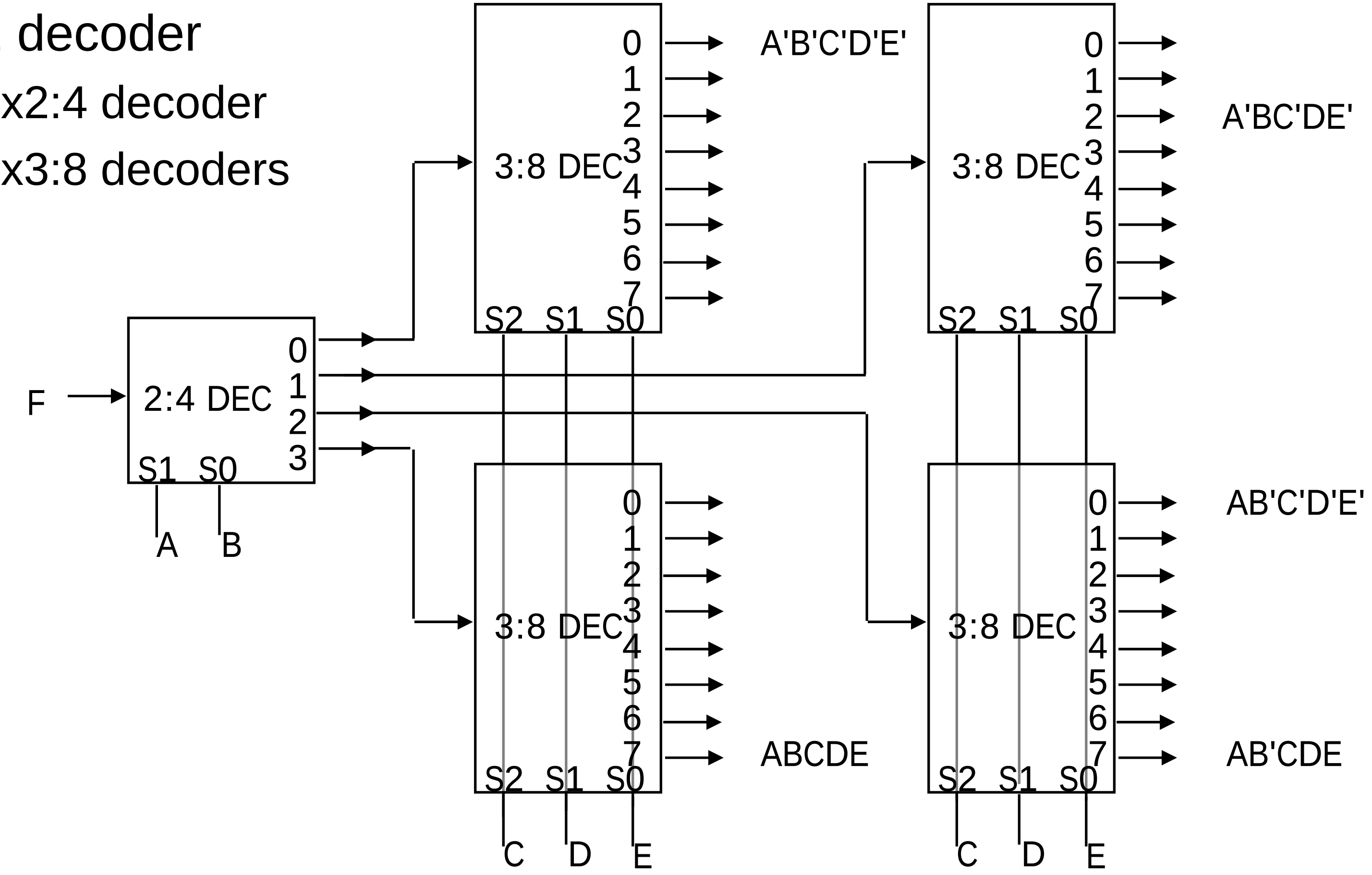
Demultiplexer/Decoder (cont'd)

- Demux as a general-purpose building block
  - F1 = A'BC'D + A'B'CD + ABCD
  - F2 = ABC'D' + ABC
  - F3 = (A' + B' + C' + D')



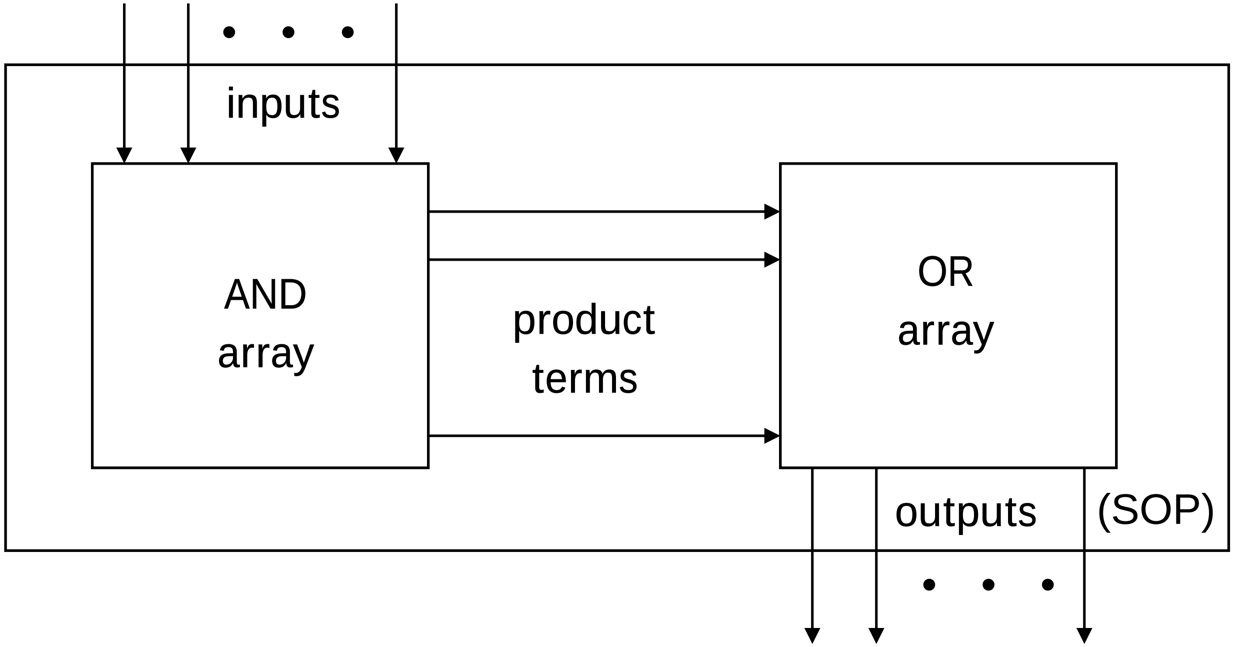
Demultiplexer/Decoder (cont'd)

- Cascading decoders
  - 5:32 decoder
    - 1x2:4 decoder
    - 4x3:8 decoders



Basic Logic Components (cont'd) - Programmable Logic Array

- Pre-fabricated building block of many AND/OR gates
  - actually NOR or NAND
  - "personalized" by making/breaking connections among the gates
  - programmable array block diagram for sum of products form



Programmable Logic Array (cont'd) - Enabling Concept

- Shared product terms among outputs

example:

F0 = A + B' C'
F1 = A C' + A B
F2 = B' C' + A B
F3 = B' C + A

personality matrix

product	inputs			outputs			
term	A	B	C	F0	F1	F2	F3
AB	1	1	–	0	$\boxed{1}$	$\boxed{1}$	0
B'C	–	0	1	0	0	0	1
AC'	1	–	0	0	1	0	0
B'C'	–	0	0	$\boxed{1}$	0	$\boxed{1}$	0
A	1	–	–	$\boxed{1}$	0	0	$\boxed{1}$

reuse of terms

input side:

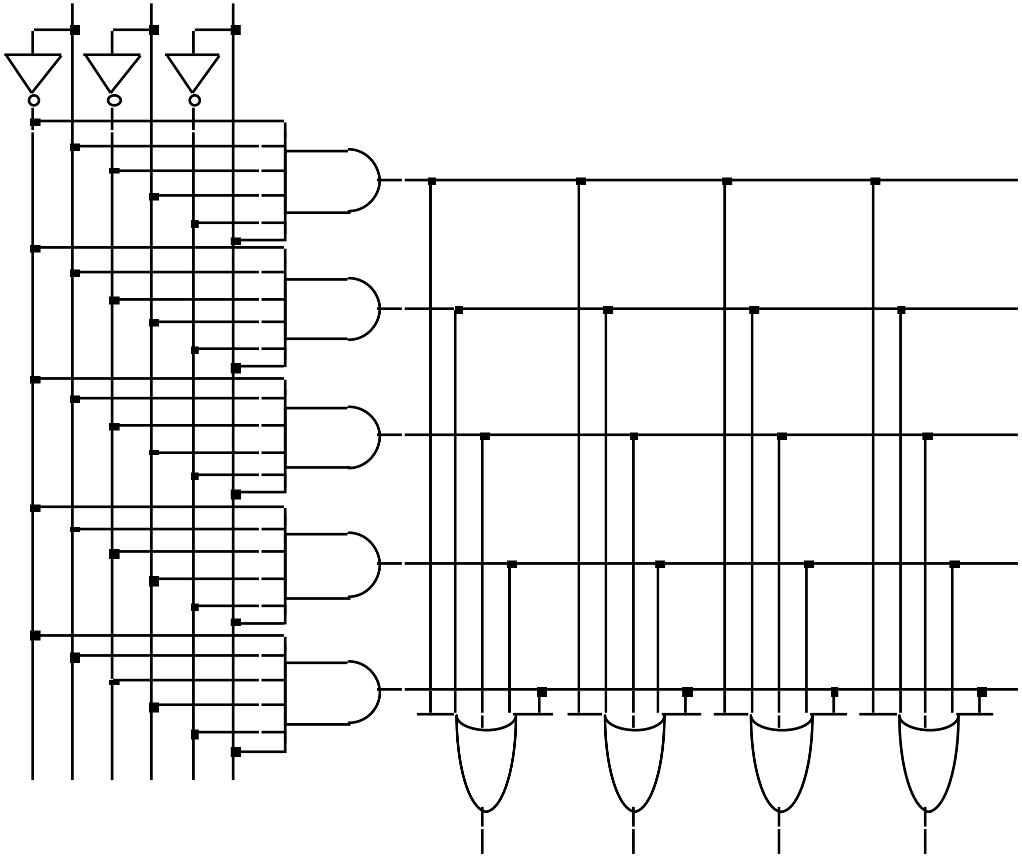
1 = uncomplemented in term
0 = complemented in term
– = does not participate

output side:

1 = term connected to output
0 = no connection to output

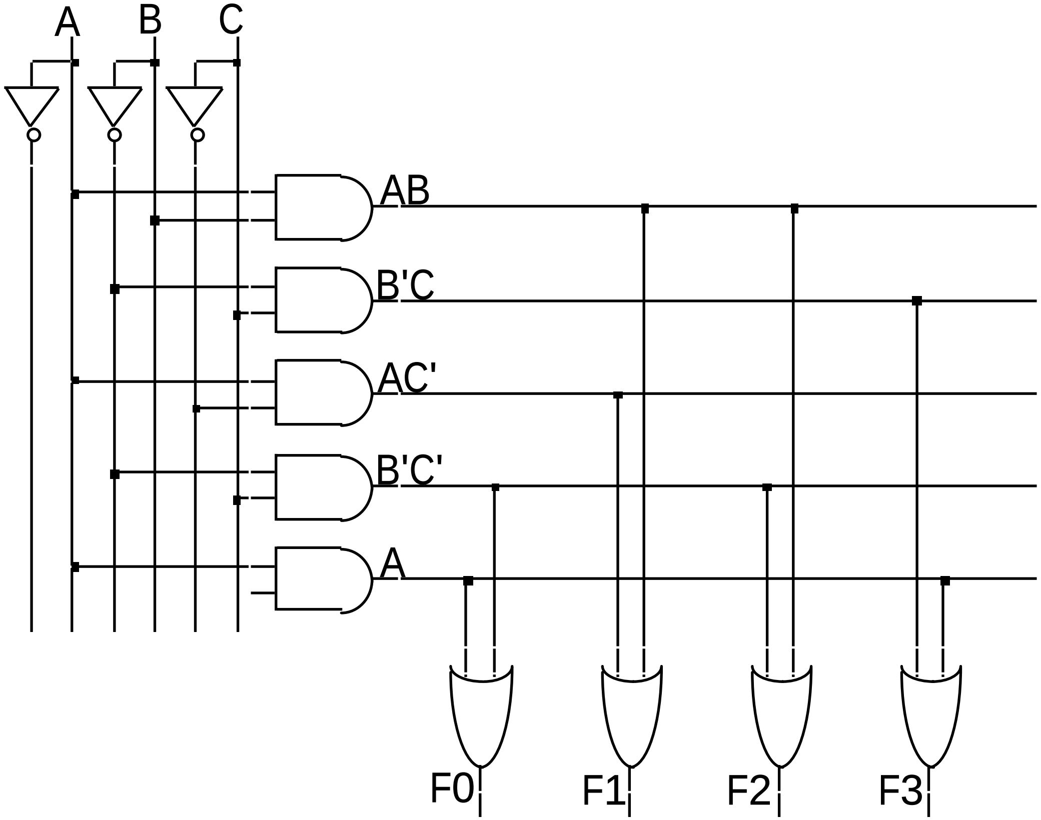
Programmable Logic Array (cont'd) - Before Programming

- All possible connections are available before "programming"
  - in reality, all AND and OR gates are NANDs



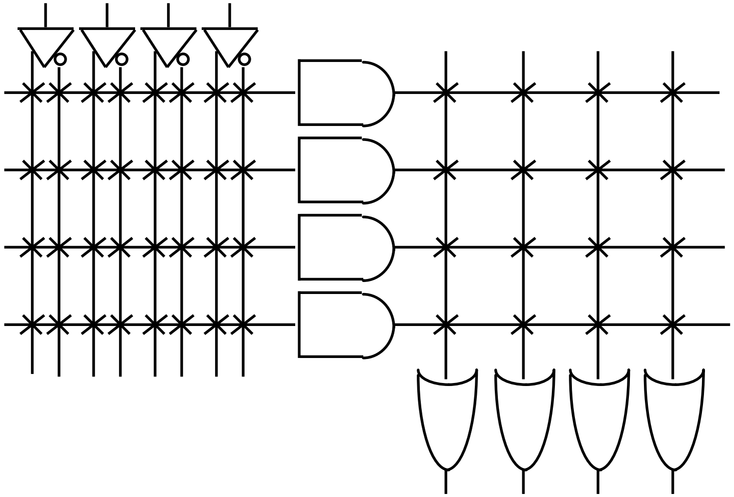
Programmable Logic Array (cont'd) - After Programming

- Unwanted connections are "blown"
  - fuse (normally connected, break unwanted ones)
  - anti-fuse (normally disconnected, make wanted connections)
  - memory cell
    - 0 or 1 can be stored indicating whether there is to be a connection or not.



Programmable Logic Array (cont'd)
- Alternate Representation for High Fan-in Structures

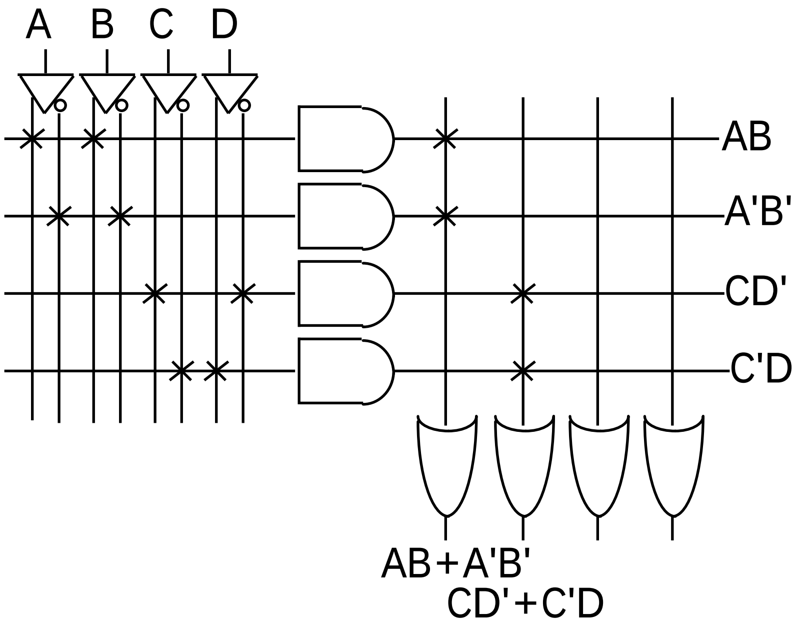
- Short-hand notation so we don't have to draw all the wires
  - × signifies a connection is present and perpendicular signal is an input to gate



notation for implementing

F0 = A B + A' B'

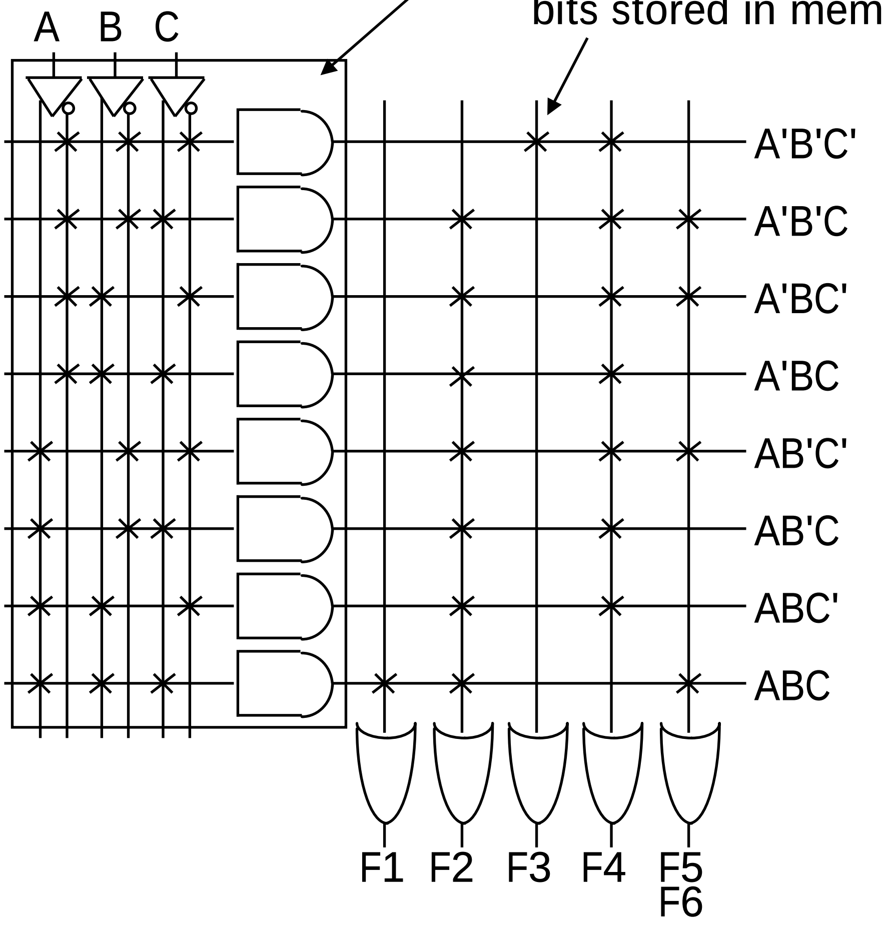
F1 = C D' + C' D



Programmable Logic Array (cont'd) - Example

- Multiple functions of A, B, C
  - F1 = A B C
  - F2 = A + B + C
  - F3 = A' B' C'
  - F4 = A' + B' + C'
  - F5 = A xor B xor C
  - F6 = A xnor B xnor C

A	B	C	F1	F2	F3	F4	F5	F6
0	0	0	0	0	1	1	0	0
0	0	1	0	1	0	1	1	1
0	1	0	0	1	0	1	1	1
0	1	1	0	1	0	1	0	0
1	0	0	0	1	0	1	1	1
1	0	1	0	1	0	1	0	0
1	1	0	0	1	0	1	0	0
1	1	1	1	1	0	0	1	1

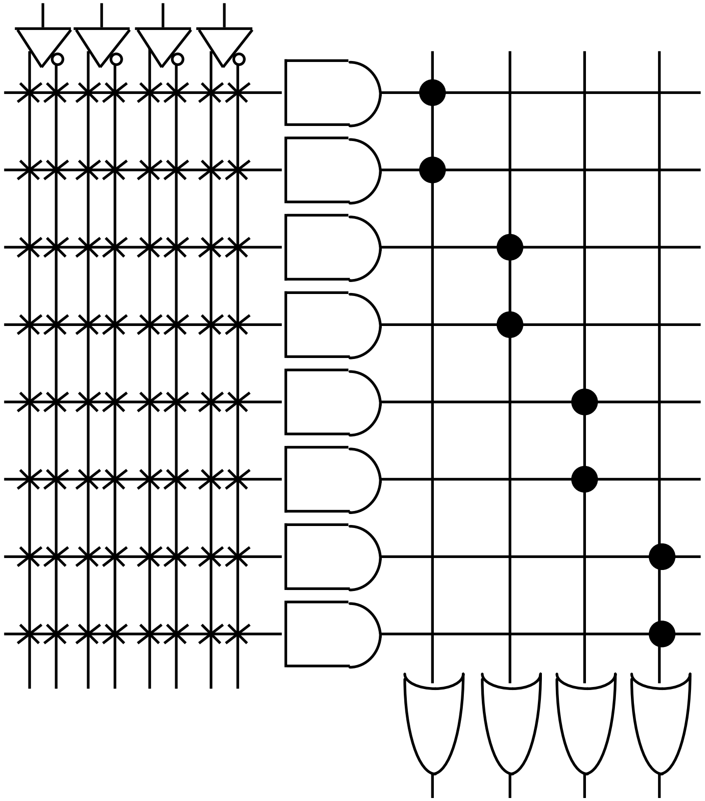


full decoder as for memory address

bits stored in memory

Basic Logic Components (cont'd) - PALs and PLAs

- Programmable logic array (PLA)
  - what we've seen so far
  - unconstrained fully-general AND and OR arrays
- Programmable array logic (PAL)
  - constrained topology of the OR array
  - innovation by Monolithic Memories
  - faster and smaller OR plane



a given column of the OR array has access to only a subset of the possible product terms

PALs and PLAs (cont'd)
- PALs and PLAs: Design Example

- BCD to Gray code converter

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

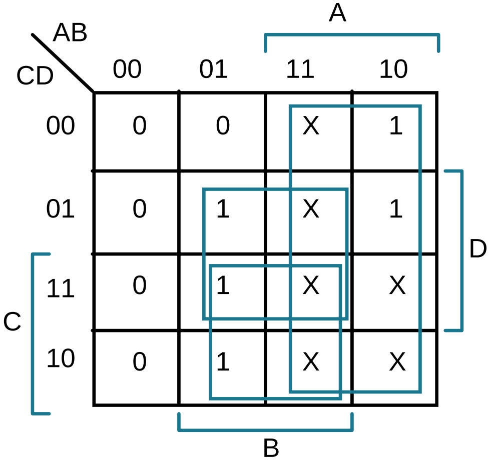
minimized functions:

W = A + BD + BC

X = BC'

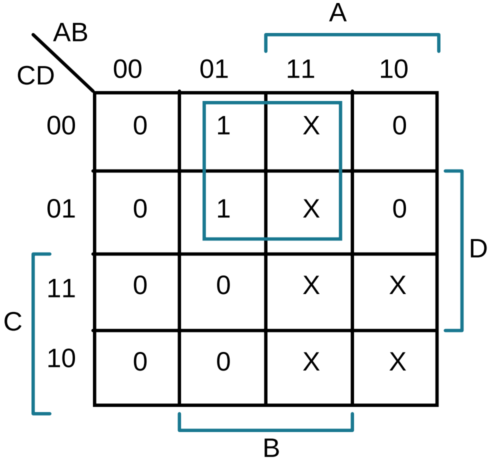
Y = B + C

Z = A'B'C'D + BCD + AD' + B'CD'



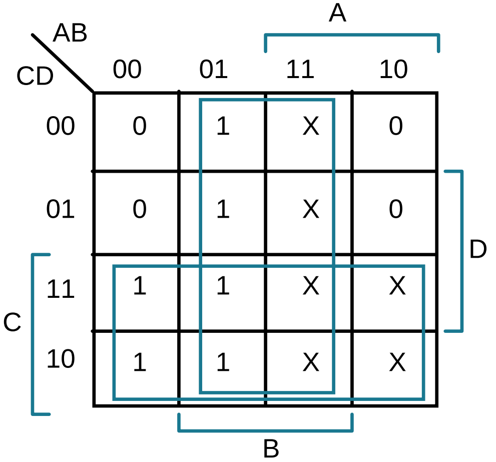
CD \ AB	00	01	11	10
00	0	0	X	1
01	0	1	X	1
11	0	1	X	X
10	0	1	X	X

K-map for W



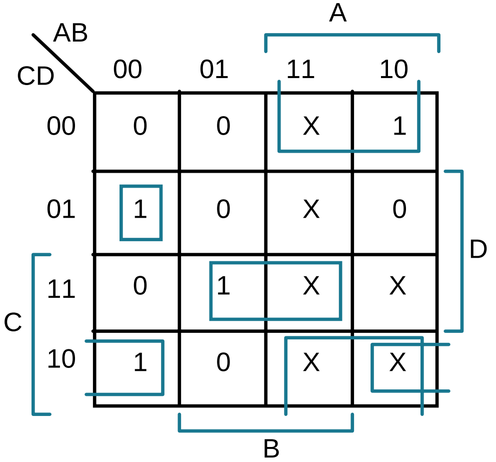
CD \ AB	00	01	11	10
00	0	1	X	0
01	0	1	X	0
11	0	0	X	X
10	0	0	X	X

K-map for X



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

K-map for Y

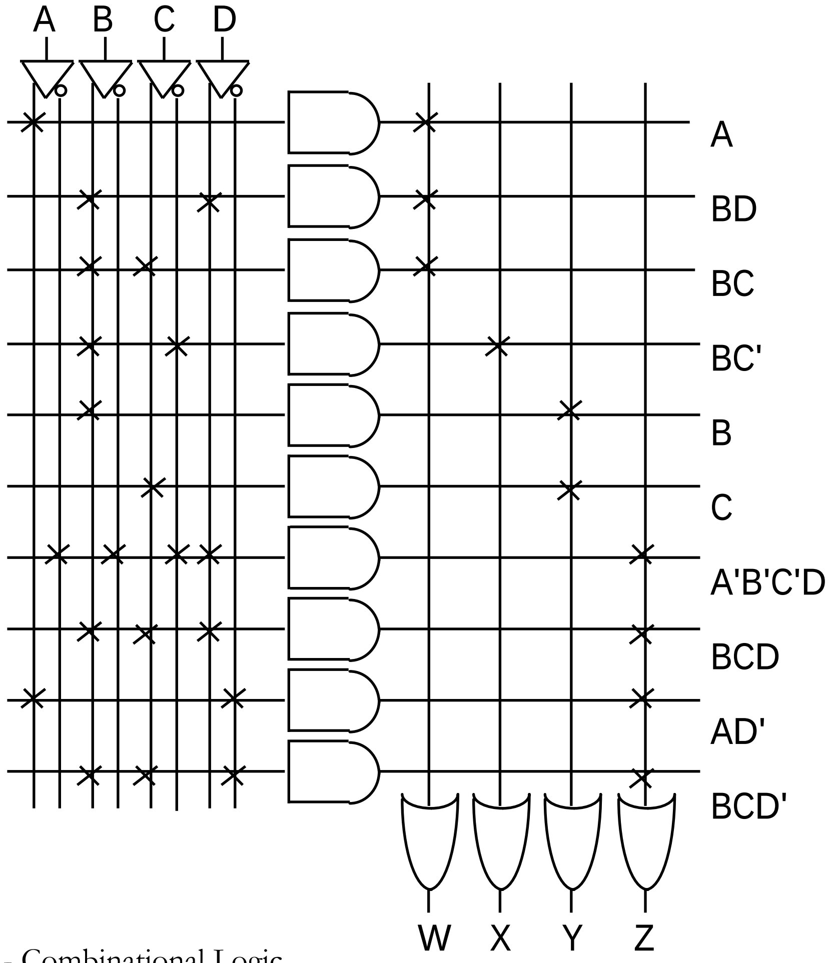


CD \ AB	00	01	11	10
00	0	0	X	1
01	1	0	X	0
11	0	1	X	X
10	1	0	X	X

K-map for Z

PALs and PLAs: Design Example (cont'd)

- Code converter: programmed PLA



minimized functions:

W = A + BD + BC
X = B C'
Y = B + C
Z = A'B'C'D + BCD + AD' + B'CD'

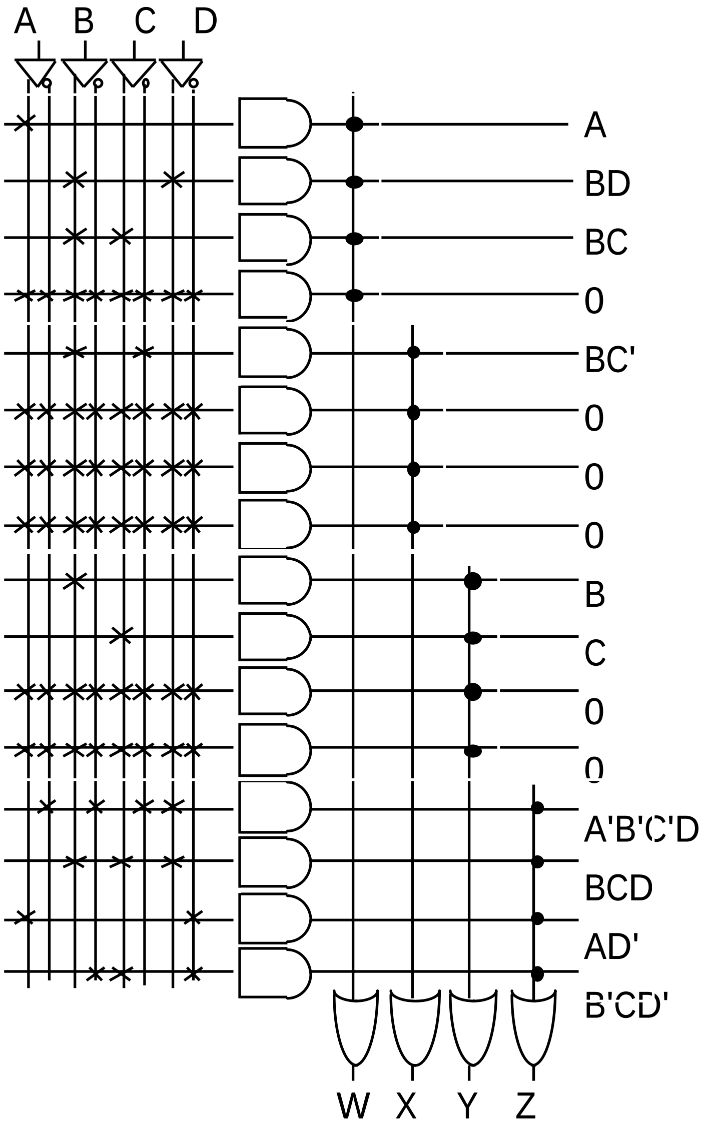
not a particularly good candidate for PLA implementation since no terms are shared among outputs

however, much more compact and regular implementation when compared with discrete AND and OR gates

PALs and PLAs: Design Example (cont'd)

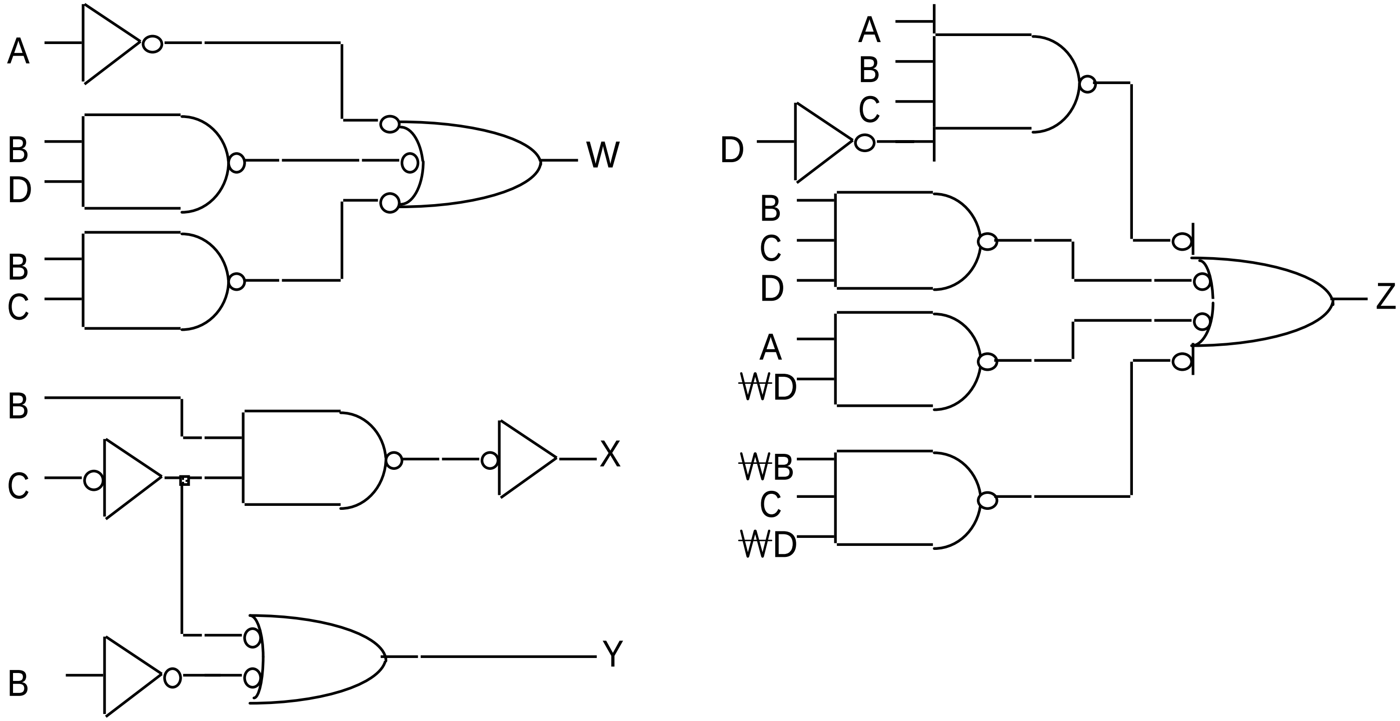
- Code converter: programmed PAL

4 product terms
per each OR gate



PALs and PLAs: Design Example (cont'd)

- Code converter: NAND gate implementation
  - loss or regularity, harder to understand
  - harder to make changes



PALs and PLAs (cont'd)
- PALs and PLAs: Another Design Example

- Magnitude comparator

$$Input: \quad AB, CD \ \ where\ A, B, C, D = \{0,1\}$$

$$Output: \quad EQ, NE, LT, GT = \{0,1\}$$

$$Function: EQ = \begin{cases} 1 \ \ when\ AB = CD \\ 0 \ \ otherwise \end{cases}$$

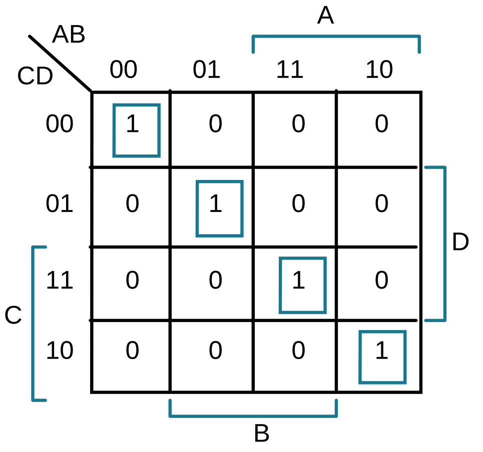
$$NE = \begin{cases} 1 \ \ when\ AB \neq CD \\ 0 \ \ otherwise \end{cases}$$

$$LT = \begin{cases} 1 \ \ when\ AB < CD \\ 0 \ \ otherwise \end{cases}$$

$$GT = \begin{cases} 1 \ \ when\ AB > CD \\ 0 \ \ otherwise \end{cases}$$

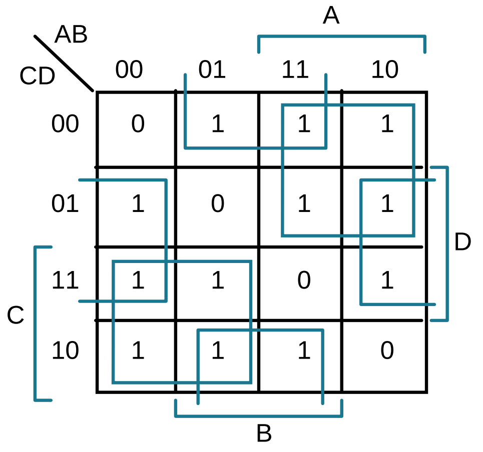
A	B	C	D	EQ	NE	LT	GT
0	0	0	0	1	0	0	0
0	0	0	1	0	1	1	0
0	0	1	0	0	1	1	0
0	0	1	1	0	1	1	0
0	1	0	0	0	1	0	1
0	1	0	1	1	0	0	0
0	1	1	0	0	1	1	0
0	1	1	1	0	1	1	0
1	0	0	0	0	1	0	1
1	0	0	1	0	1	0	1
1	0	1	0	1	0	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	1	0	1
1	1	0	1	0	1	0	1
1	1	1	0	0	1	0	1
1	1	1	1	1	0	0	0

PALs and PLAs: Another Design Example (cont'd)



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

K-map for EQ



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

K-map for NE

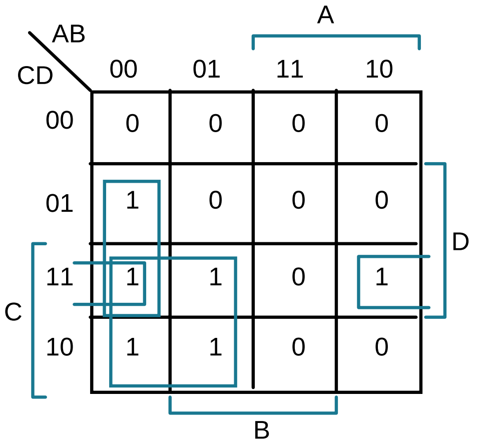
minimized functions:

EQ = A'B'C'D' + A'BC'D
+ ABCD + AB'CD'

NE = AC' + A'C + B'D + BD'

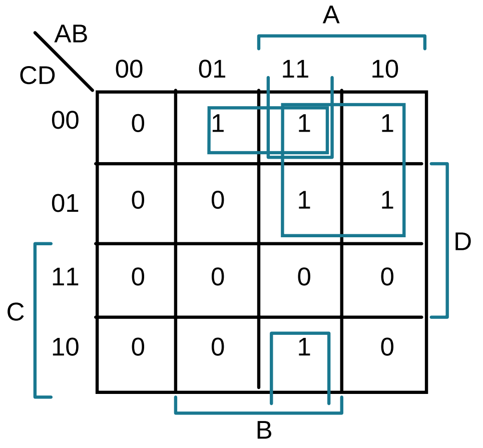
LT = A'C + A'B'D + B'CD

GT = AC' + ABC + BC'D'



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

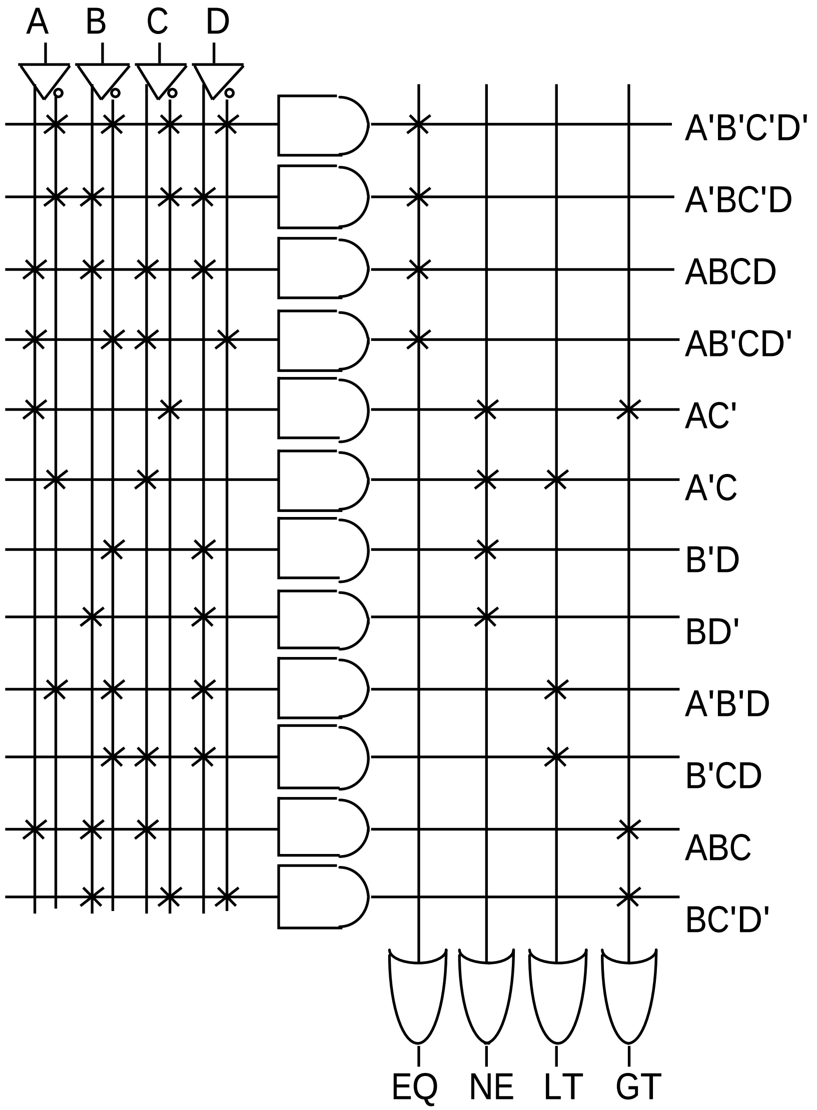
K-map for LT



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

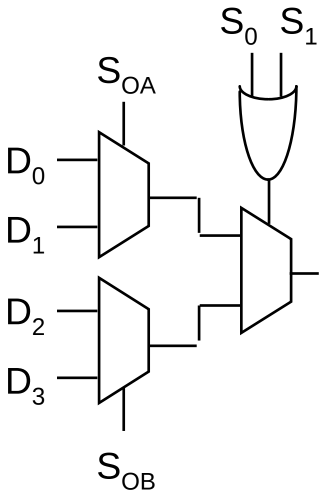
K-map for GT

PALs and PLAs: Another Design Example (cont'd)

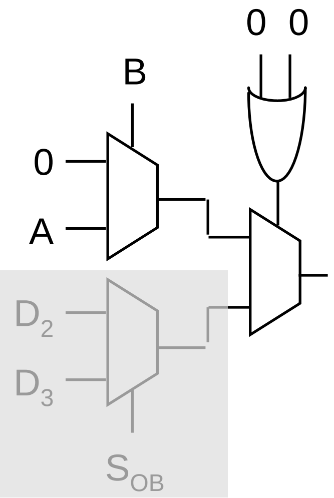


Basic Logic Components (cont'd) - Multiplexer-Based FPGA

- Actel logic module

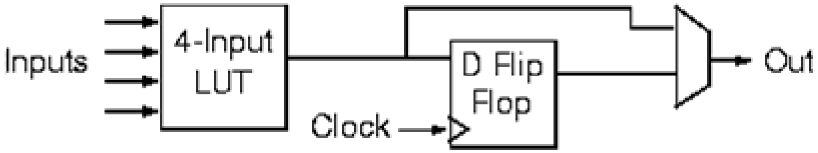


- Example using a logic module
  - AND
    - $Y=(S_0+S_1)'(S'_{OA}D_0+S_{OA}D_1)$ $+(S_0+S_1)(S'_{OB}D_2+S_{OB}D_3)$
    - $Y=(0+0)'(B'0+BA)$ $+(0+0)(S'_{OB}D_2+S_{OB}D_3)$
    - $Y=AB$

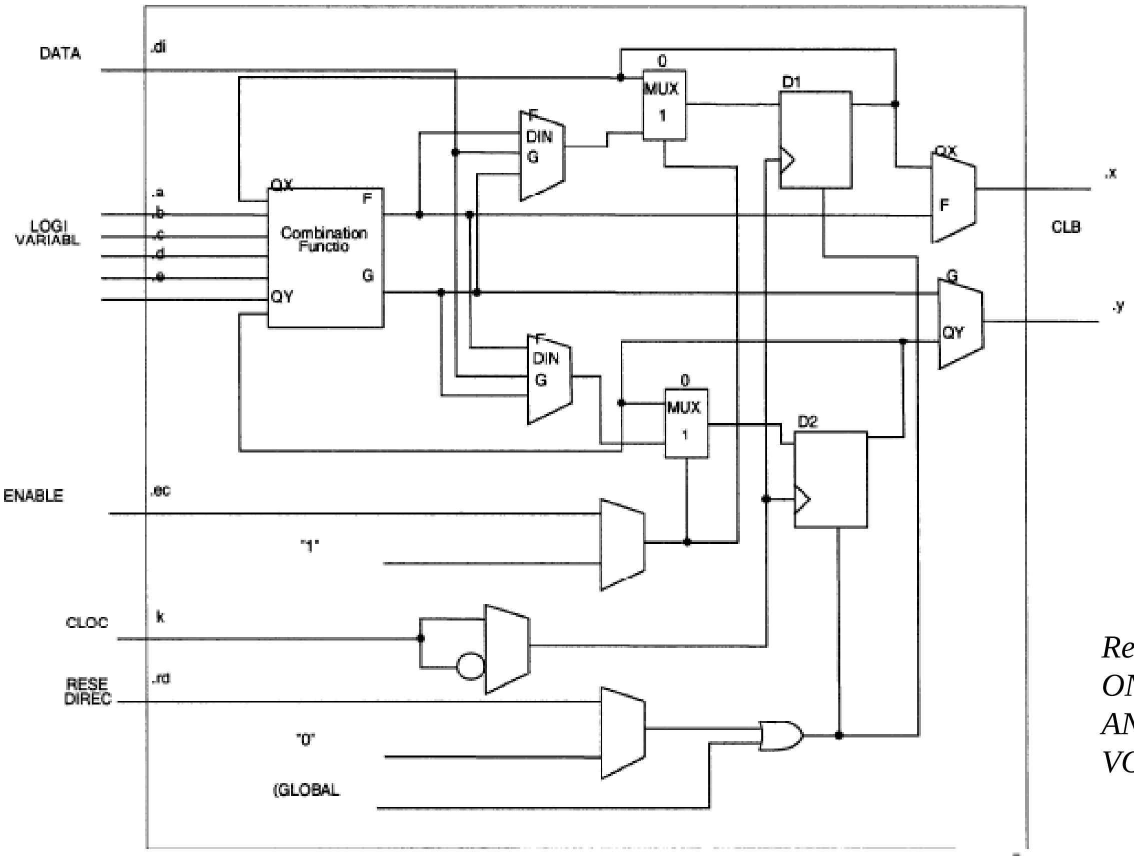


Basic Logic Components (cont'd) - Look-Up Table Based FPGA

- Simplified basic building block of FPGA



- Xilinx Configurable Logic Block (CLB)



Ref.: IEEE TRANSACTIONS ON INSTRUMENTATION AND MEASUREMENT, VOL. 52, NO. 5, OCTOBER 2003

Two-Level and Multilevel Logic
- Regular Logic Structures for Two-Level Logic

- ROM – full AND plane, general OR plane
  - cheap (high-volume component)
  - can implement any function of n inputs
  - medium speed
- PAL – programmable AND plane, fixed OR plane
  - intermediate cost
  - can implement functions limited by number of terms
  - high speed (only one programmable plane that is much smaller than ROM's decoder)
- PLA – programmable AND and OR planes
  - most expensive (most complex in design, need more sophisticated tools)
  - can implement any function up to a product term limit
  - slow (two programmable planes)

Regular Logic Structures for Two-Level Logic (cont'd) - ROM vs. PLA

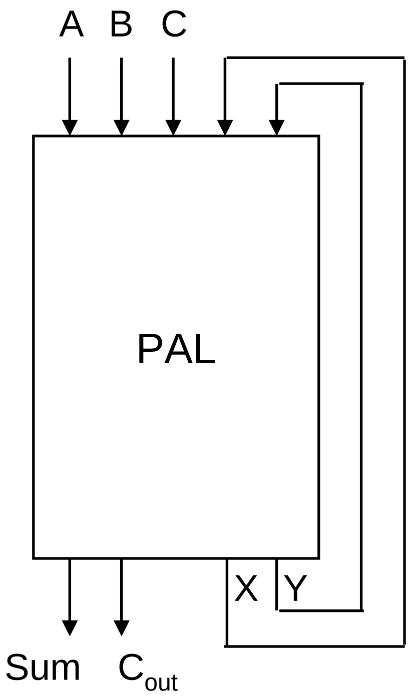
- ROM approach advantageous when
  - design time is short (no need to minimize output functions)
  - most input combinations are needed (e.g., code converters)
  - little sharing of product terms among output functions
- ROM problems
  - size doubles for each additional input
  - can't exploit don't cares
- PLA approach advantageous when
  - design tools are available for multi-output minimization
  - there are relatively few unique minterm combinations
  - many minterms are shared among the output functions
- PAL problems
  - constrained fan-ins on OR plane

Two-Level and Multilevel Logic (cont'd) - Regular Logic Structures for Multilevel Logic

- Difficult to devise a regular structure for arbitrary connections between a large set of different types of gates
    - efficiency/speed concerns for such a structure
    - FPGA: just such programmable multi-level structures
        - programmable multiplexers for wiring
        - lookup tables for logic functions (programming fills in the table)
        - multi-purpose cells (utilization is the big issue)

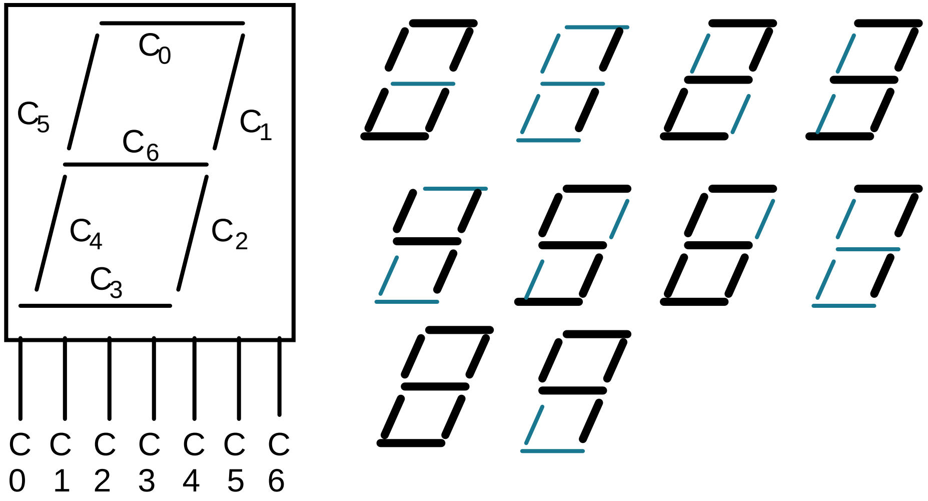
Regular Logic Structures for Multilevel Logic (cont'd)

- Using multiple levels of PALs/PLAs/ROMs
  - output intermediate result
  - make it an input to be used in further logic
  - Example: Full Adder
    - Boolean equations
      - Sum = $A'B'C_{in} + A'BC_{in}' + AB'C_{in}' + ABC_{in}$
      - $C_{out} = AB + BC_{in} + AC_{in}$
    - Boolean equations
      for feedback PAL implementation
      - $X = AB' + A'B$, $Y=AB$
      - Sum = $XC_{in}' + X'C_{in}$
      - $C_{out} = XC_{in} + Y$

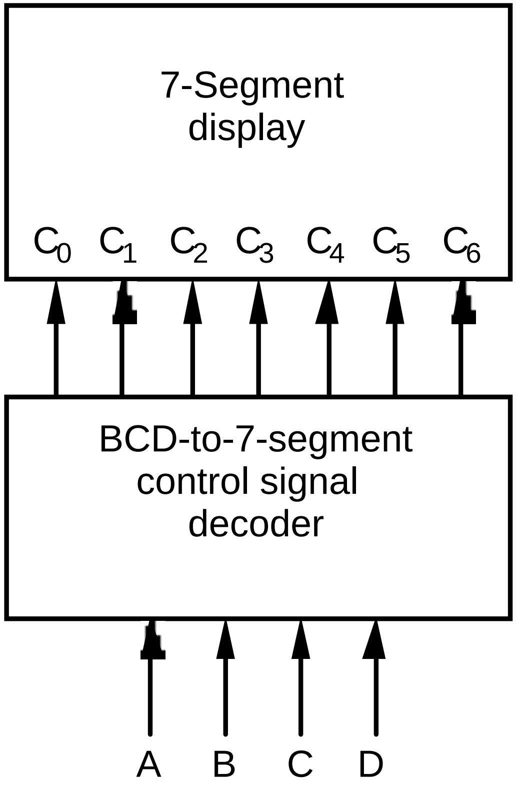


Two-level and Multilevel Logic (cont'd) - 7-Segment Display Decoder

- Understanding the problem:
  - input is a 4 bit BCD digit
  - output is the control signals for the display
  - 4 inputs A, B, C, D
  - 7 outputs C0 ~ C6



7-segment display


7-Segment display

BCD-to-7-segment control signal decoder

Block Diagram

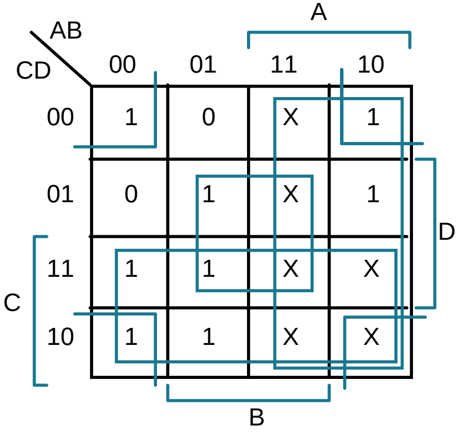
7-Segment Display Decoder (cont'd)

- Truth table

ABCD	C0	C1	C2	C3	C4	C5	C6
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
101x	x	x	x	x	x	x	x
11xx	x	x	x	x	x	x	x

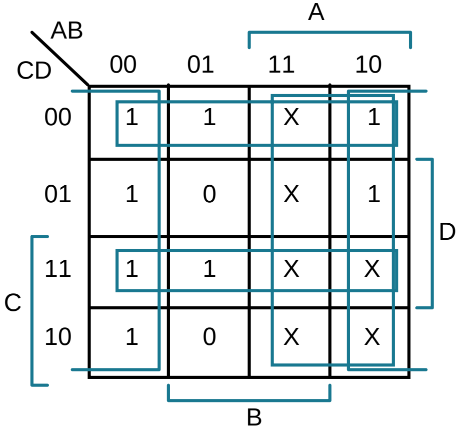
- Formulate the problem in terms of a truth table
- Choose implementation target:
  - if ROM, we are done
  - don't cares imply PAL/PLA may be attractive
- Follow implementation procedure:
  - hand reduced K-maps
  - espresso

7-Segment Display Decoder (cont'd)



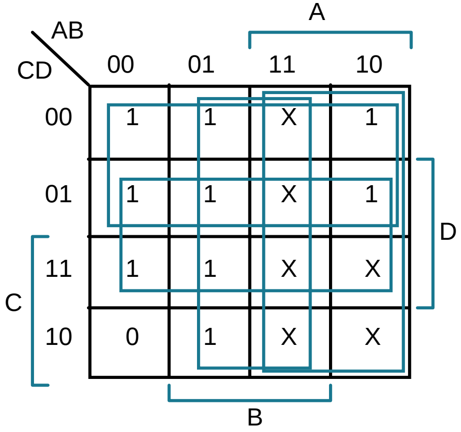
CD \ AB	00	01	11	10
00	1	0	X	1
01	0	1	X	1
11	1	1	X	X
10	1	1	X	X

K-map for C_0



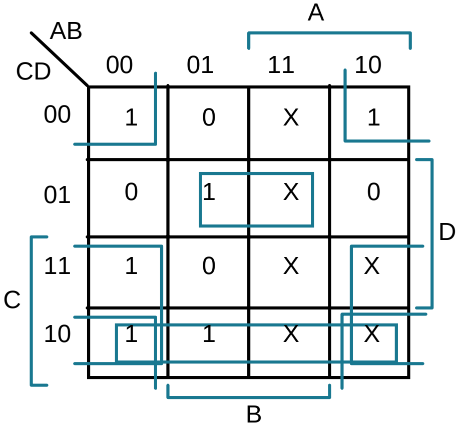
CD \ AB	00	01	11	10
00	1	1	X	1
01	1	0	X	1
11	1	1	X	X
10	1	0	X	X

K-map for C_1



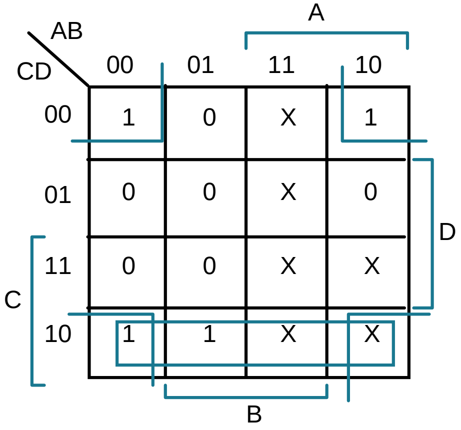
CD \ AB	00	01	11	10
00	1	1	X	1
01	1	1	X	1
11	1	1	X	X
10	0	1	X	X

K-map for C_2



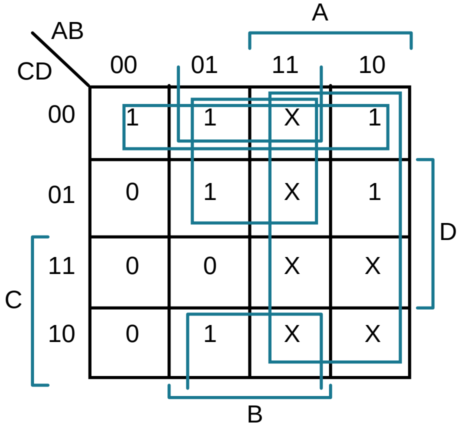
CD \ AB	00	01	11	10
00	1	0	X	1
01	0	1	X	0
11	1	0	X	X
10	1	1	X	X

K-map for C_3



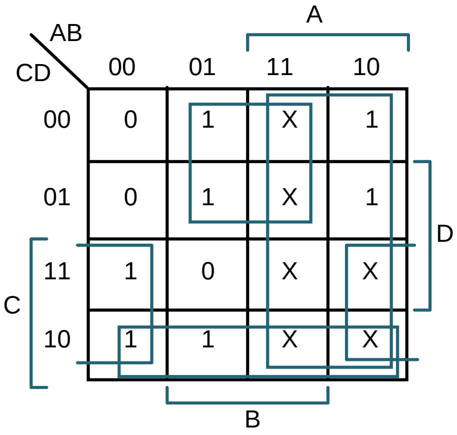
CD \ AB	00	01	11	10
00	1	0	X	1
01	0	0	X	0
11	0	0	X	X
10	1	1	X	X

K-map for C_4



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

K-map for C_5



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

K-map for C_6

C0 = A + B D + C + B' D'
C1 = C' D' + C D + B'
C2 = B + C' + D

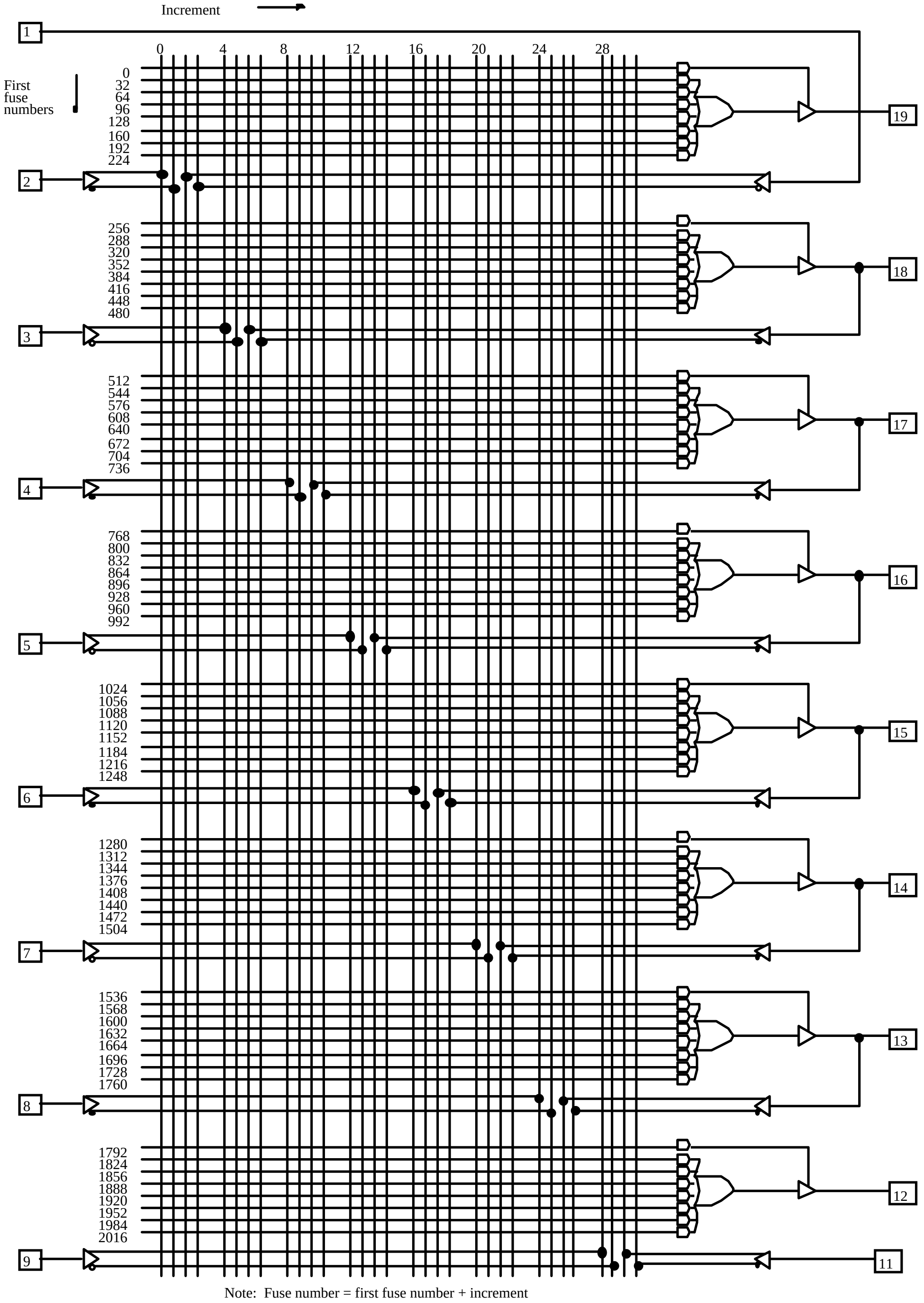
14 Unique Product Terms

C3 = B' D' + C D' + B C' D + B' C
C4 = B' D' + C D'
C5 = A + C' D' + B D' + B C'
C6 = A + C D' + B C' + B' C

7-Segment Display Decoder (cont'd)

16H8PAL
(10 external inputs, 6 feedback inputs 8 outputs)

Can Implement the function

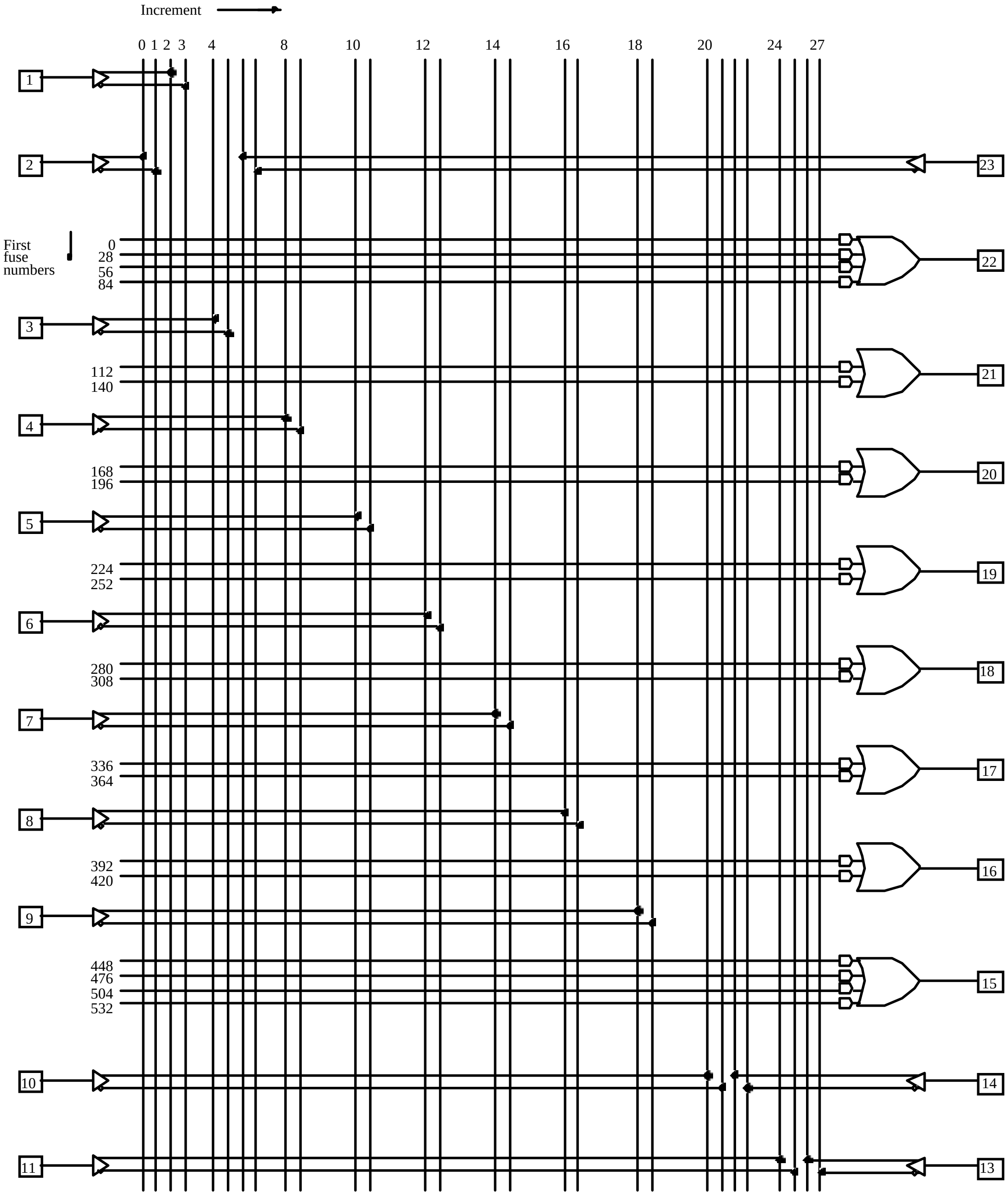


Note: Fuse number = first fuse number + increment

7-Segment Display Decoder (cont’d)

14H8PAL
(14 inputs, 8 outputs,
but only two
4-input product terms
can be computed.)

Cannot Implement the function



Note: Fuse number = first fuse number + increment

7-Segment Display Decoder (cont’d)

CAD (espresso) input

```
.i 4
.o 7
.ilb a b c d
.ob c0 c1 c2 c3 c4 c5 c6
.p 16
0000 1111110
0001 0110000
0010 1101101
0011 1111001
0100 0110011
0101 1011011
0110 1011111
0111 1110000
1000 1111111
1001 1110011
1010 -------
1011 -------
1100 -------
1101 -------
1110 -------
1111 -------
.e
```

CAD (espresso) output

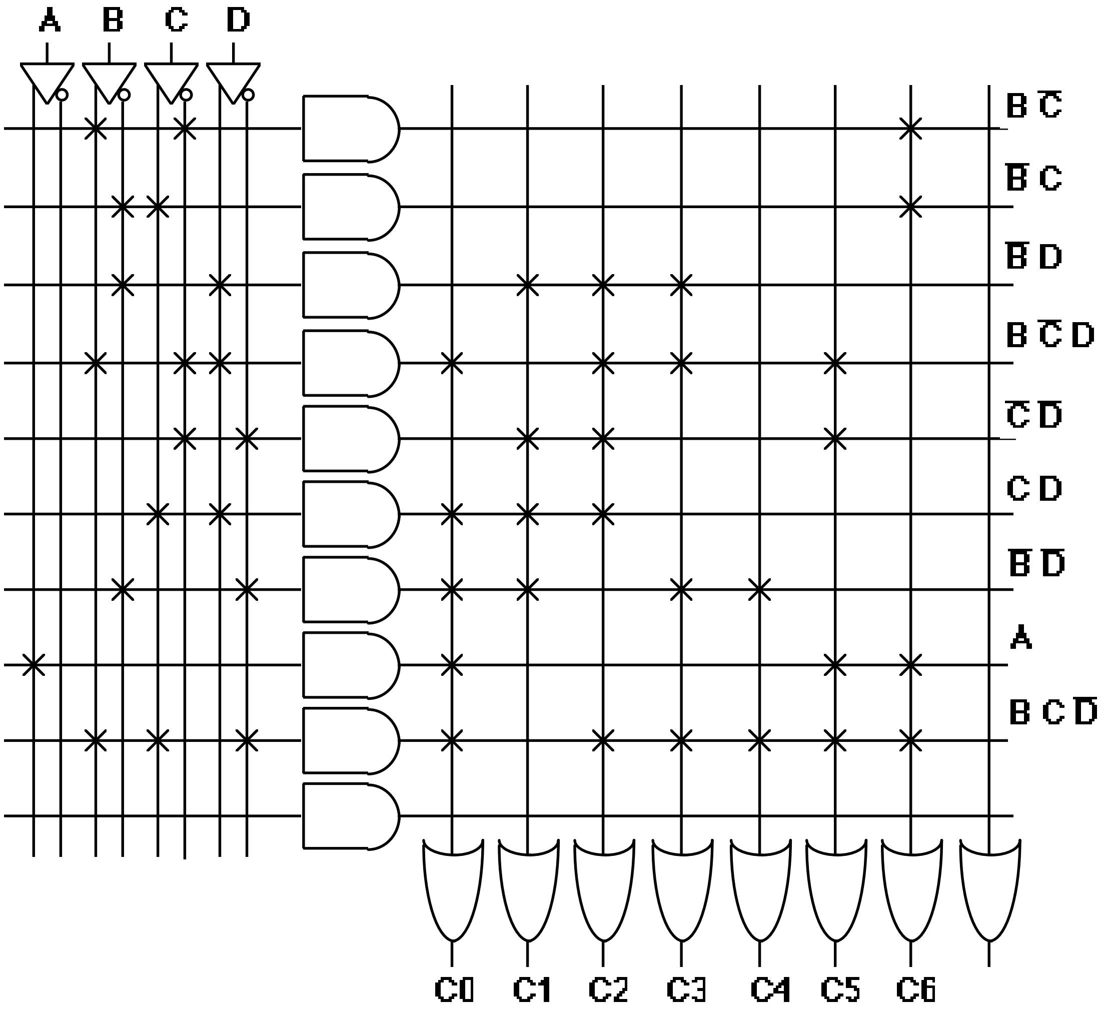
```
.i 4
.o 7
.ilb a b c d
.ob c0 c1 c2 c3 c4 c5 c6
.p 9
-10-  0000001
-01-  0001001
-0-1  0110000
-101 1011010
--00  0110010
--11  1110000
-0-0  1101100
1---   1000011
-110 1011111
.e
```

C0 = B C' D + C D + B' D' + B C D' + A
C1 = B' D + C' D' + C D + B' D'
C2 = B' D + B C' D + C' D' + C D + B C D'
C3 = B C' D + B' D + B' D' + B C D'
C4 = B' D' + B C D'
C5 = B C' D + C' D' + A + B C D'
C6 = B' C + B C' + B C D' + A

63 Literals, 20 Gates
9 Unique Product Terms!

The results are more complex than those of manual method. However, #unique product terms has been reduced 15->9

7-Segment Display Decoder (cont'd) - PLA Implementation



7-Segment Display Decoder (cont'd) - Multilevel Implementation

X = C' + D'

Y = B' C'

C0 = C3 + A' B X' + A D Y

C1 = Y + A' C5' + C' D' C6

C2 = C5 + A' B' D + A' C D

C3 = C4 + B D C5 + A' B' X'

C4 = D' Y + A' C D'

C5 = C' C4 + A Y + A' B X

C6 = A C4 + C C5 + C4' C5 + A' B' C

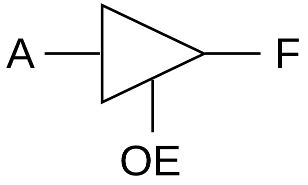
52 literals

33 gates

Ineffective use of don't cares

Non-Gate Logic
- Tri-State Outputs

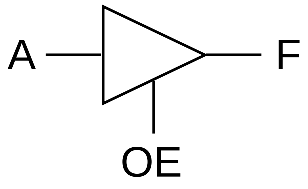
- The Third State
  - Logic States: "0", "1"
  - Don't Care/Don't Know State: "X" (must be some value in real circuit!)
  - Third State: "Z“: high impedance: infinite resistance, no connection
- Tri-state gates:
  - output values are "0", "1", and "Z“
  - additional input: output enable (OE)
    - When OE is high, this gate is a non-inverting "buffer"
    - When OE is low, it is as though the gate was disconnected from the output!
  - This allows more than one gate to be connected to the same output wire, as long as only one has its output enabled at the same time



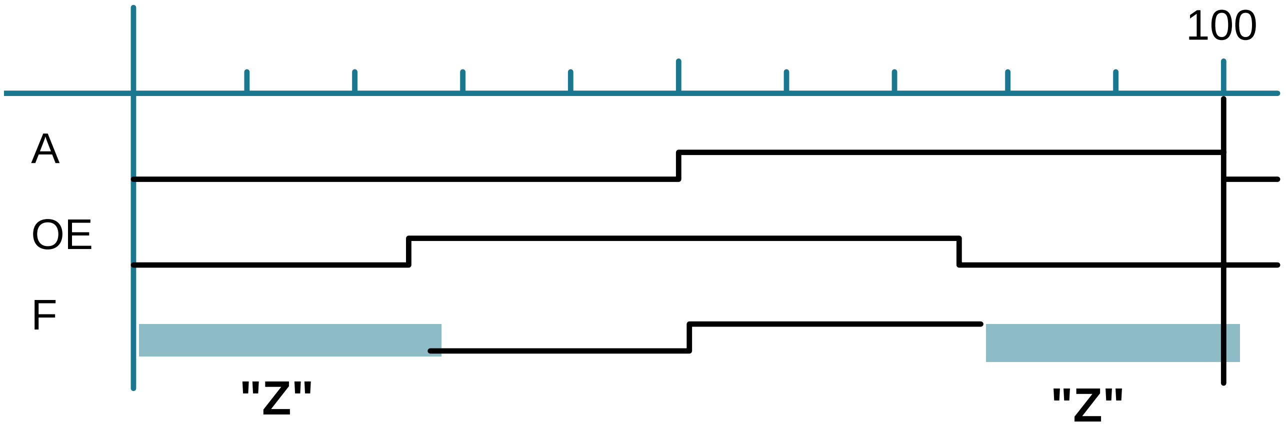
Tri-State Outputs (cont'd)

- Tri-state gates (cont'd)
  - Truth table

A	OE	F
X	0	Z
0	1	0
1	1	1

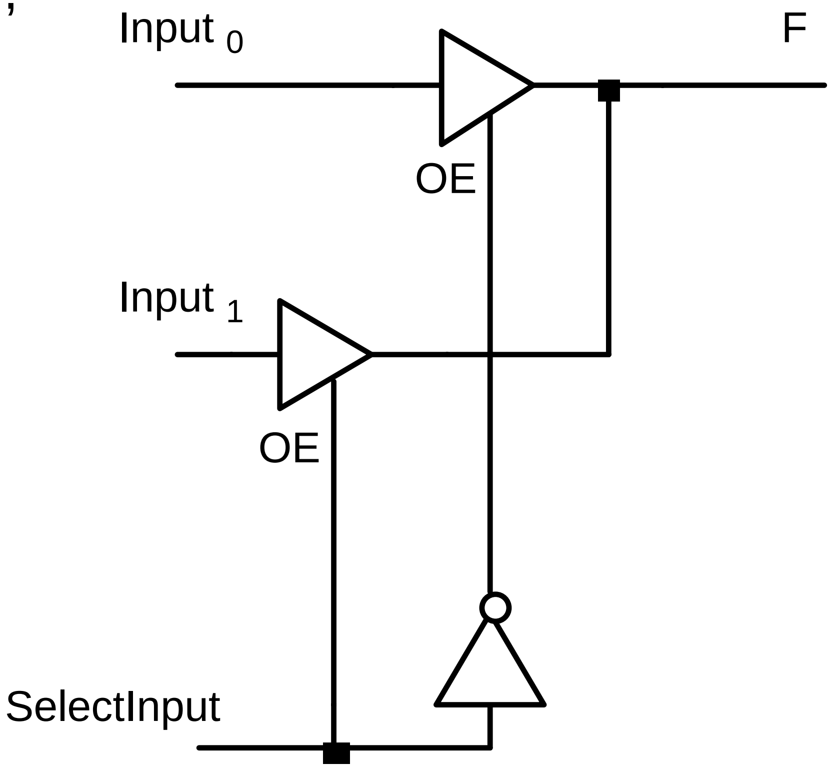


  - Non-inverting buffer's timing waveform



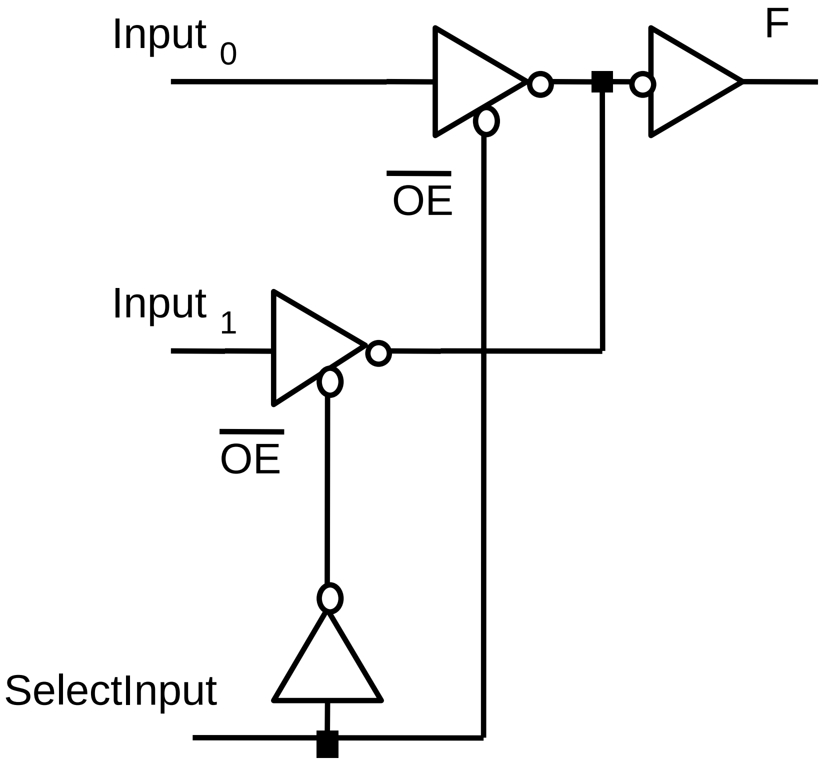
Tri-State Outputs (cont'd)

- Using tri-state gates to implement an economical multiplexer:
  - When SelectInput is asserted high, Input1 is connected to F
  - When SelectInput is driven low, Input0 is connected to F
  - This is essentially a 2:1 Mux



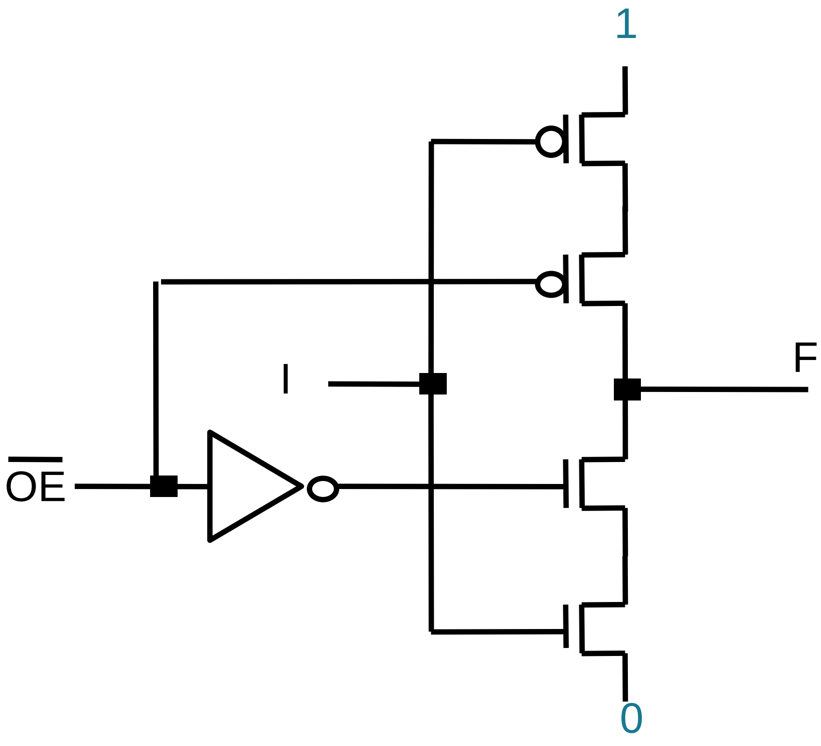
Tri-State Outputs (cont'd)

- Alternative Tri-state Fragment



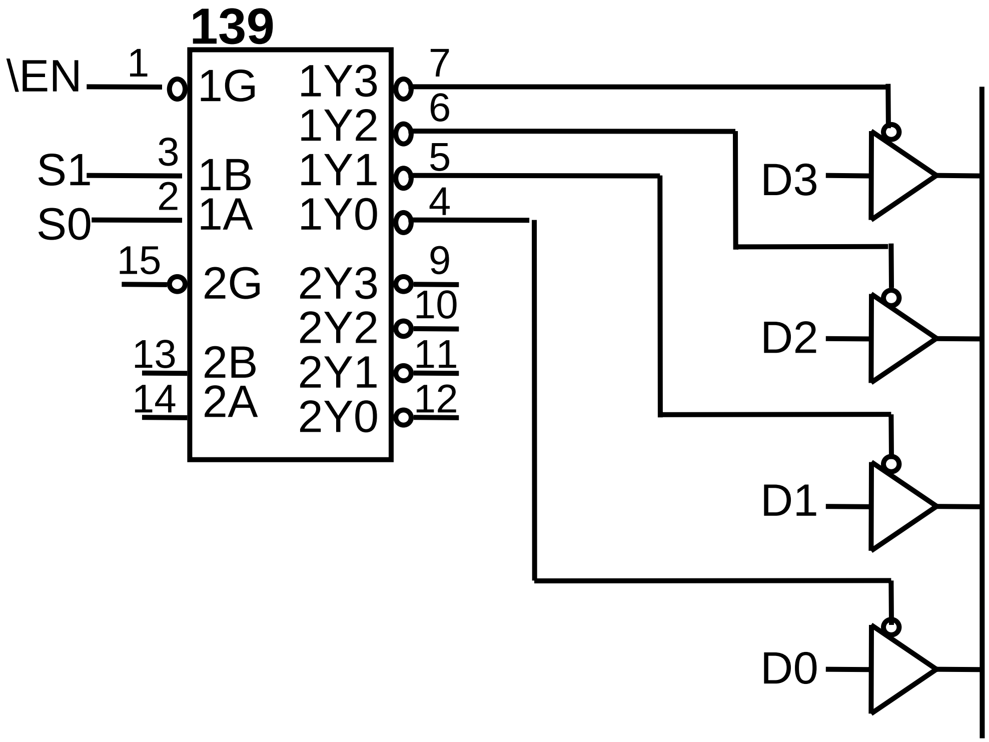
Active low tri-state enables plus inverting tri-state buffers

- Switch Level Implementation of tri-state gate



Tri-State Outputs (cont'd)

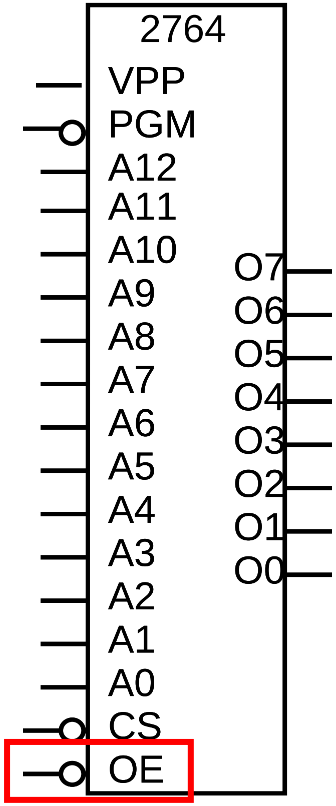
- 4:1 Multiplexer, Revisited



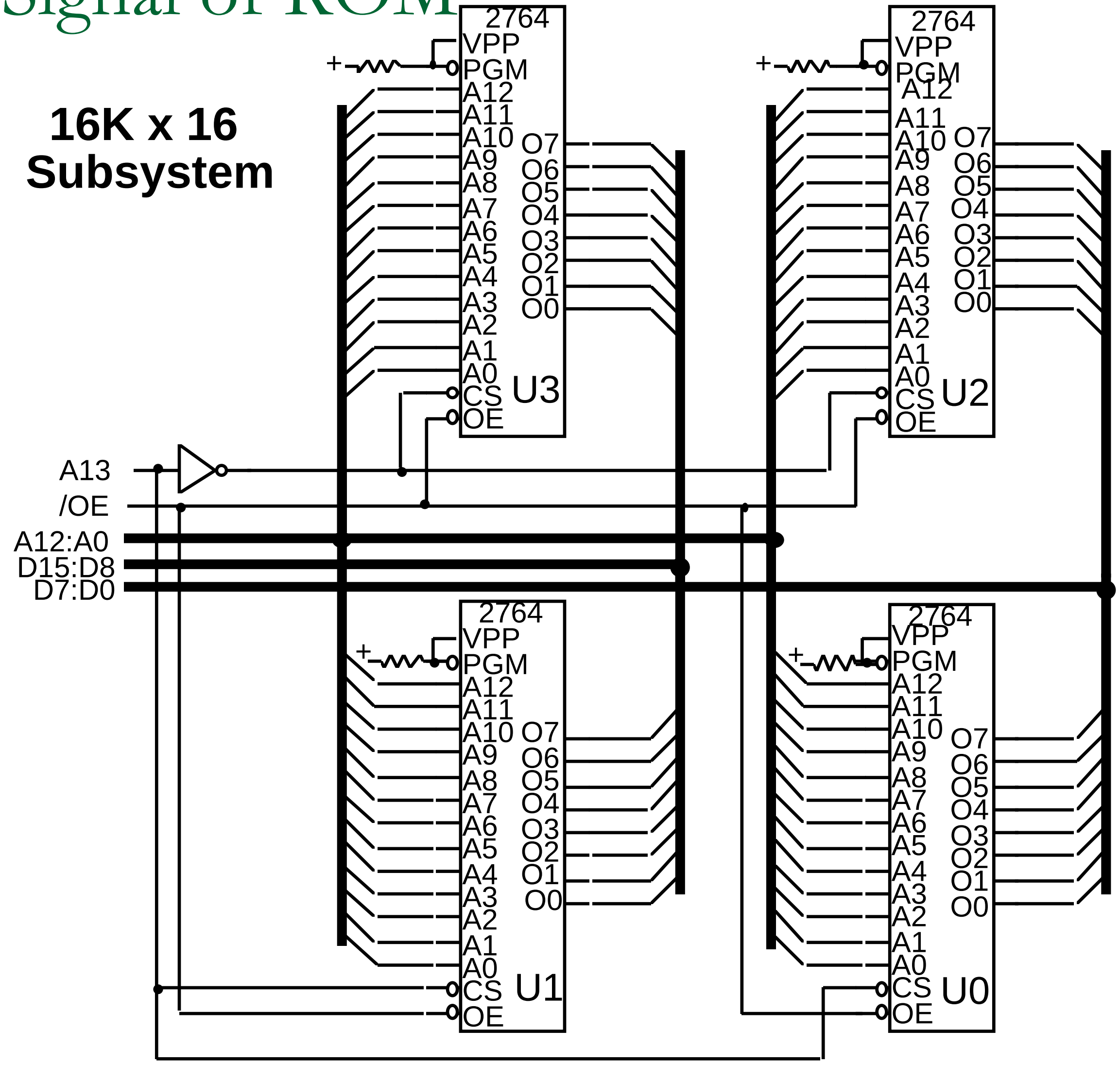
Decoder + 4 tri-state Gates

Tri-State Outputs (cont'd)
- Output Enable Signal of ROM

2764 EPROM
8K x 8



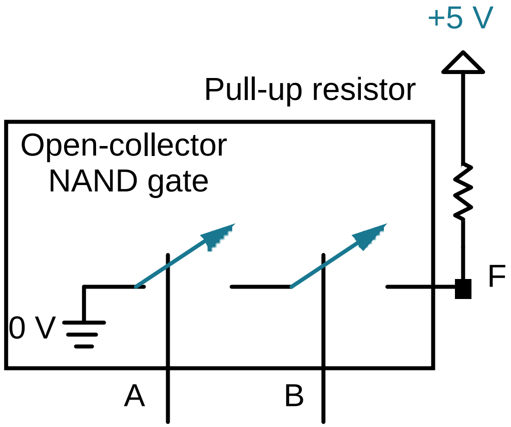
16K x 16
Subsystem



Non-Gate Logic (cont'd)
- Open-Collector Logic

- Open Collector
  - another way to connect multiple gates to the same output wire
  - The gate only has the ability to pull its output low;
    it cannot actively drive the wire high
  - this is done by pulling the wire up to a logic 1 voltage through a resistor

- Switch representation
  of an OC NAND

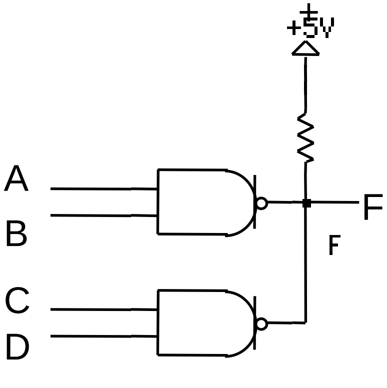


- When A=1 and B=1,
  equivalent circuit of OC-NAND:

(Ra+Rb)
Rr
0 V
+5 V

Open-Collector Logic (cont'd)

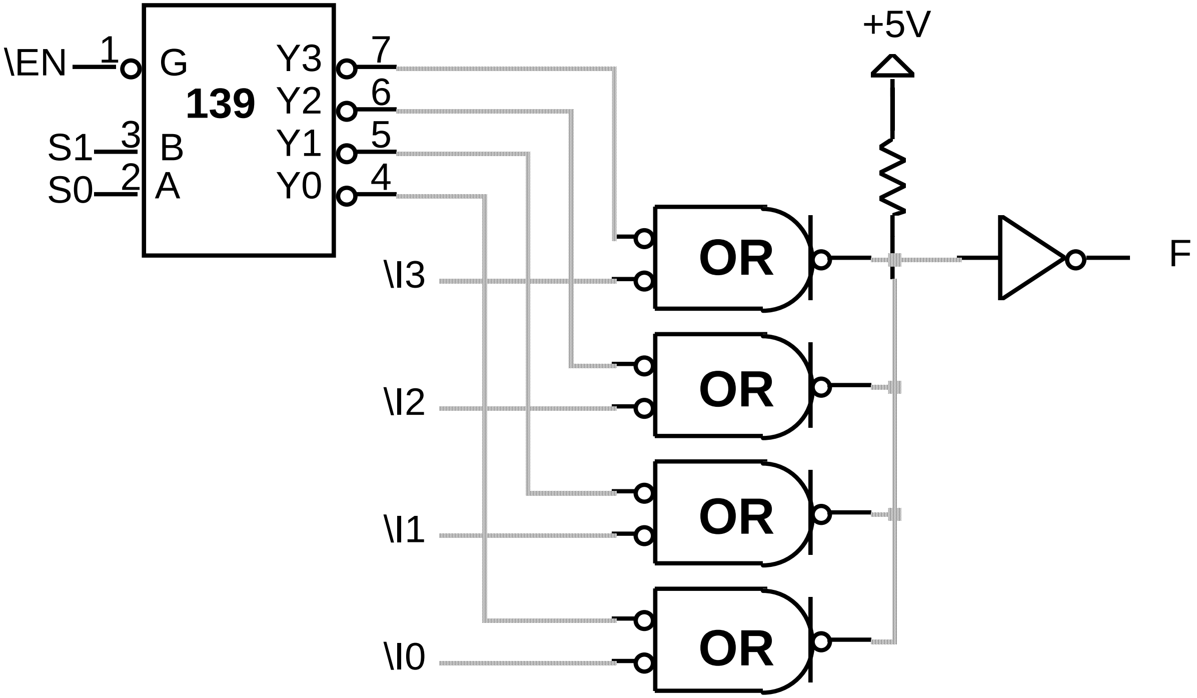
- Example: OC-NANDs in wired-AND configuration



•If A and B are "1", output is actively pulled low
•if C and D are "1", output is actively pulled low
•if one gate is low, the other high, then low wins
•if both gates are "1", the output floats, pulled high by resistor
•Hence, the two NAND functions are AND'd together!

Open-Collector Logic (cont'd)

- 4:1 Multiplexer



Decoder + 4 Open Collector Gates

Summary

- Theme:
  - How to construct digital systems with more complex logic building blocks than discrete gates
- Overview of the different classes of basic components:
  - Fixed logic
  - Look-up table based logic (ROM)
  - Steering logic (Mux, Demux)
  - PLA, PAL
  - FPGA
- Non-gate logic
  - Enables efficient multiplexing of large numbers of signals
  - Tristate
  - Open-collector