

Combinational Logic Technologies

- Standard gates
 - gate packages
 - cell libraries
- Regular logic
 - multiplexers
 - decoders
- Two-level programmable logic
 - PALs
 - PLAs
 - ROMs

Random logic

- Transistors quickly integrated into logic gates (1960s)
- Catalog of common gates (1970s)
 - Texas Instruments Logic Data Book – the yellow bible
 - all common packages listed and characterized (delays, power)
 - typical packages:
 - in 14-pin IC: 6-inverters, 4 NAND gates, 4 XOR gates
- Today, very few parts are still in use
- However, parts libraries exist for chip design
 - designers reuse already characterized logic gates on chips
 - same reasons as before
 - difference is that the parts don't exist in physical inventory – created as needed

Random logic

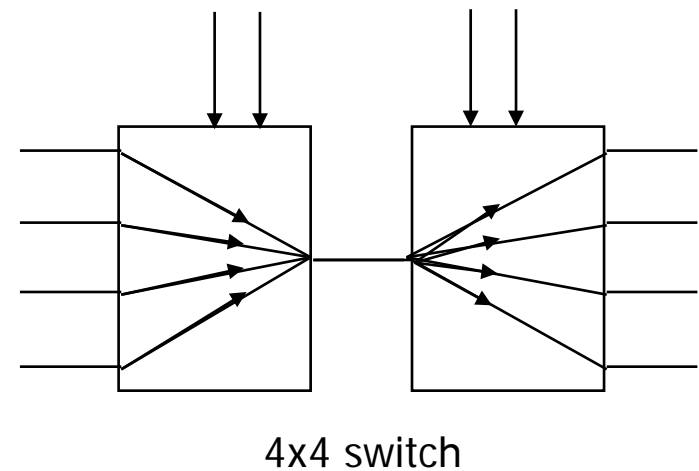
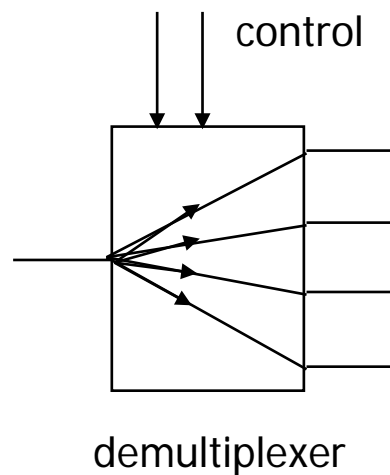
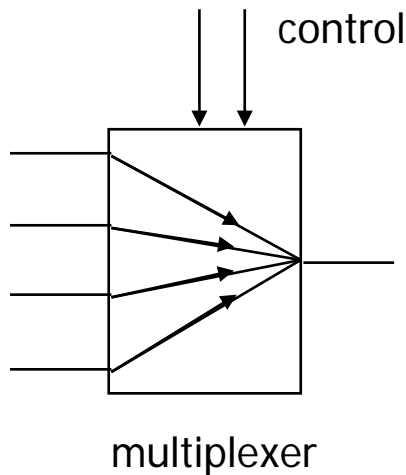
- Too hard to figure out exactly what gates to use
 - map from logic to NAND/NOR networks
 - determine minimum number of packages
 - slight changes to logic function could decrease cost
- Changes to difficult to realize
 - need to rewire parts
 - may need new parts
 - design with spares (few extra inverters and gates on every board)

Regular logic

- Need to make design faster
- Need to make engineering changes easier to make
- Simpler for designers to understand and map to functionality
 - harder to think in terms of specific gates
 - better to think in terms of a large multi-purpose block

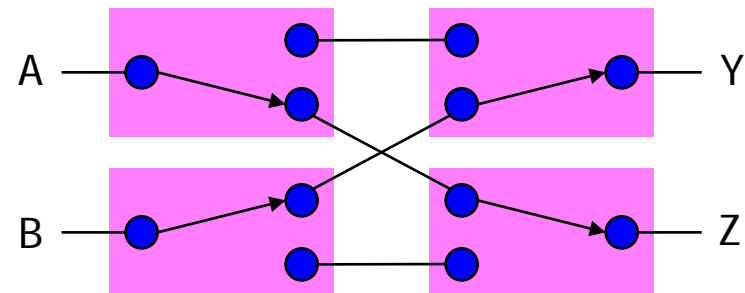
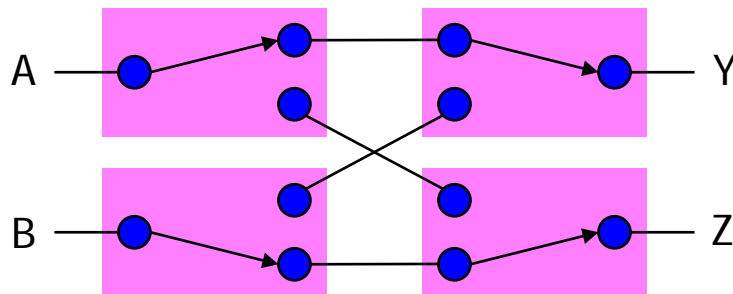
Making connections

- Direct point-to-point connections between gates
 - wires we've seen so far
- Route one of many inputs to a single output --- multiplexer
- Route a single input to one of many outputs --- demultiplexer



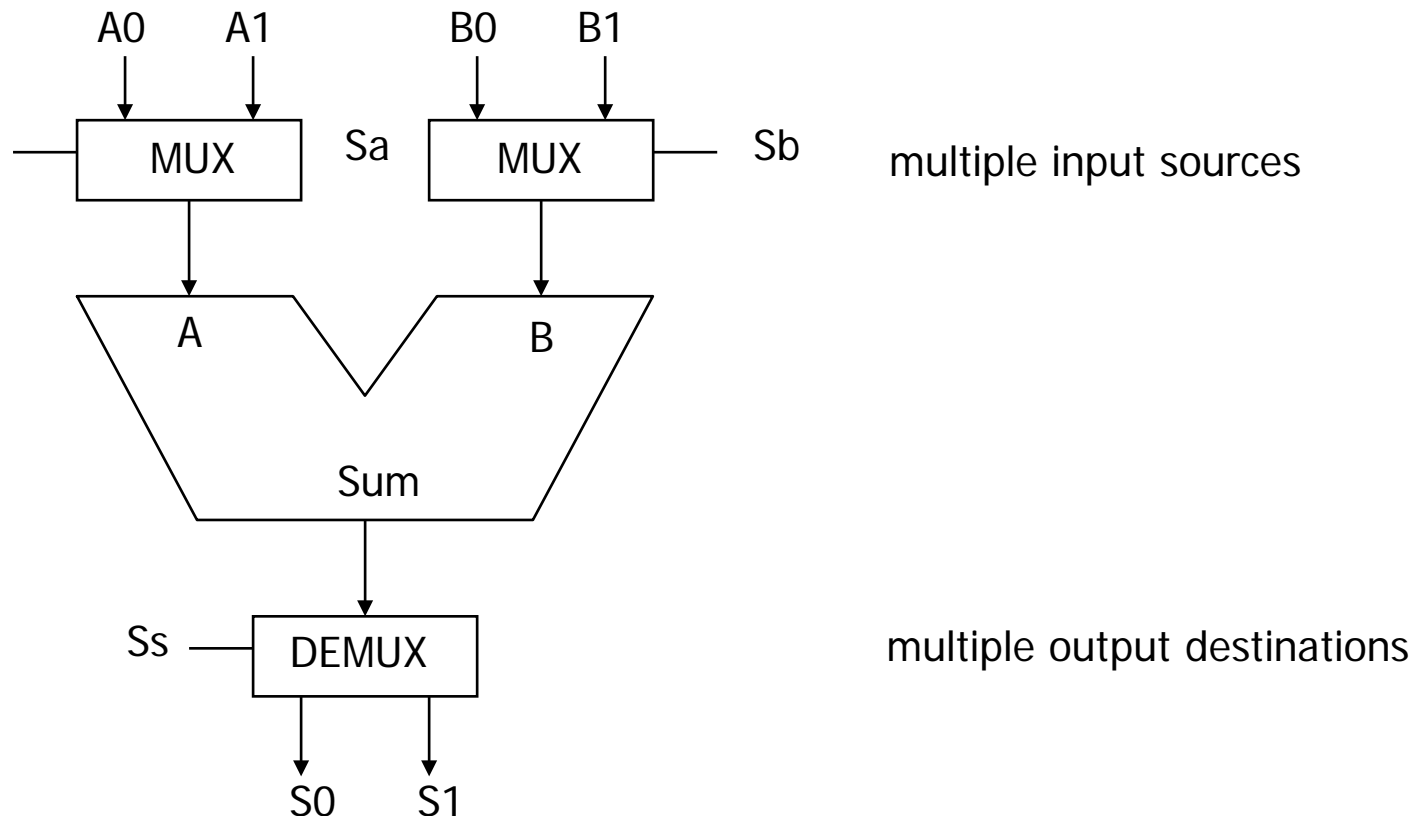
Mux and demux

- 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



Multiplexers/selectors

- Multiplexers/selectors: general concept
 - 2^n data inputs, n control inputs (called "selects"), 1 output
 - used to connect 2^n points to a single point
 - control signal pattern forms binary index of input connected to output

$$Z = A' I_0 + A I_1$$

A	Z
0	I_0
1	I_1

functional form

logical form

two alternative forms
for a 2:1 Mux truth table

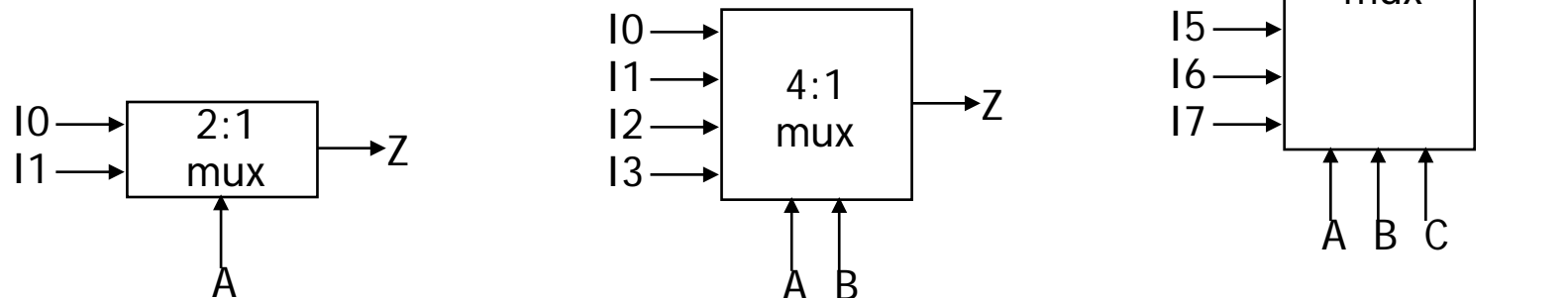
I_1	I_0	A	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Multiplexers/selectors (cont'd)

- 2:1 mux: $Z = A'I_0 + AI_1$
- 4:1 mux: $Z = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$
- 8:1 mux: $Z = A'B'C'I_0 + A'B'CI_1 + A'BC'I_2 + A'BCI_3 + AB'C'I_4 + AB'CI_5 + ABC'I_6 + ABCI_7$

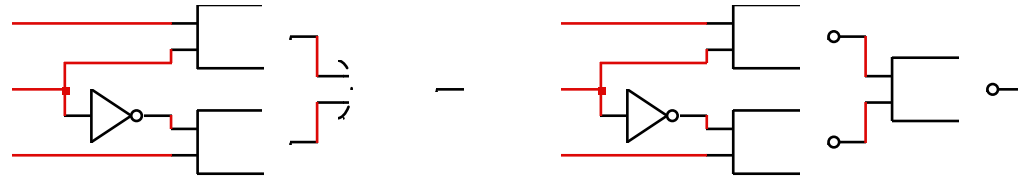
- In general: $Z = \sum_{k=0}^{2^n-1} (m_k I_k)$

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

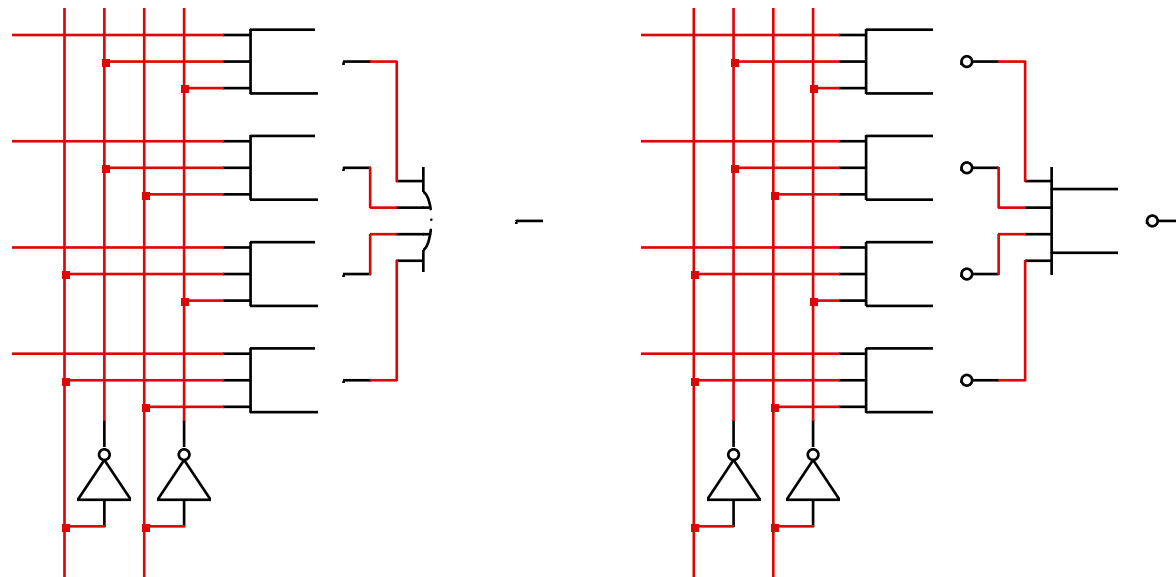


Gate level implementation of muxes

■ 2:1 mux

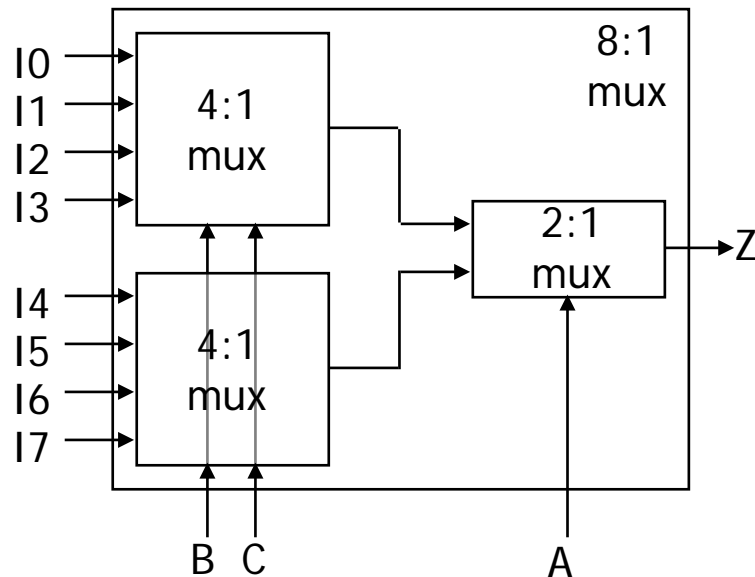


■ 4:1 mux



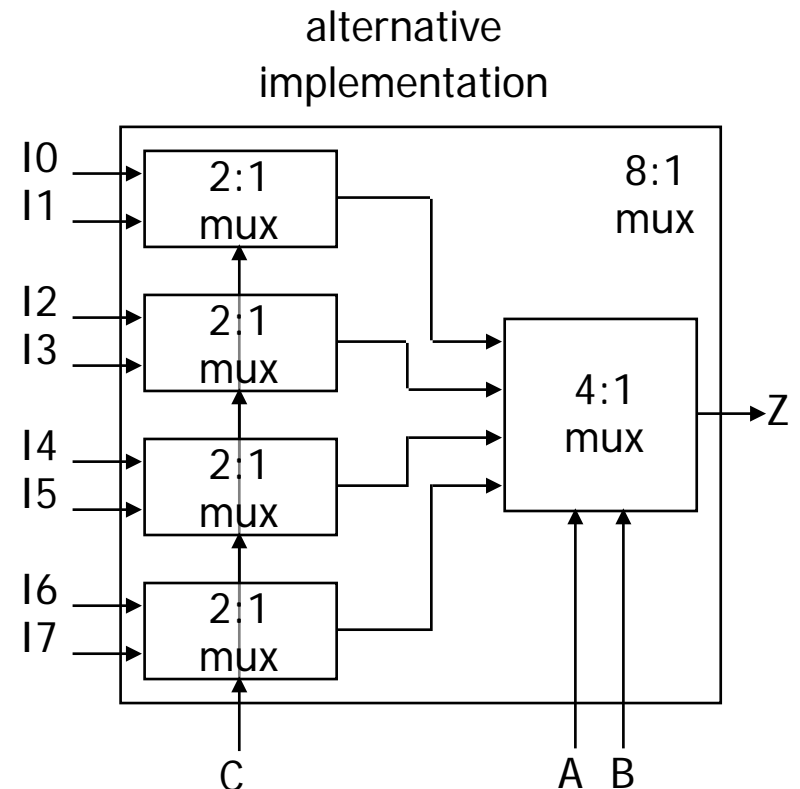
Cascading multiplexers

- 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

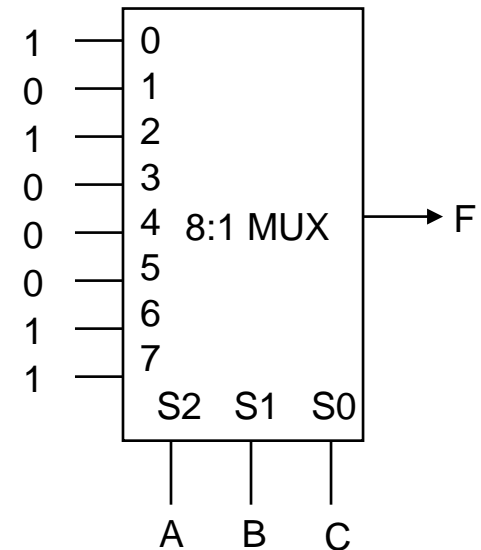


Multiplexers as general-purpose logic

- 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:

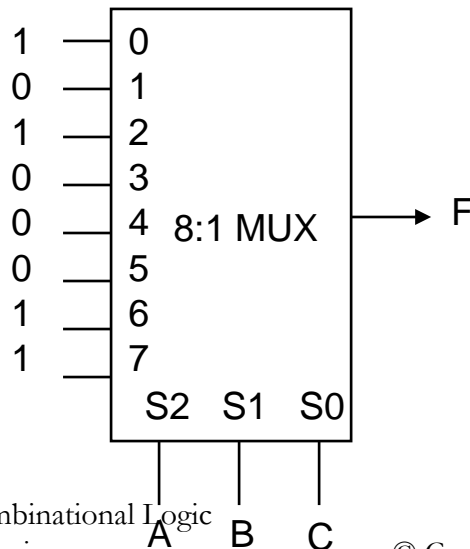
- $$\begin{aligned} F(A,B,C) &= m_0 + m_2 + m_6 + m_7 \\ &= A'B'C' + A'BC' + ABC' + ABC \\ &= A'B'C'(1) + A'B'C(0) \\ &\quad + A'BC'(1) + A'BC(0) \\ &\quad + AB'C'(0) + AB'C(0) \\ &\quad + ABC'(1) + ABC(1) \end{aligned}$$



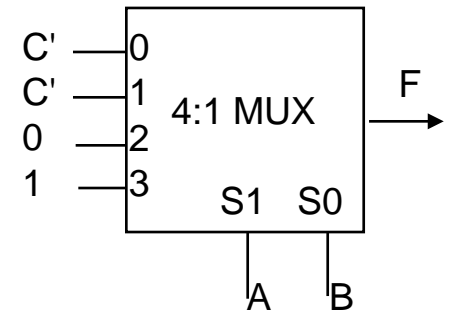
$$\begin{aligned} Z = & A'B'C'I_0 + A'B'CI_1 + A'BC'I_2 + A'BCI_3 + \\ & AB'C'I_4 + AB'CI_5 + ABC'I_6 + ABCI_7 \end{aligned}$$

Multiplexers as general-purpose logic (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:
 - $F(A,B,C) = m_0 + m_2 + m_6 + m_7$
 $= A'B'C' + A'BC' + ABC' + ABC$
 $= A'B'(C') + A'B(C') + AB'(0) + AB(1)$



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

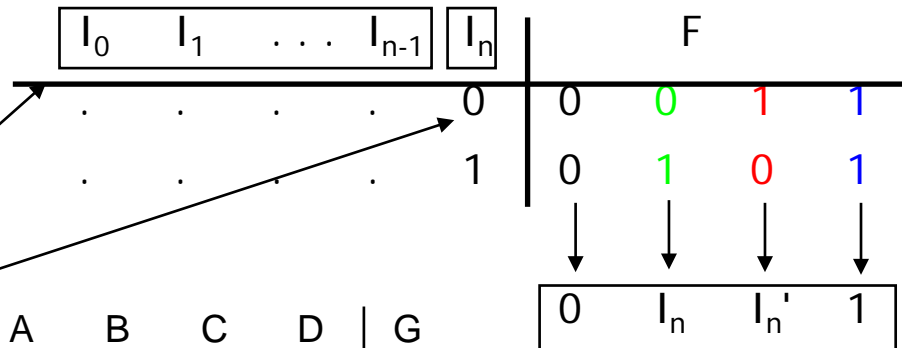


Multiplexers as general-purpose logic (cont'd)

Generalization

n-1 mux control variables

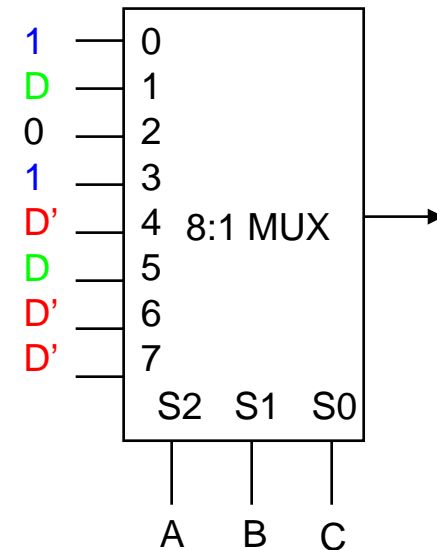
single mux data variable



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

choose A,B,C as
control variables

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



Activity

- Realize $F = B'CD' + ABC'$ with a 4:1 multiplexer and a minimum of other gates:

Demultiplexers/decoders

- Decoders/demultiplexers: general concept
 - 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)

1:2 Decoder:

$$O0 = G \bullet S'$$

$$O1 = G \bullet S$$

2:4 Decoder:

$$O0 = G \bullet S1' \bullet S0'$$

$$O1 = G \bullet S1' \bullet S0$$

$$O2 = G \bullet S1 \bullet S0'$$

$$O3 = G \bullet S1 \bullet S0$$

3:8 Decoder:

$$O0 = G \bullet S2' \bullet S1' \bullet S0'$$

$$O1 = G \bullet S2' \bullet S1' \bullet S0$$

$$O2 = G \bullet S2' \bullet S1 \bullet S0'$$

$$O3 = G \bullet S2' \bullet S1 \bullet S0$$

$$O4 = G \bullet S2 \bullet S1' \bullet S0'$$

$$O5 = G \bullet S2 \bullet S1' \bullet S0$$

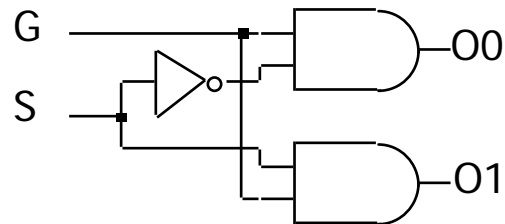
$$O6 = G \bullet S2 \bullet S1 \bullet S0'$$

$$O7 = G \bullet S2 \bullet S1 \bullet S0$$

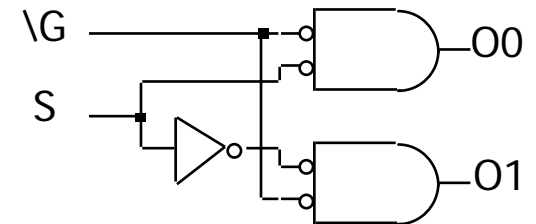
Gate level implementation of demultiplexers

■ 1:2 decoders

active-high
enable

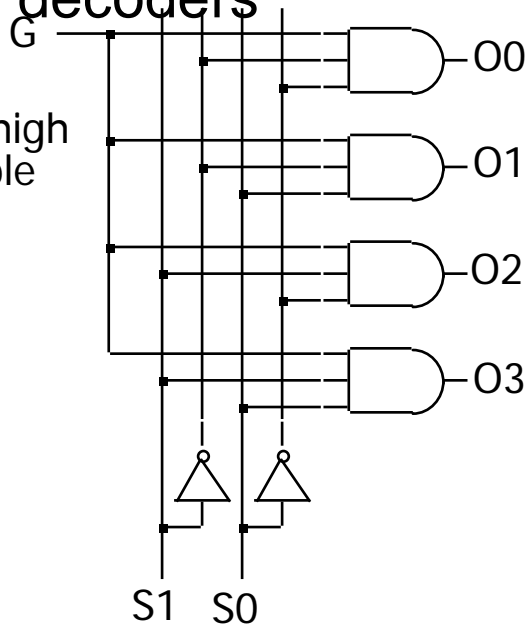


active-low
enable

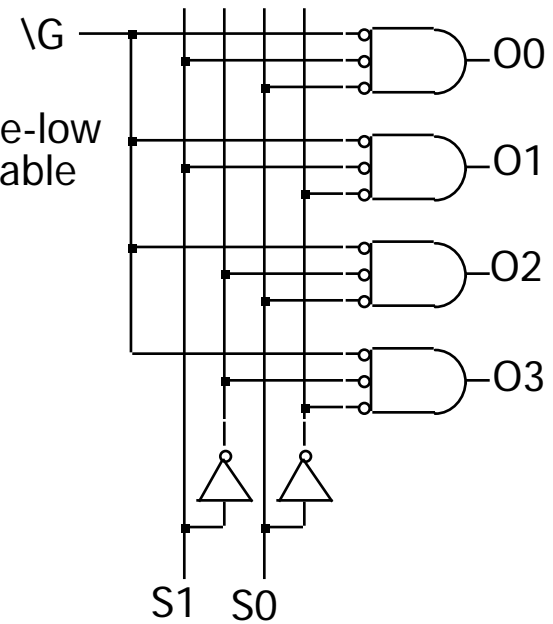


■ 2:4 decoders

active-high
enable

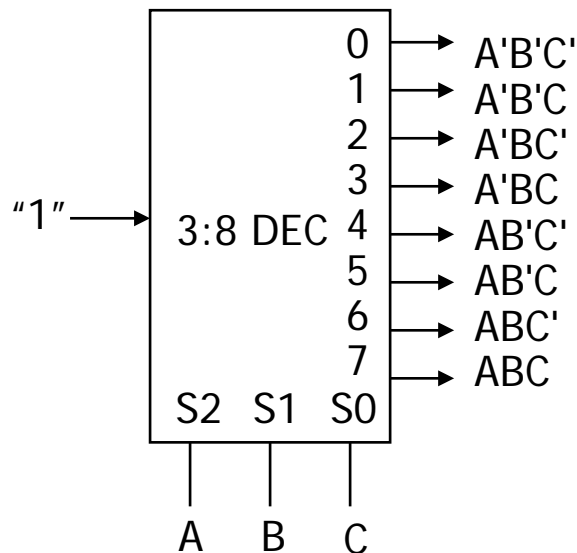


active-low
enable



Demultiplexers as general-purpose logic

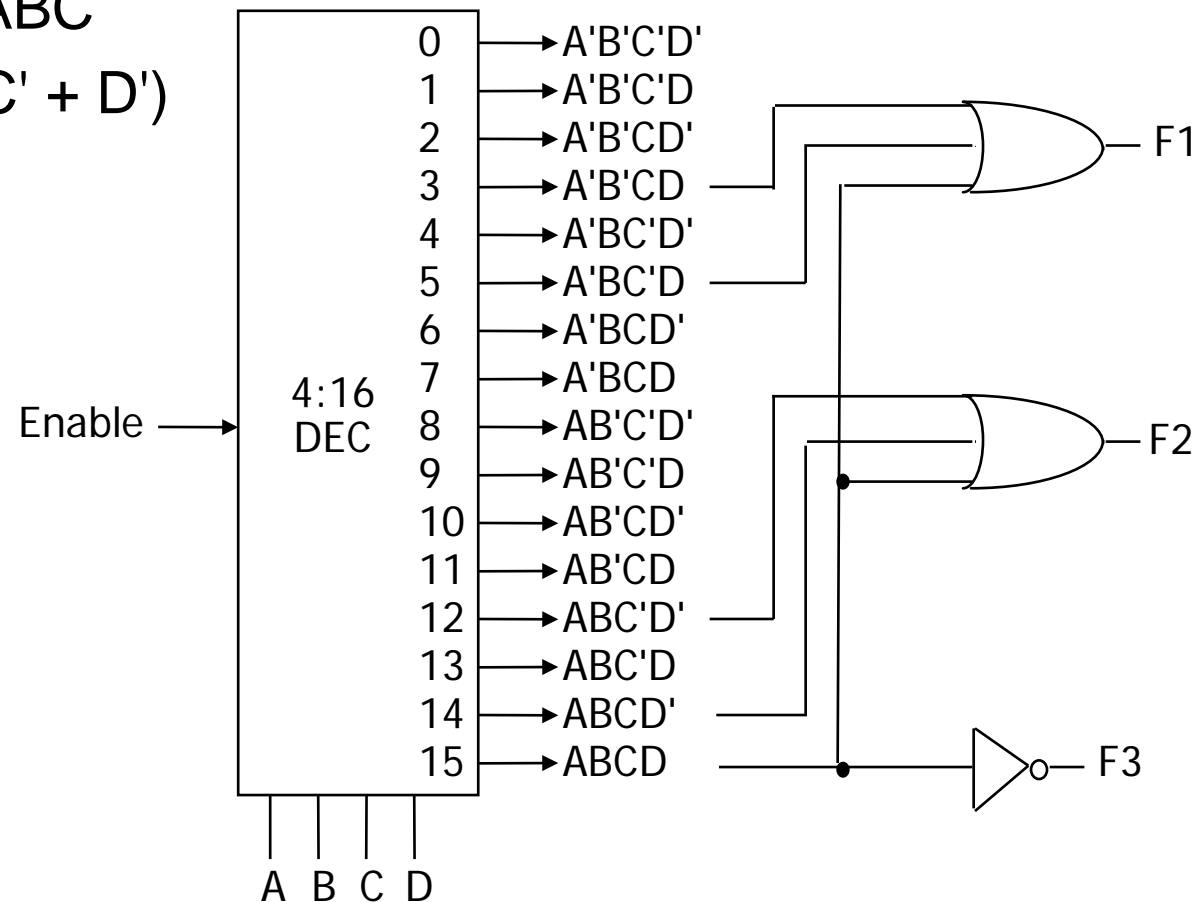
- A $n:2^n$ decoder can implement any function of n variables
 - with the variables used as control inputs
 - the enable inputs tied to 1 and
 - the appropriate minterms summed to form the function



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

Demultiplexers as general-purpose logic (cont'd)

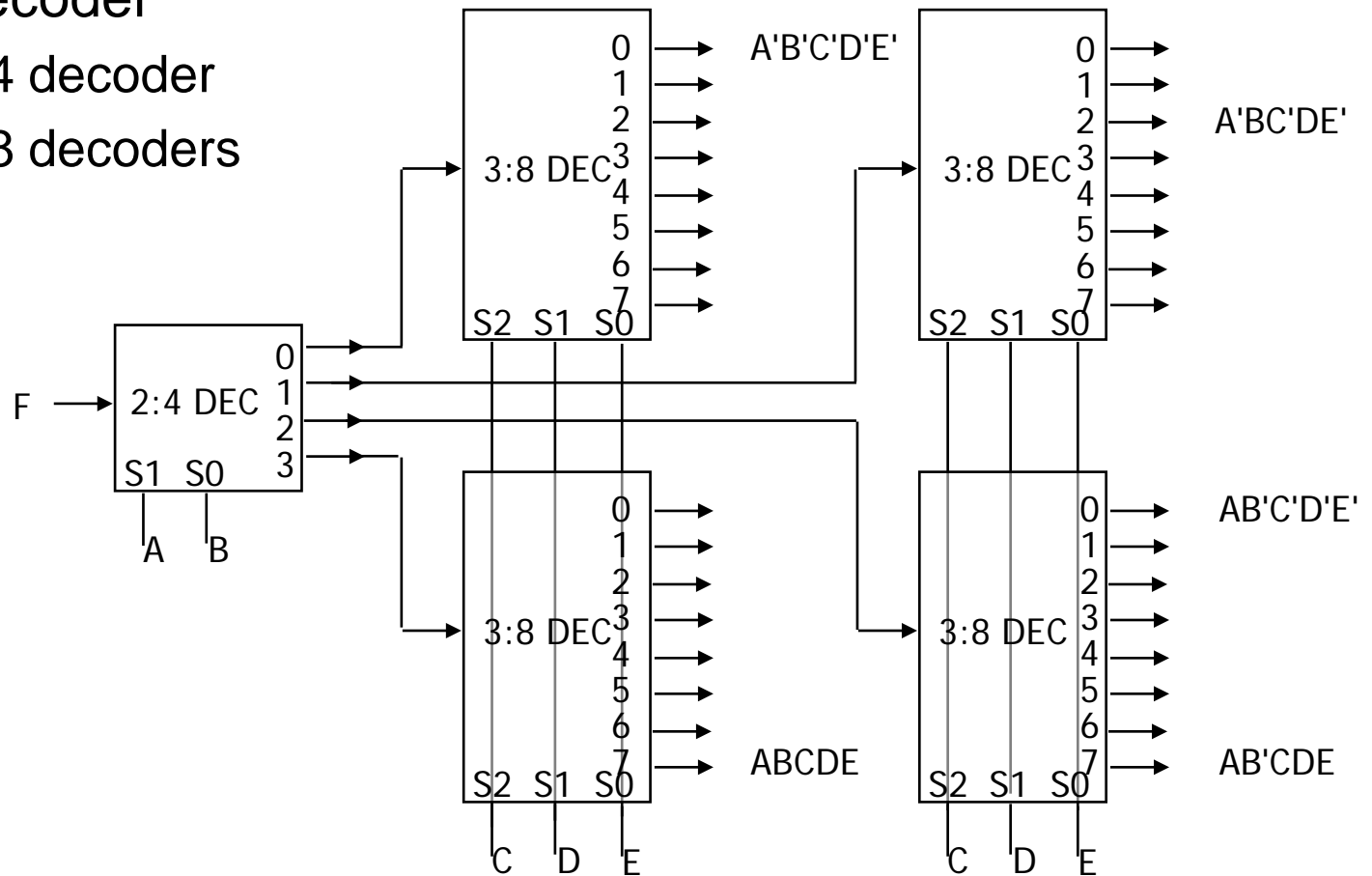
- $F1 = A'BC'D + A'B'CD + ABCD$
- $F2 = ABC'D' + ABC$
- $F3 = (A' + B' + C' + D')$



Cascading decoders

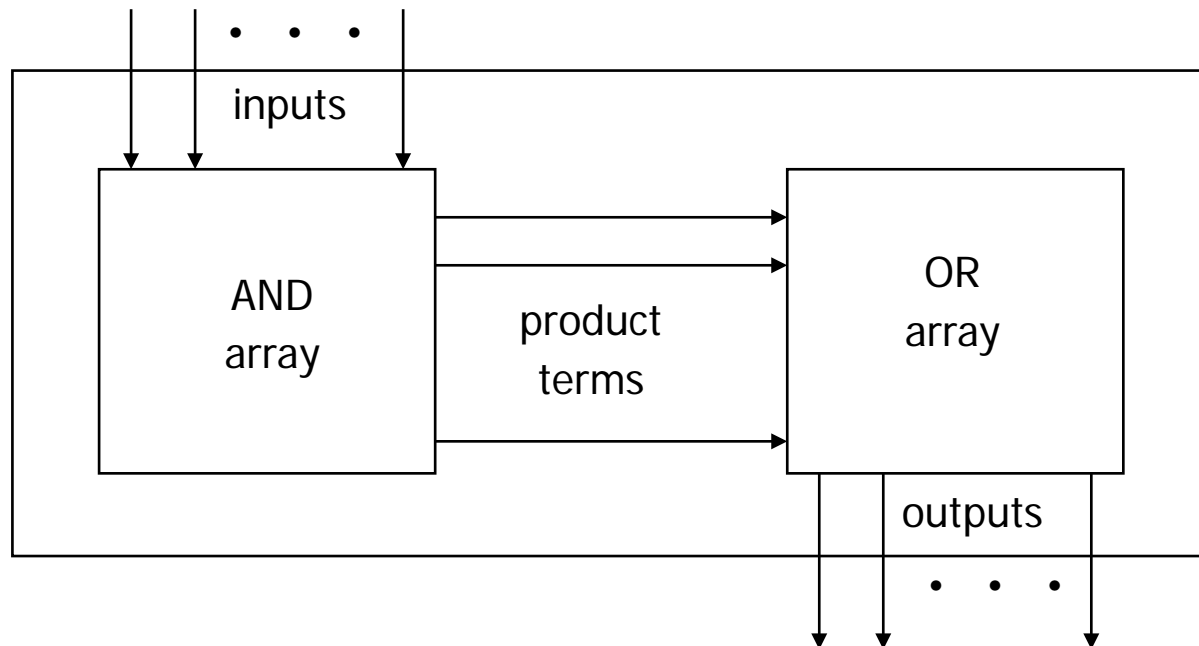
■ 5:32 decoder

- 1x2:4 decoder
- 4x3:8 decoders



Programmable logic arrays

- 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



Enabling concept

■ Shared product terms among outputs

example:

$$\begin{aligned}F0 &= A + B' C' \\F1 &= A C' + A B \\F2 &= B' C' + A B \\F3 &= B' C + A\end{aligned}$$

personality matrix

product term	inputs			outputs			
	A	B	C	F0	F1	F2	F3
AB	1	1	–	0	1	1	0
B'C	–	0	1	0	0	0	1
AC'	1	–	0	0	1	0	0
B'C'	–	0	0	1	0	1	0
A	1	–	–	1	0	0	1

input side:

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

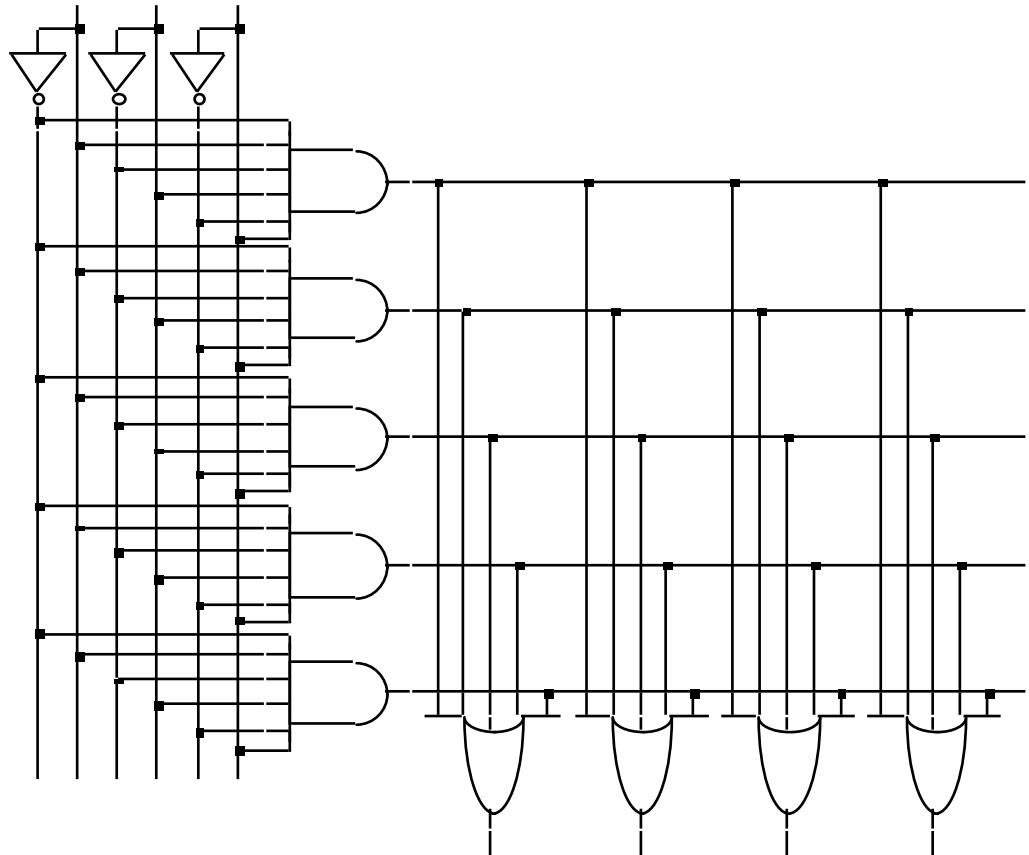
output side:

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

reuse of terms

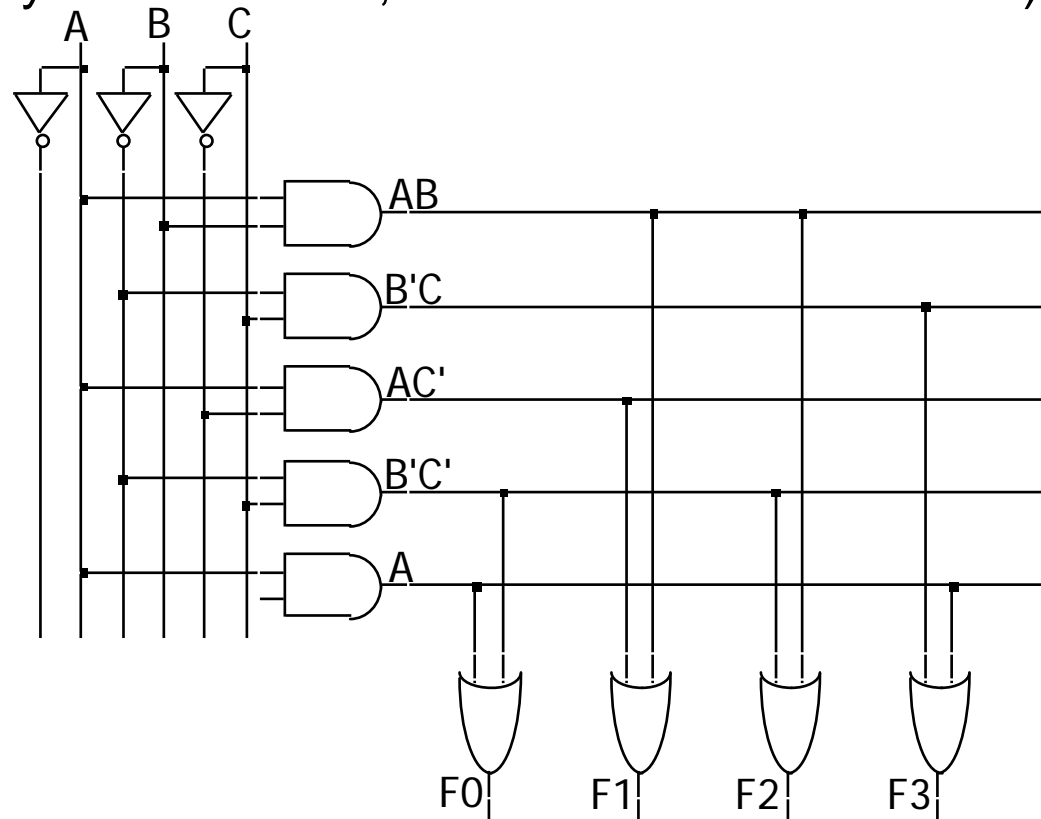
Before programming

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



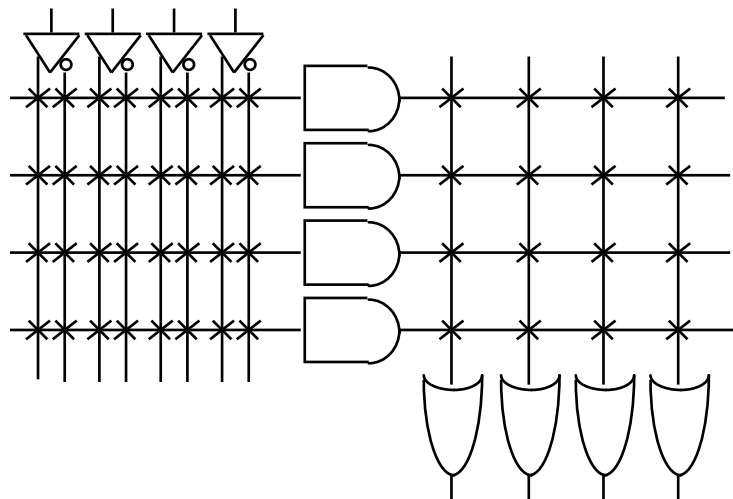
After programming

- Unwanted connections are "blown"
 - fuse (normally connected, break unwanted ones)
 - anti-fuse (normally disconnected, make wanted connections)



Alternate representation for high fan-in structures

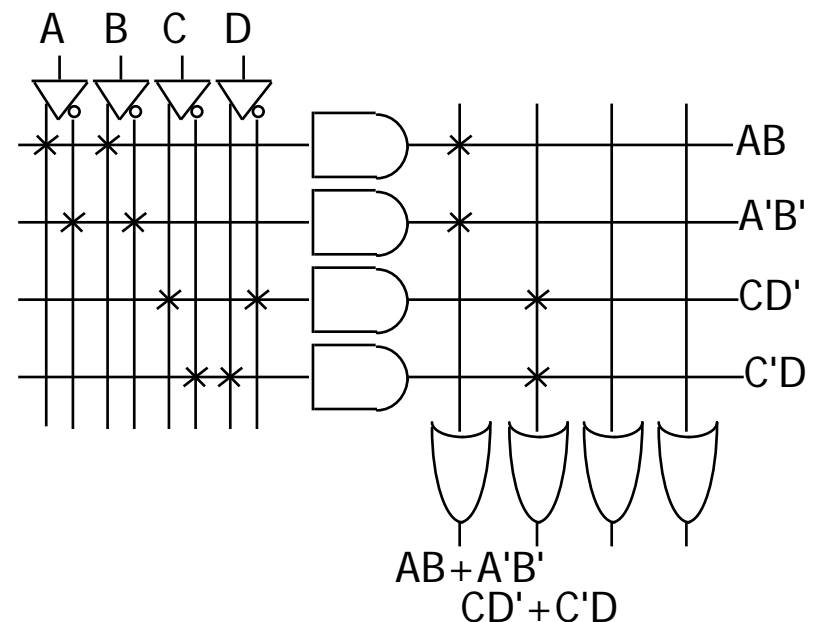
- Short-hand notation so we don't have to draw all the wires
 - × signifies a connection is present and perpendicular signal is an input to gate



notation for implementing

$$F0 = A B + A' B'$$

$$F1 = C D' + C' D$$

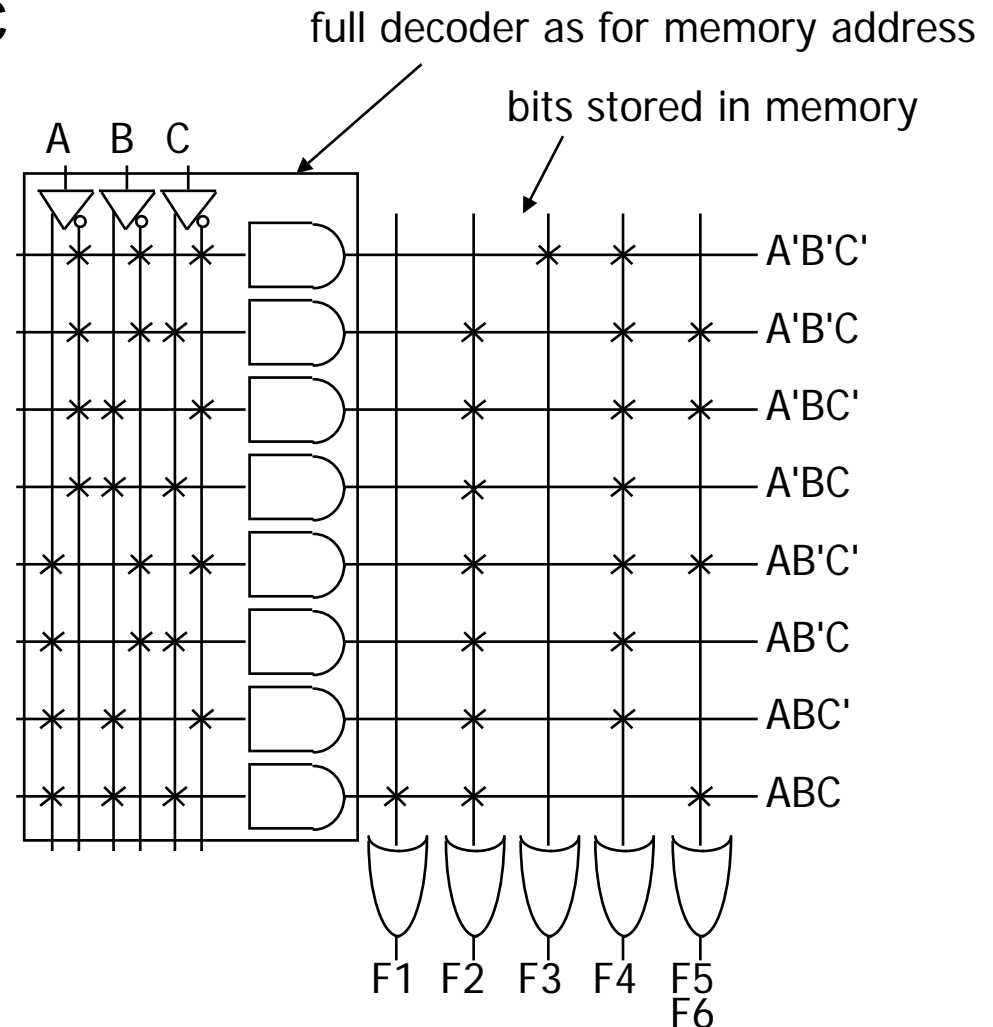


Programmable logic array example

■ Multiple functions of A, B, C

- $F1 = A B C$
- $F2 = A + B + C$
- $F3 = A' B' C'$
- $F4 = A' + B' + C'$
- $F5 = A \text{ xor } B \text{ xor } C$
- $F6 = A \text{ xnor } B \text{ xnor } C$

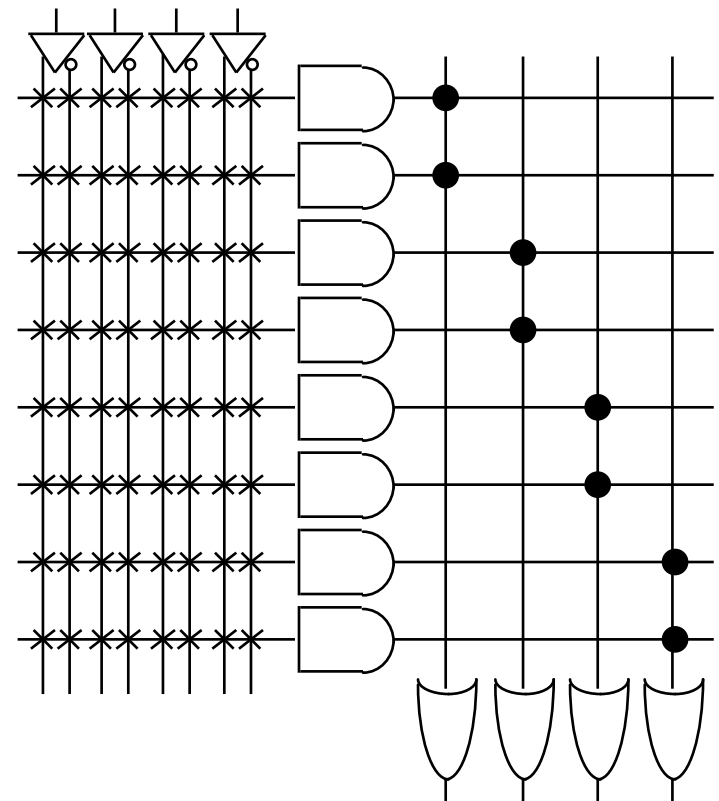
A	B	C	F1	F2	F3	F4	F5	F6
0	0	0	0	0	1	1	0	0
0	0	1	0	1	0	1	1	1
0	1	0	0	1	0	1	1	1
0	1	1	0	1	0	1	0	0
1	0	0	0	1	0	1	1	1
1	0	1	0	1	0	1	0	0
1	1	0	0	1	0	1	0	0
1	1	1	1	1	0	0	1	1



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: 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:

$$W = A + BD + BC$$

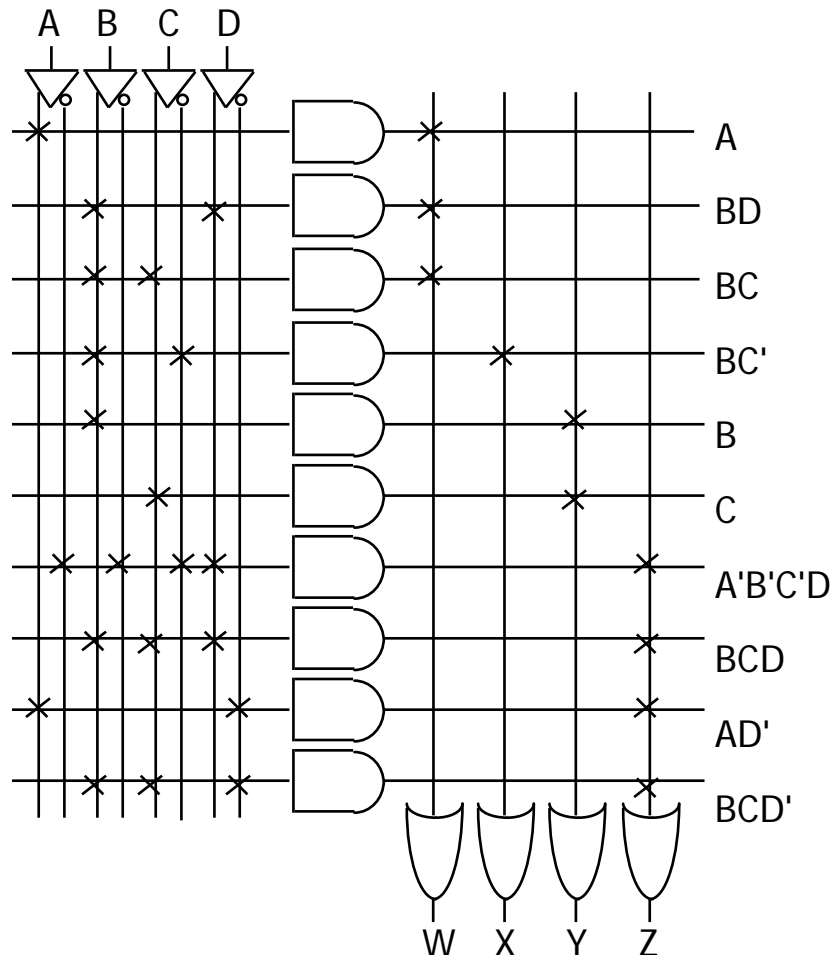
$$X = BC'$$

$$Y = B + C$$

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

PALs and PLAs: design example (cont'd)

■ Code converter: programmed PLA



minimized functions:

$$W = A + BD + BC$$

$$X = B C'$$

$$Y = B + C$$

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

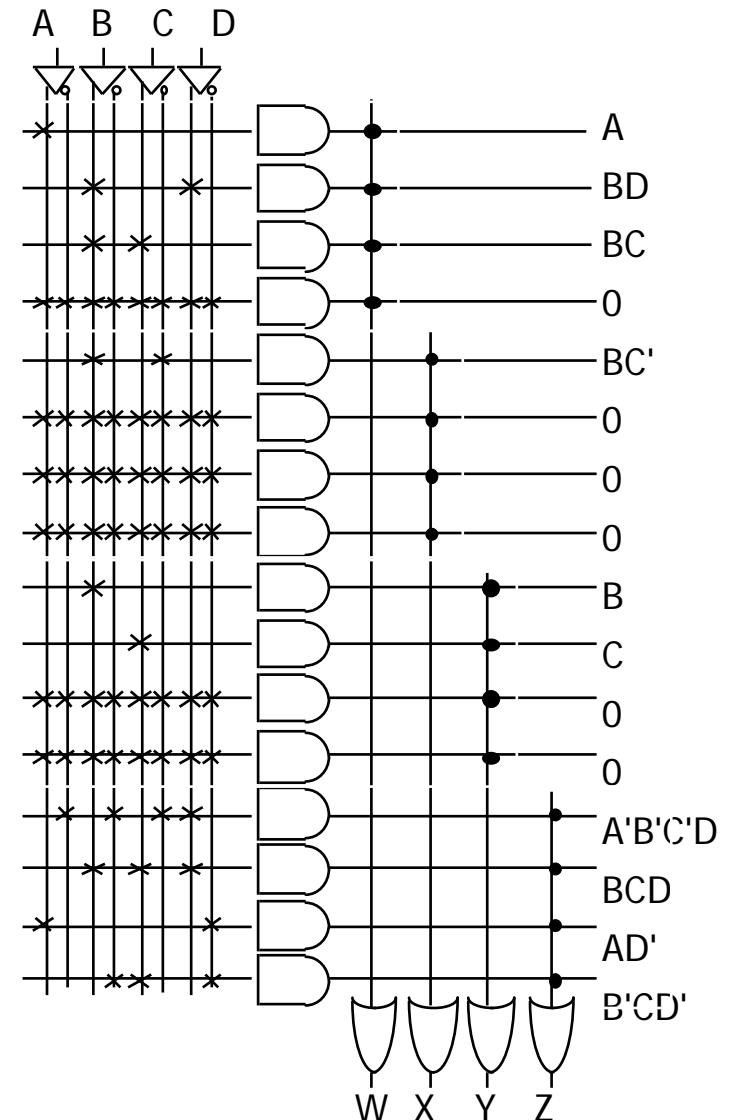
not a particularly good
candidate for PAL/PLA
implementation since no terms
are shared among outputs

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

PALs and PLAs: design example (cont'd)

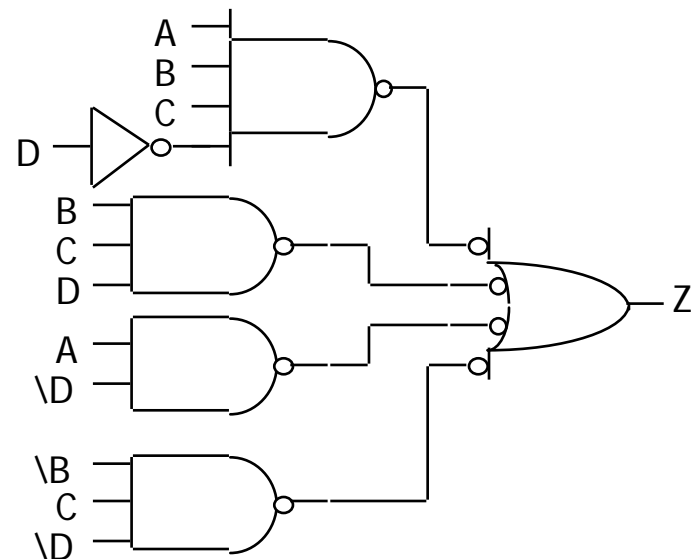
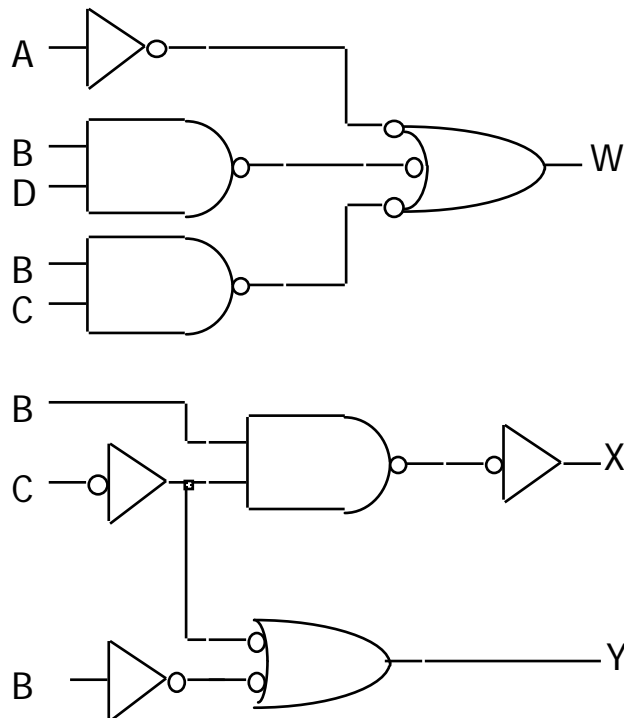
■ Code converter: programmed PAL

4 product terms
per each OR gate



PALs and PLAs: design example (cont'd)

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



PALs and PLAs: another design example

■ Magnitude comparator

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

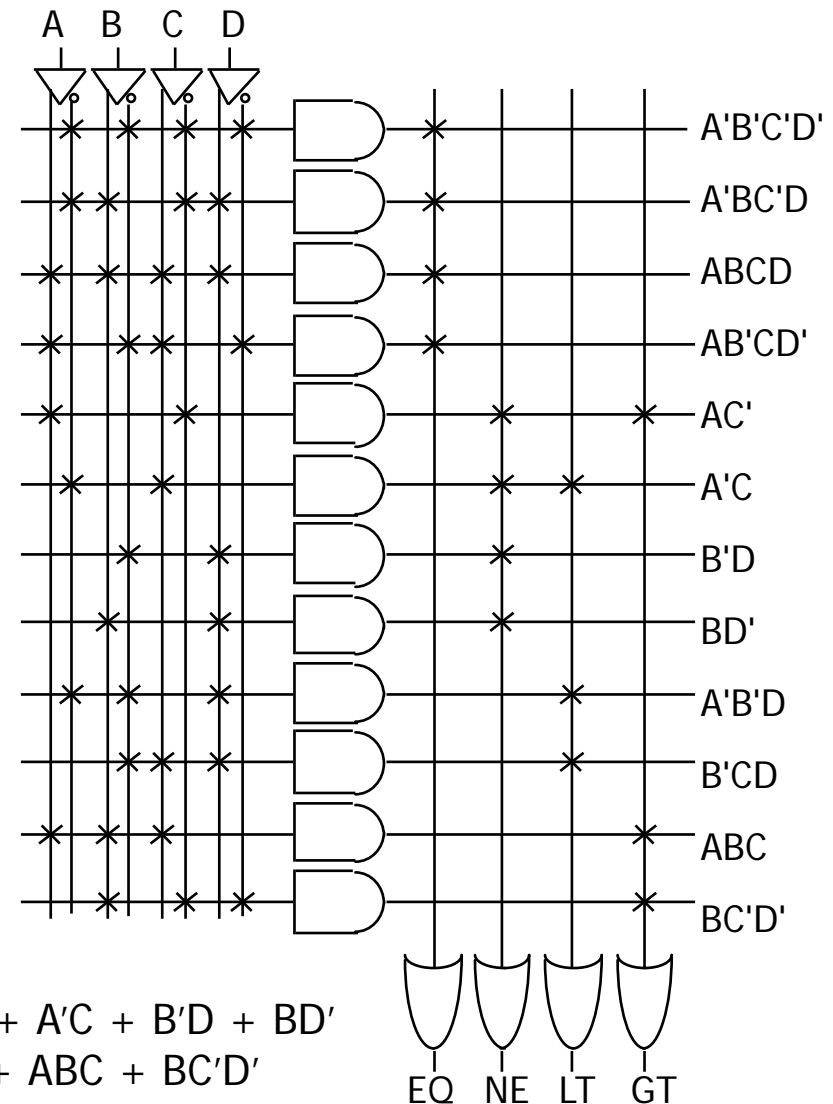
minimized functions:

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

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

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

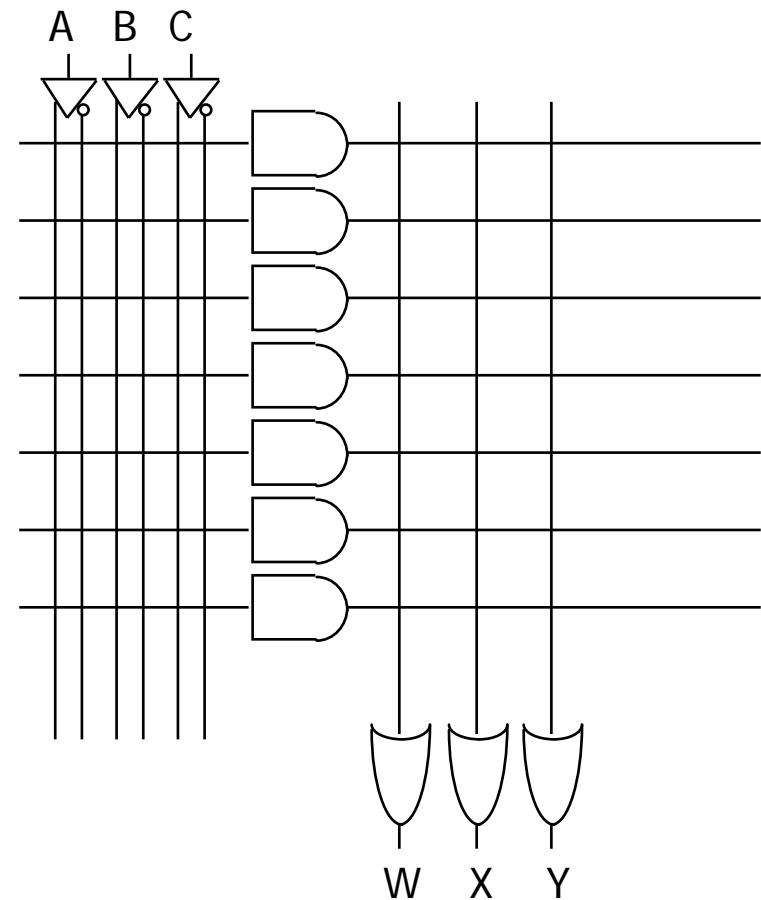
$$GT = AC' + ABC + BC'D'$$



Activity

■ Map the following functions to the PLA below:

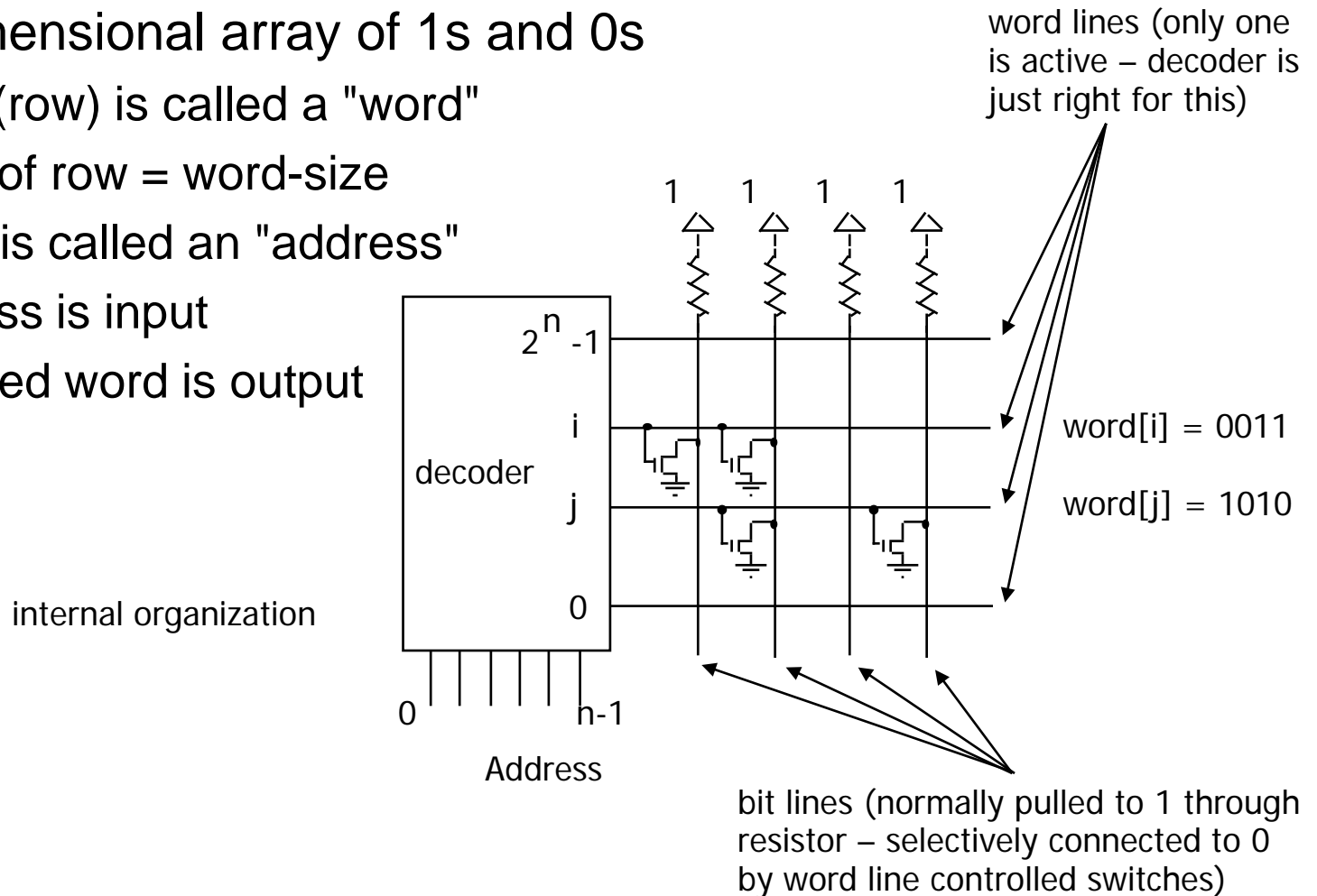
- $W = AB + A'C' + BC'$
- $X = ABC + AB' + A'B$
- $Y = ABC' + BC + B'C'$



Activity (cont'd)

Read-only memories

- 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



ROMs and combinational logic

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

$$F0 = A' B' C + A B' C' + A B' C$$

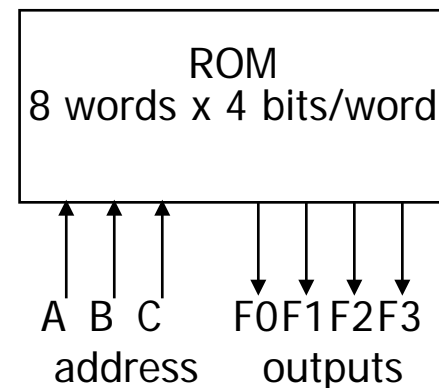
$$F1 = A' B' C + A' B C' + A B C$$

$$F2 = A' B' C' + A' B' C + A B' C'$$

$$F3 = A' B C + A B' C' + A B C'$$

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

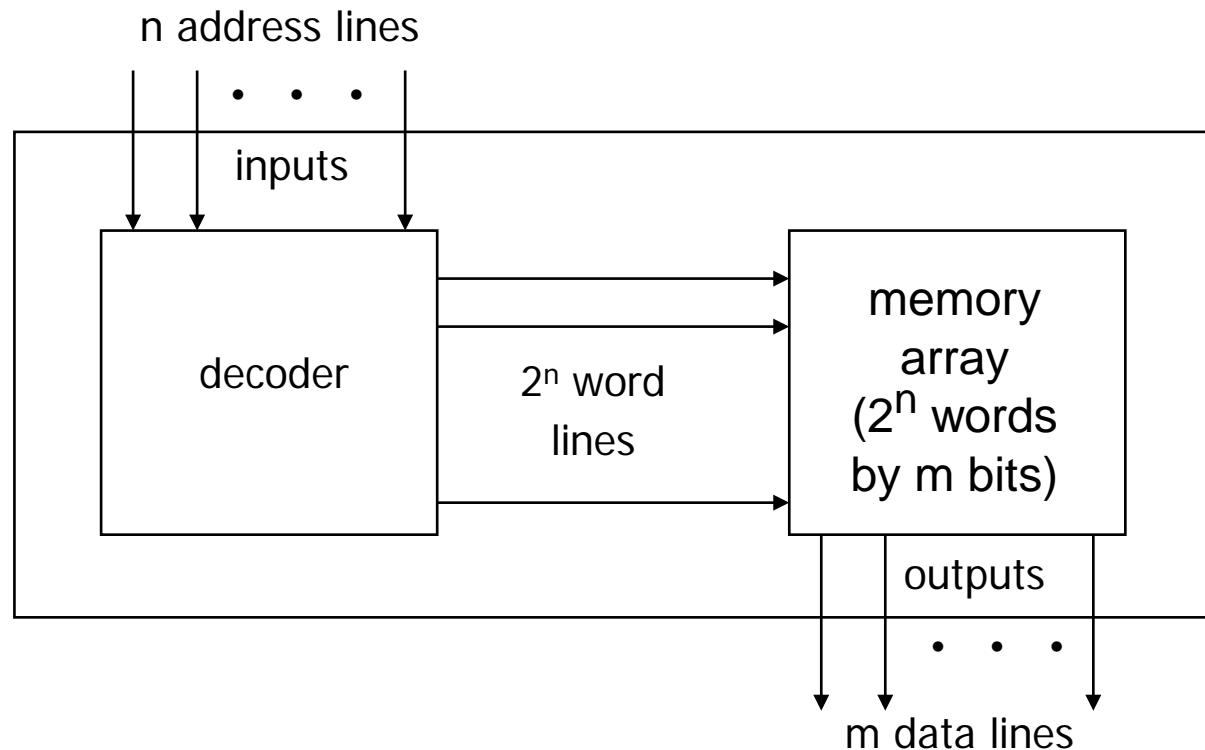
truth table



block diagram

ROM structure

- Similar to a PLA structure but with a fully decoded AND array
 - completely flexible OR array (unlike PAL)



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

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 multi-level 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
 - ❑ in 467 you'll learn about field programmable gate arrays (FPGAs) that are 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)
- Use multiple levels of PALs/PLAs/ROMs
 - ❑ output intermediate result
 - ❑ make it an input to be used in further logic

Combinational logic technology summary

- Random logic
 - ❑ Single gates or in groups
 - ❑ conversion to NAND-NAND and NOR-NOR networks
 - ❑ transition from simple gates to more complex gate building blocks
 - ❑ reduced gate count, fan-ins, potentially faster
 - ❑ more levels, harder to design
- Time response in combinational networks
 - ❑ gate delays and timing waveforms
 - ❑ hazards/glitches (what they are and why they happen)
- Regular logic
 - ❑ multiplexers/decoders
 - ❑ ROMs
 - ❑ PLAs/PALs
 - ❑ advantages/disadvantages of each