
Digital Logic Design

ECEN 3233

Module 4: Combinational Logic Technologies

Louis G. Johnson

School of Electrical and Computer Engineering

Oklahoma State University

Fall 2007

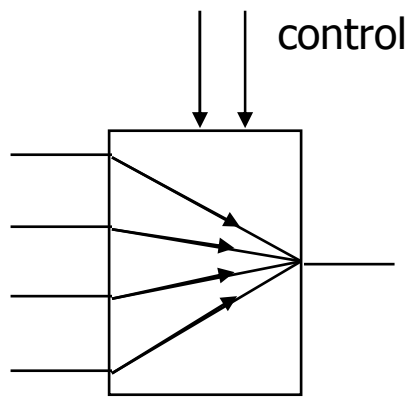
Combinational Logic Technologies

- Switching logic
 - Multiplexers
 - decoders

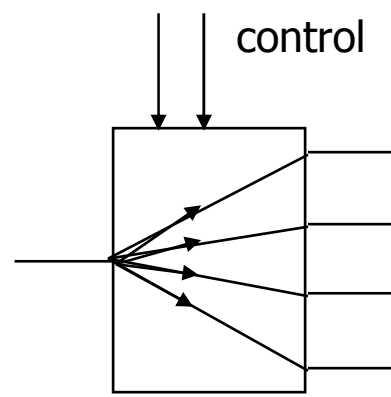
With this topic we will think in terms of “function” as opposed to logic – although we can certainly use Boolean algebra to describe any logic. It’s easier this way when the functions become more complex.

Making connections – Switching Logic

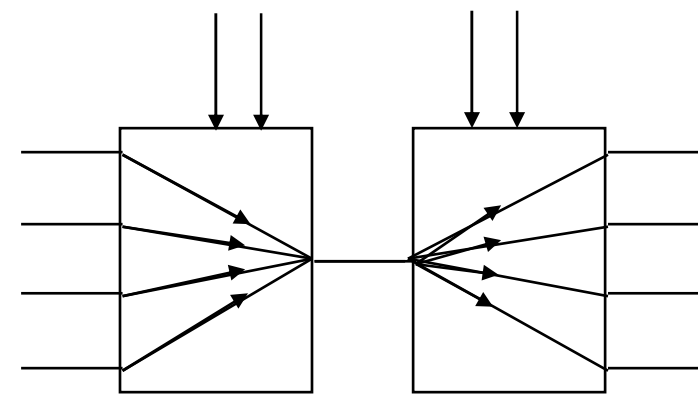
- Direct point-to-point connections between gates
 - wires we've seen so far
- Route one of many inputs to a single output --- multiplexer
- Route a single input to one of many outputs --- demultiplexer



multiplexer



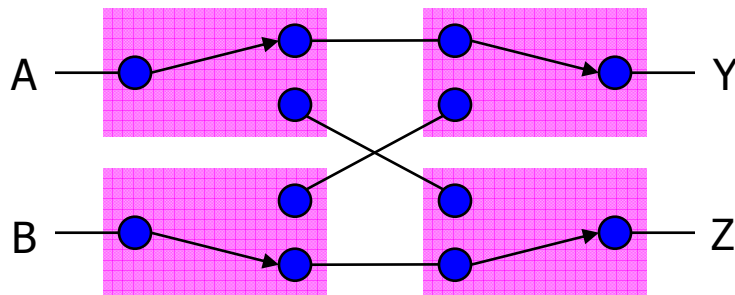
demultiplexer



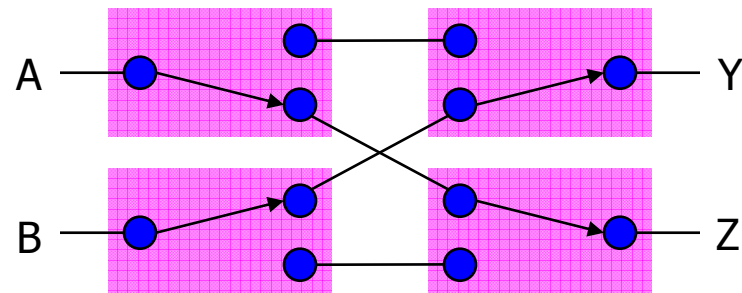
4x4 switch

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

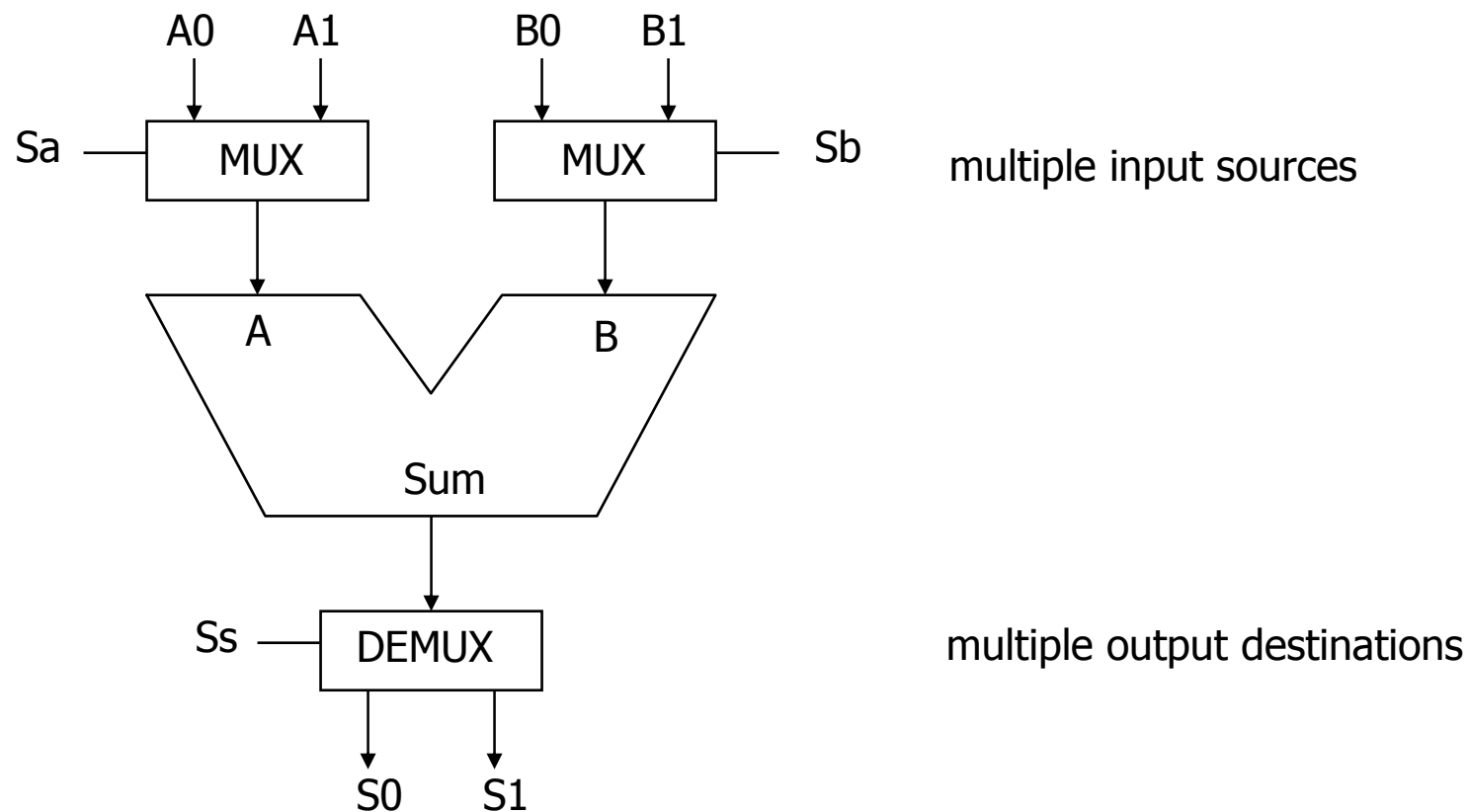


Multiplexers and demultiplexers are sometimes called selectors and decoders, respectively.



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

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

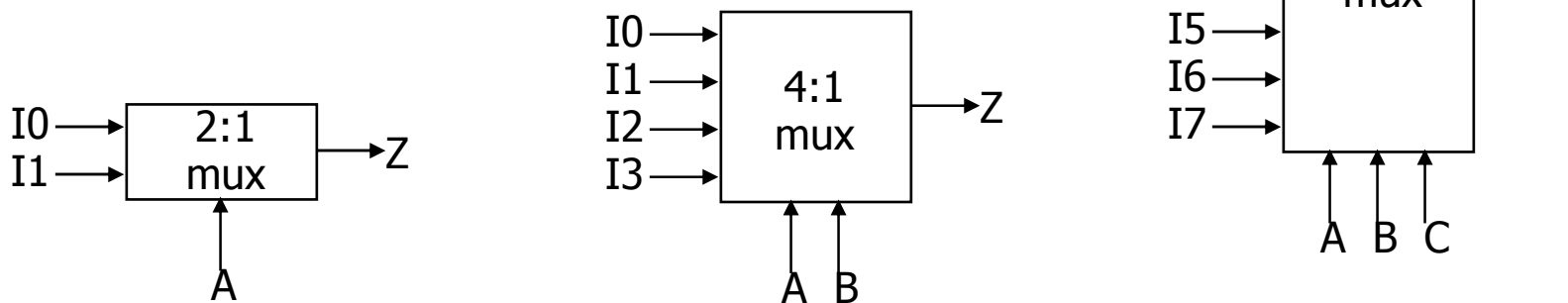
two alternative forms
for a 2:1 Mux truth table

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

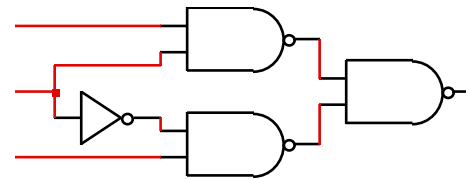
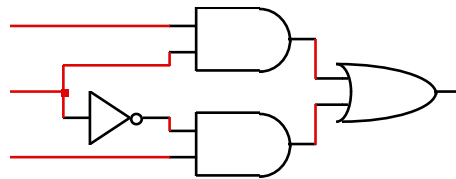
- In general: $Z = \sum_{k=0}^{2^n-1} (m_k I_k)$

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

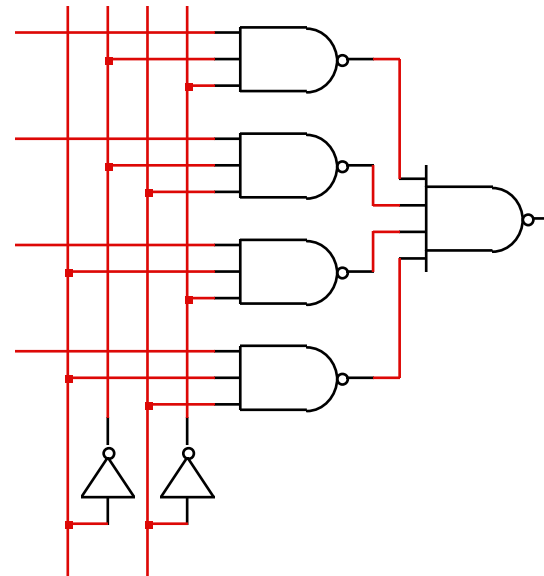
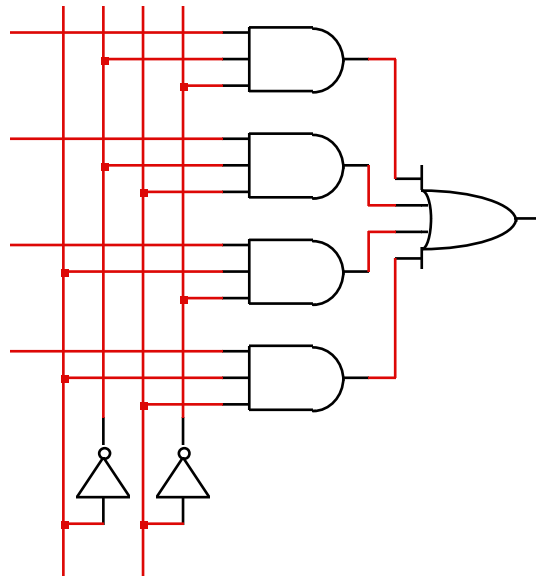


Gate level implementation of muxes

- 2:1 mux

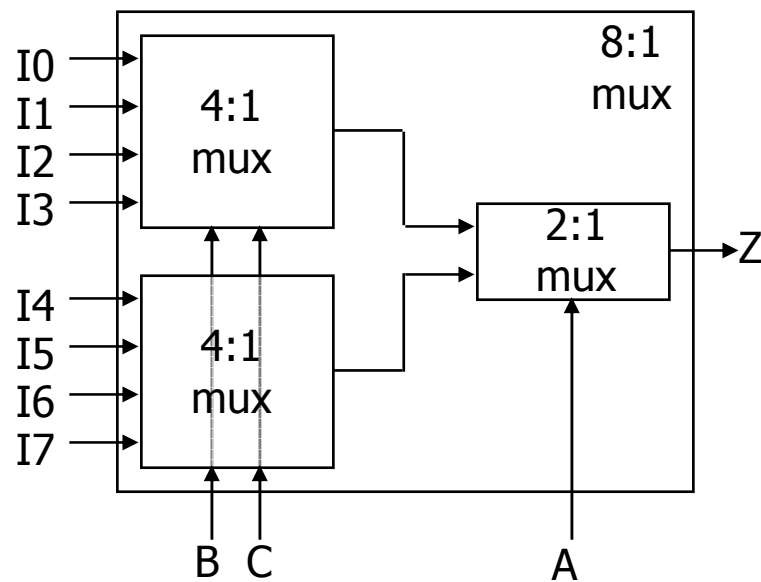


- 4:1 mux



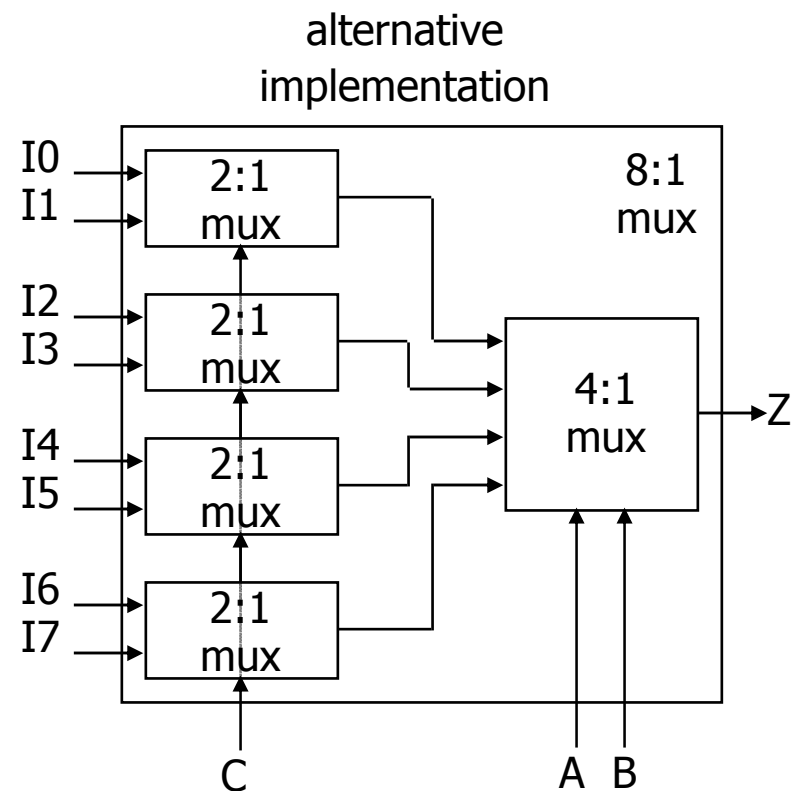
Cascading multiplexers

- 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

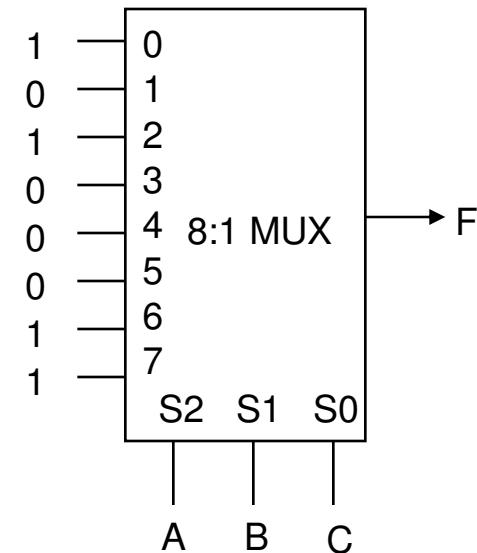
- A $2^n:1$ multiplexer can implement any function of n variables
 - with the variables used as control inputs and
 - the data inputs tied to 0 or 1
 - in essence, a lookup table

- Example:

- $F(A,B,C) = m_0 + m_2 + m_6 + m_7$
 $= A'B'C' + A'BC' + ABC' + ABC$

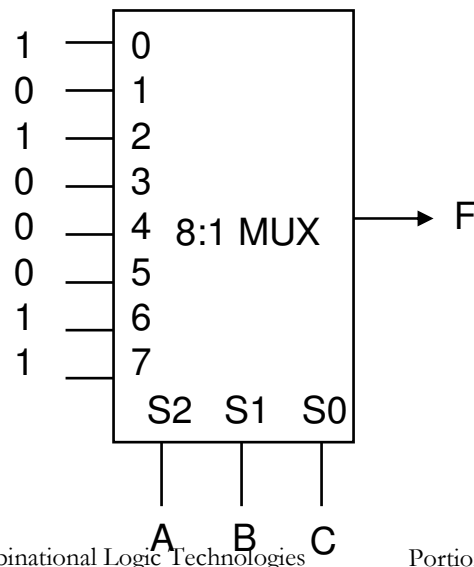
$$\begin{aligned} &= A'B'C'(1) + A'B'C(0) \\ &\quad + A'BC'(1) + A'BC(0) \\ &\quad + AB'C'(0) + AB'C(0) \\ &\quad + ABC'(1) + ABC(1) \end{aligned}$$

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

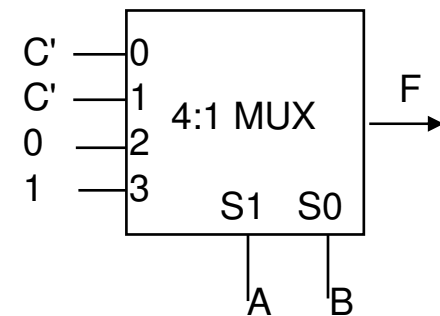


Multiplexers as general-purpose logic (cont'd)

- 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



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:

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

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

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

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

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

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

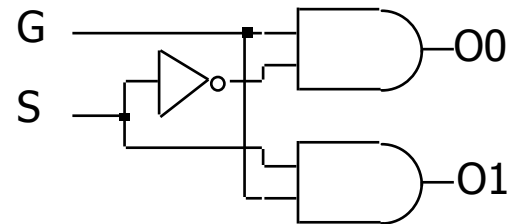
$$O6 = G \bullet S2 \bullet S1 \bullet S0'$$

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

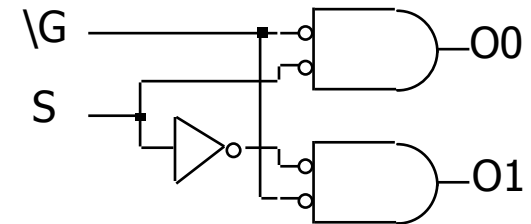
Gate level implementation of demultiplexers

- 1:2 decoders

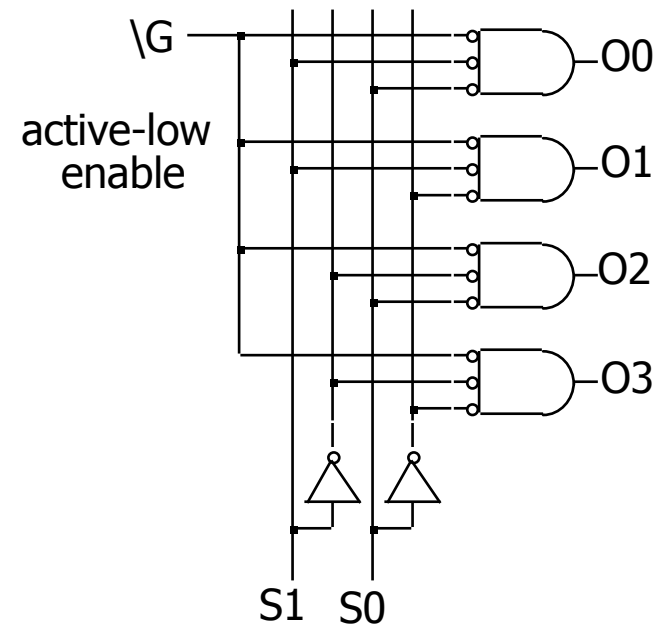
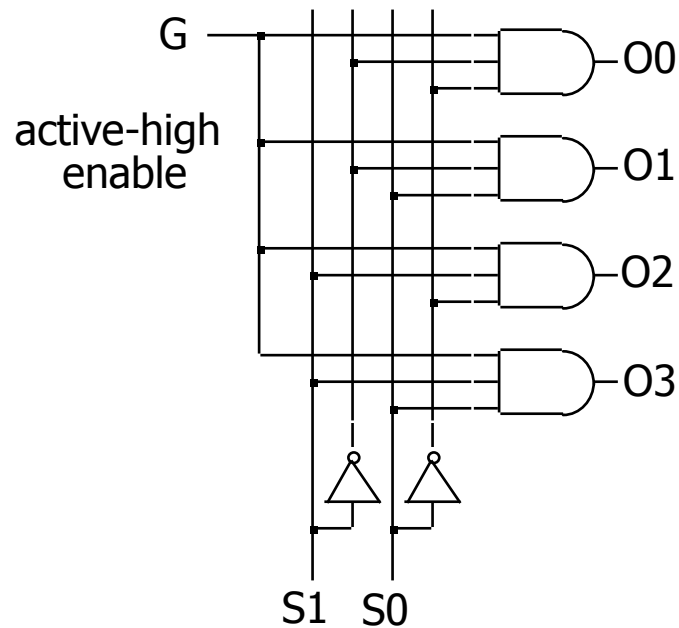
active-high enable



active-low enable

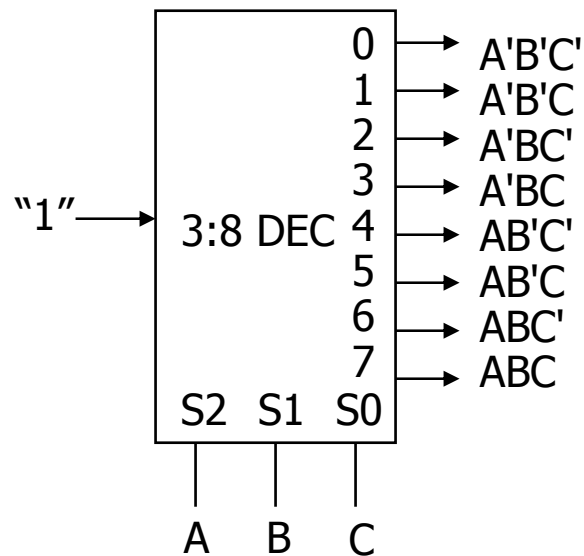


- 2:4 decoders



Demultiplexers as general-purpose logic

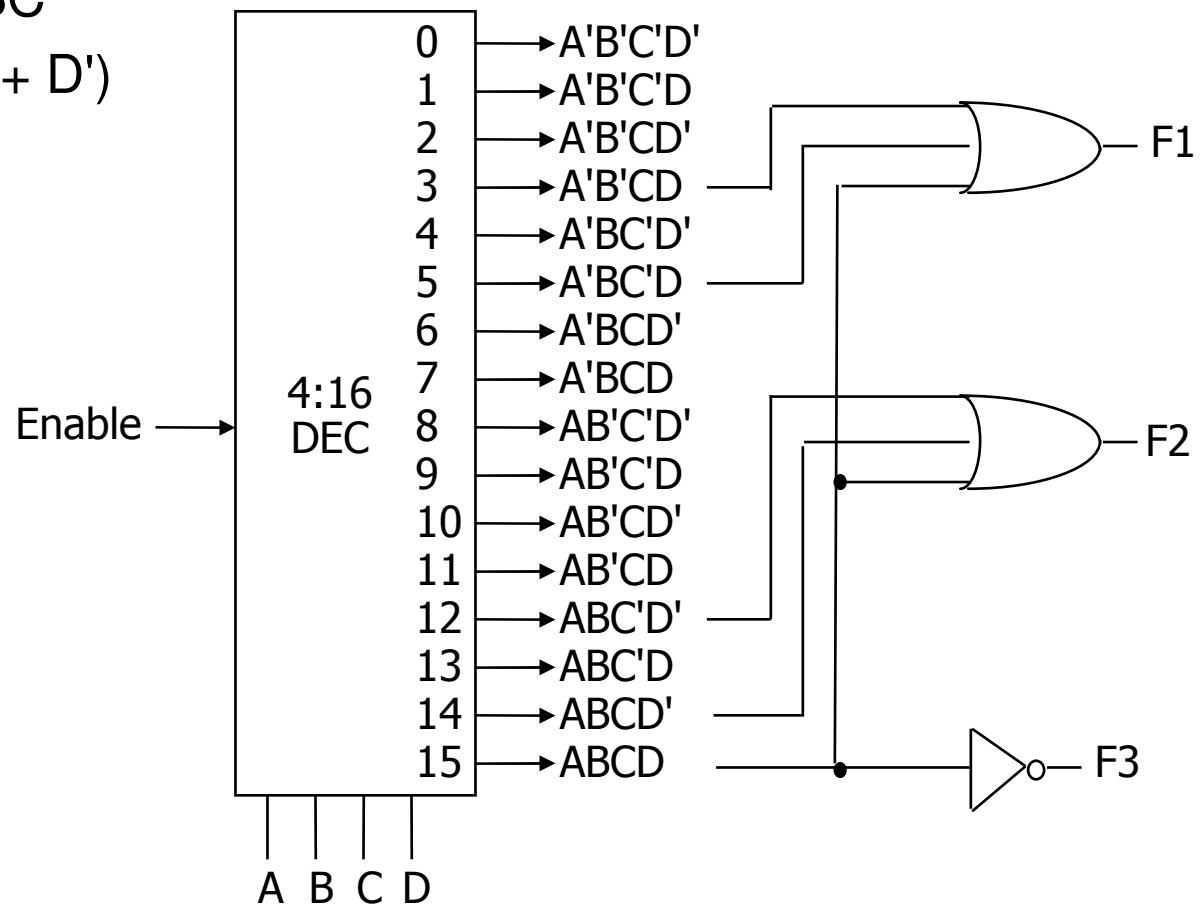
- Can be used as general logic building blocks
- A $n:2^n$ decoder can implement any function of n variables
 - with the variables used as control inputs
 - the enable inputs tied to 1 and
 - the appropriate minterms summed to form the function



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

Demultiplexers as general-purpose logic (cont'd)

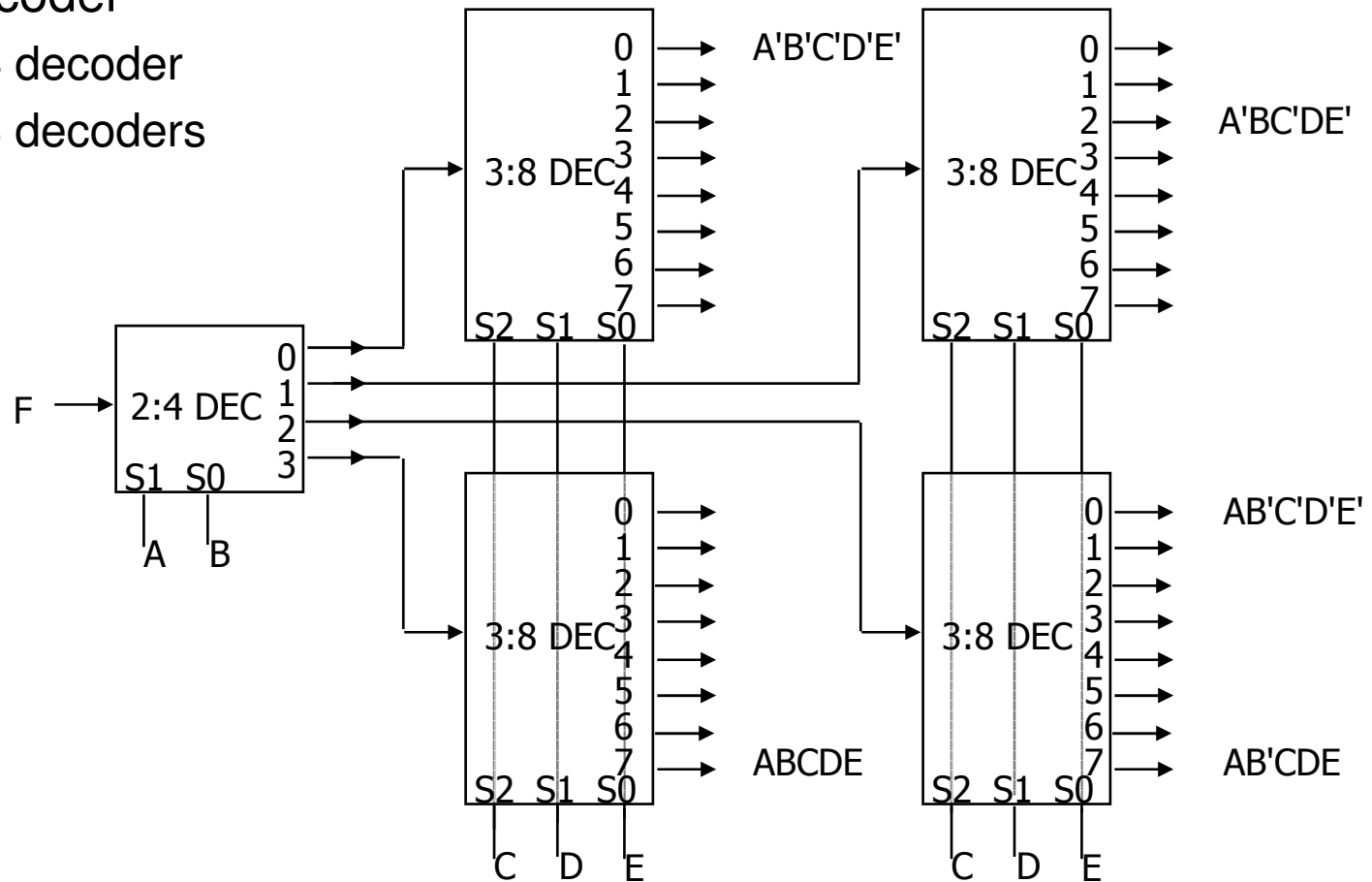
- $F1 = A'BC'D + A'B'CD + ABCD$
- $F2 = ABC'D' + ABC$
- $F3 = (A' + B' + C' + D')$



Cascading decoders

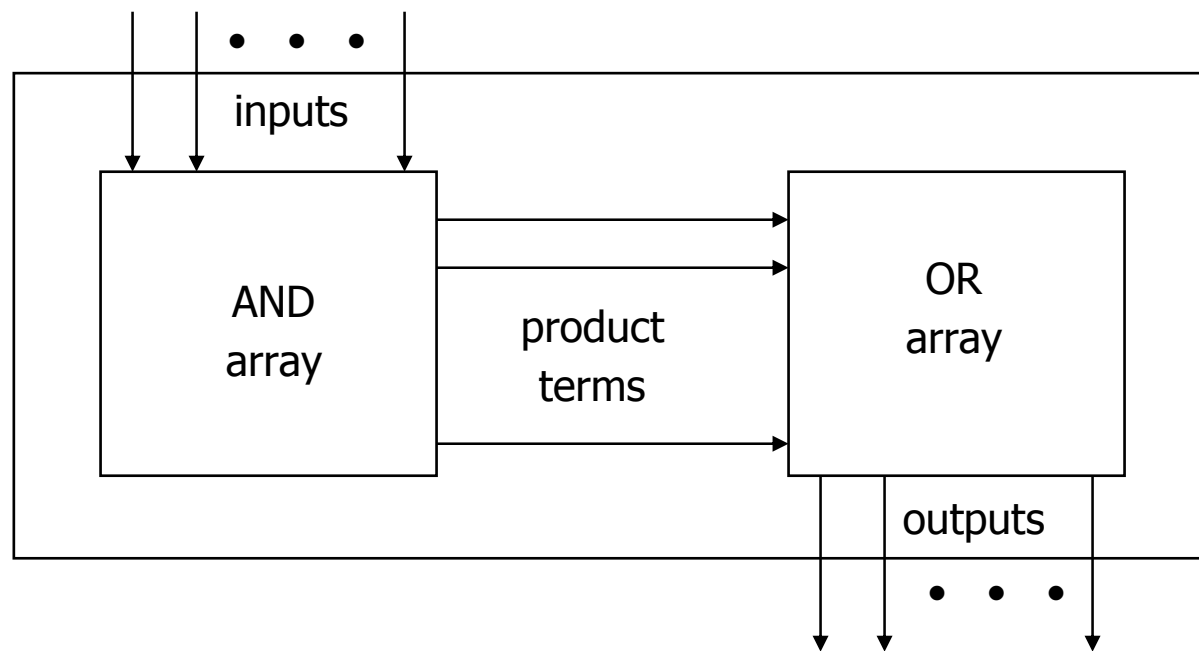
- 5:32 decoder

- 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:

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

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

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

$$F3 = B' C + A$$

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

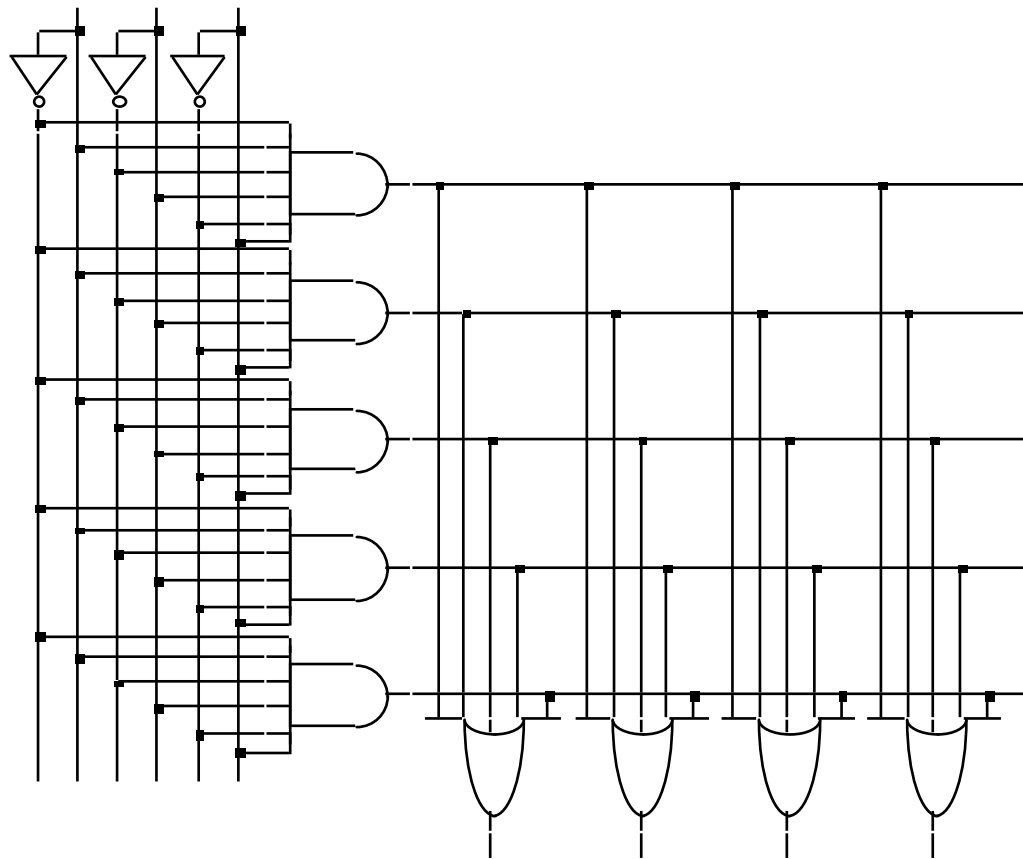
output side:

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

reuse of terms

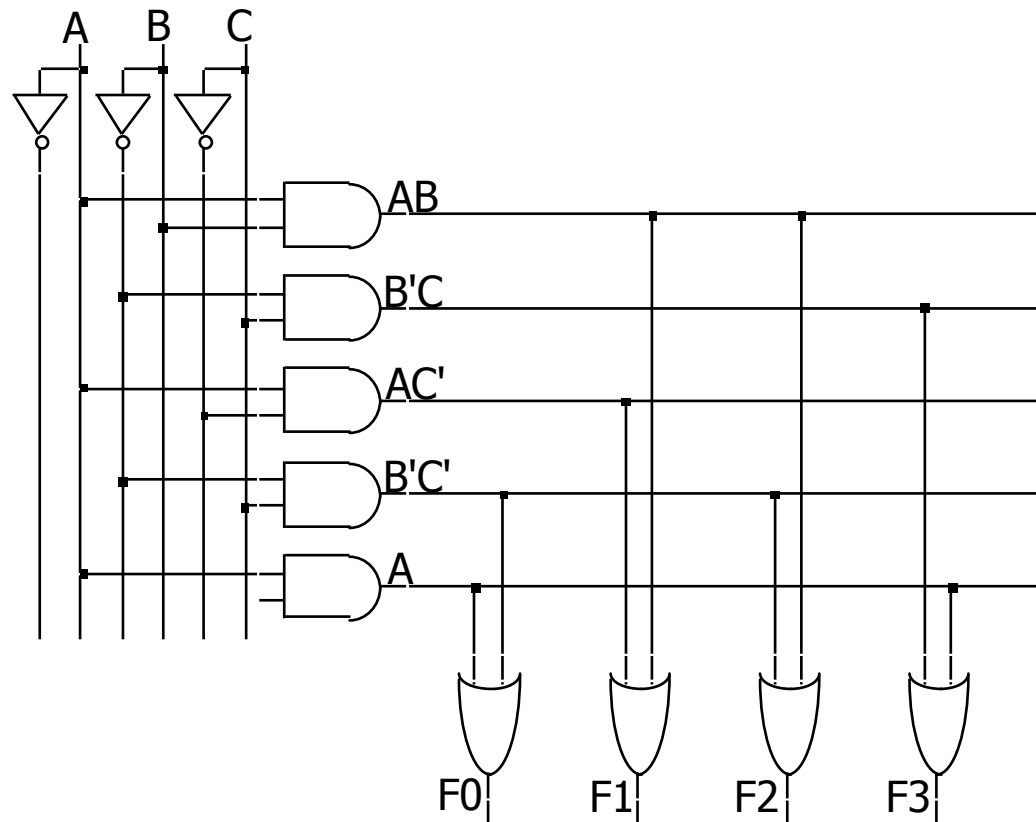
Before programming

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



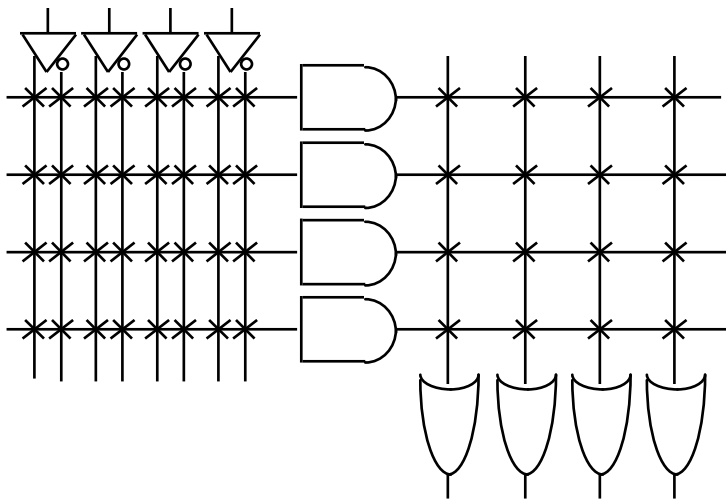
After programming

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



Alternate representation for high fan-in structures

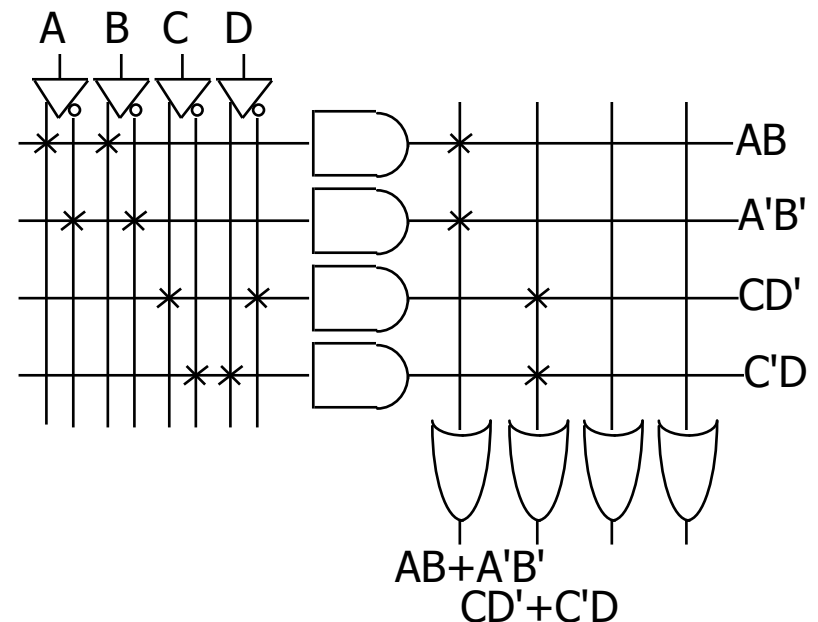
- Short-hand notation so we don't have to draw all the wires
 - × signifies a connection is present and perpendicular signal is an input to gate



notation for implementing

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

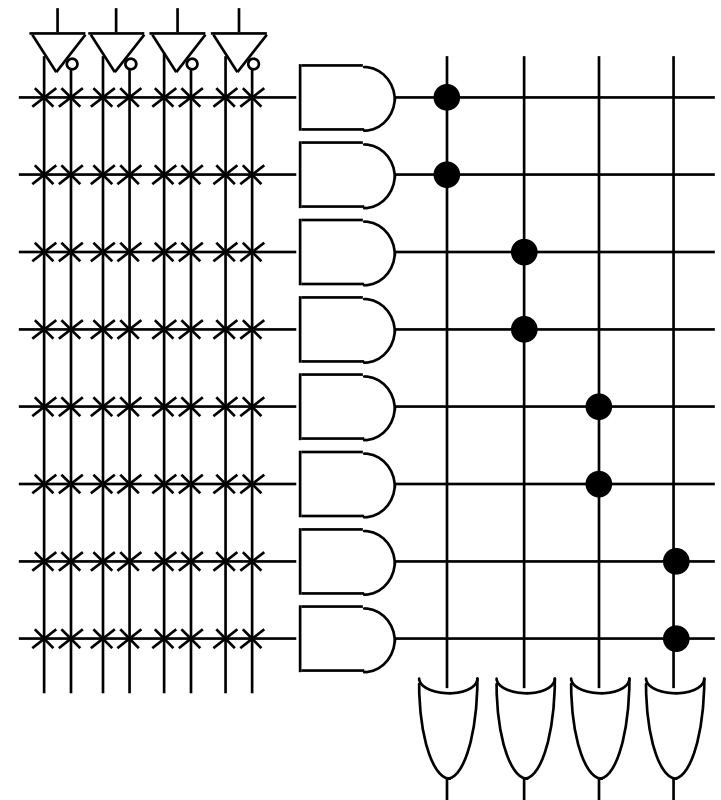
$$F1 = C D' + C' D$$



PALs and PLAs

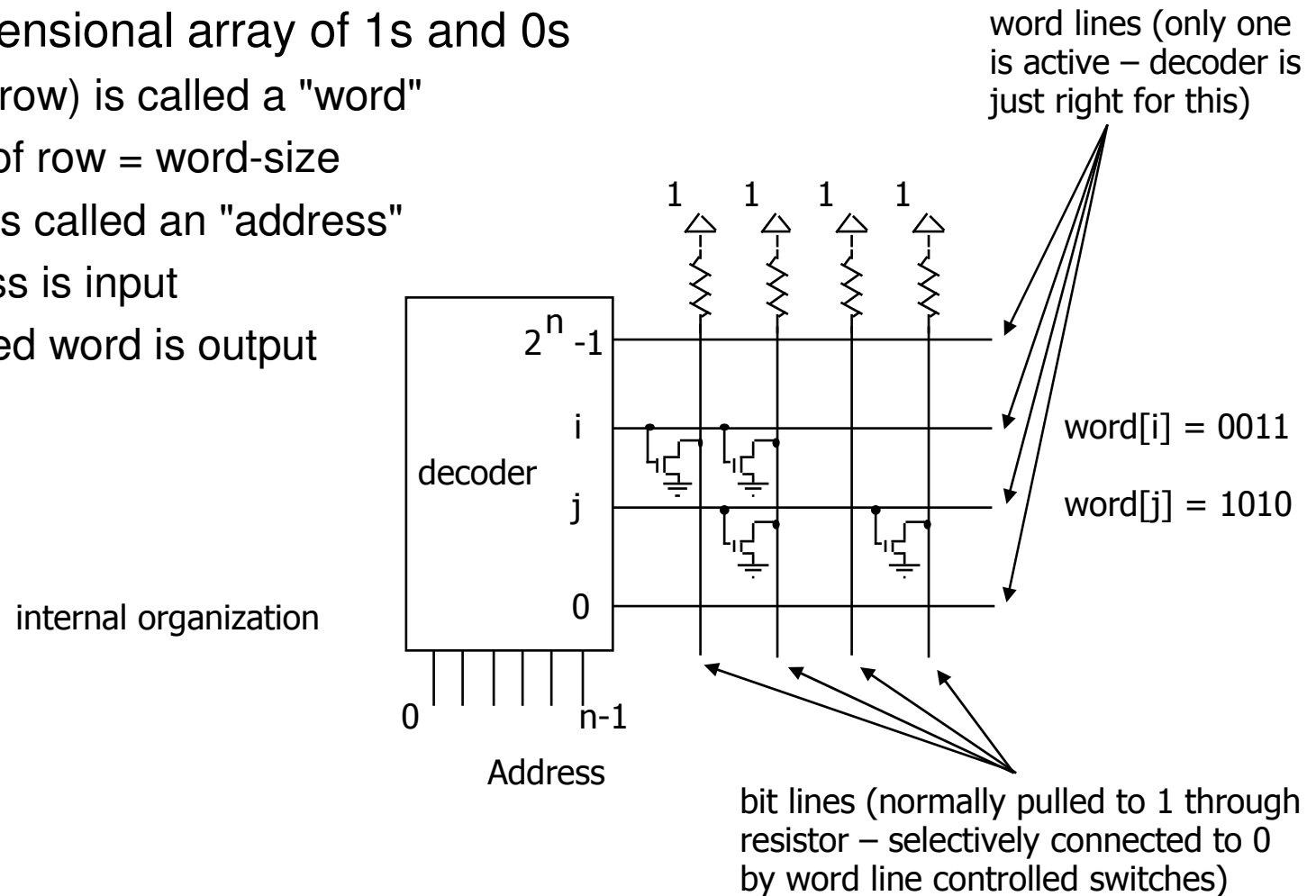
- Programmable logic array (PLA)
 - what we've seen so far
 - unconstrained fully-general AND and OR arrays
- Programmable array logic (PAL)
 - constrained topology of the OR array
 - innovation by Monolithic Memories
 - faster and smaller OR plane

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



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

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

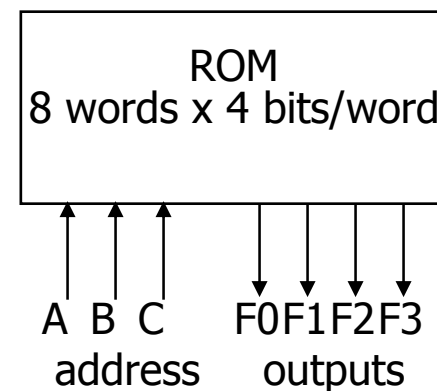
$$F1 = A' B' C + A' B C' + A B C$$

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

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

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

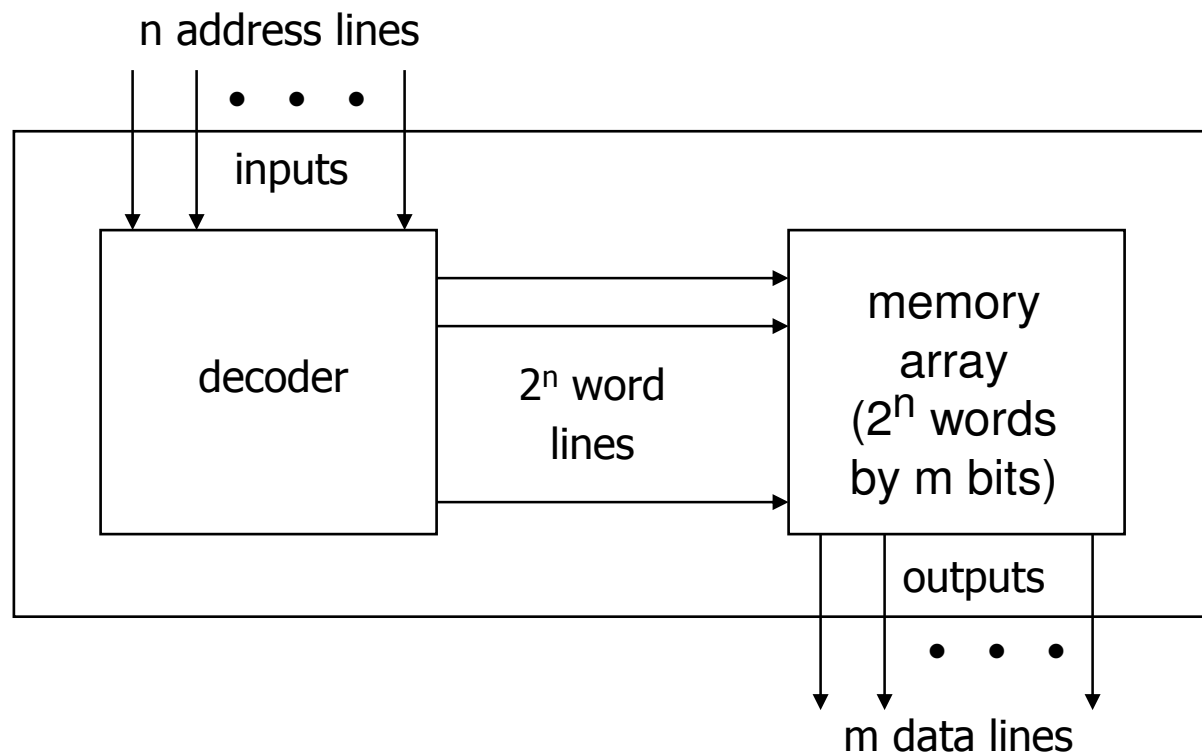
truth table



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
 - field programmable gate arrays (FPGAs) 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)

Tri-State and Open-Collector

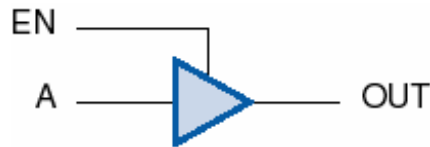
The Third State

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 (EN)

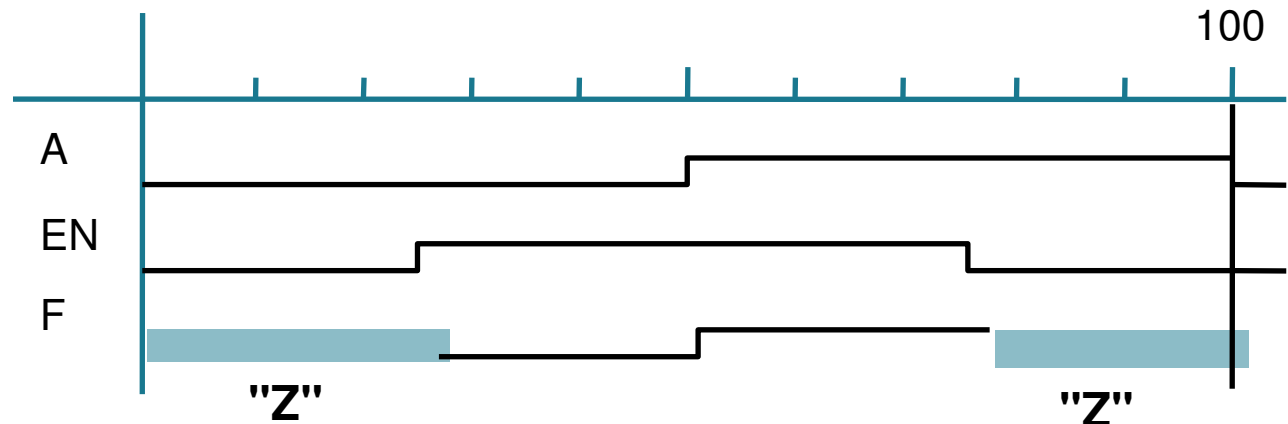


When EN is high, this gate is a non-inverting "buffer"

When EN 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*

Non-inverting buffer's timing waveform



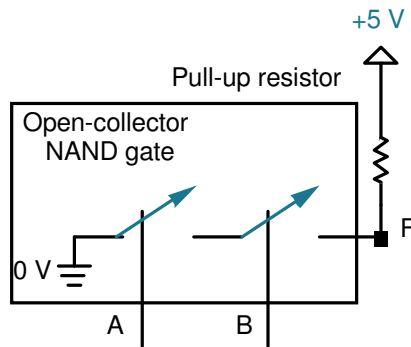
Tri-State and Open Collector

Open Collector

Another way to connect multiple gates to the same output wire

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



OC NAND gates

Wired AND:

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!

