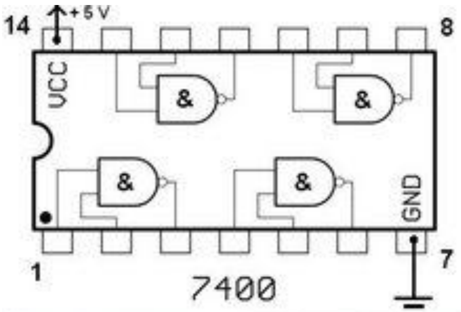
Chapter 4. Combinational logic technologies

Combinational Logic Technologies

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



The simplest way to implement logic circuits would be using standard gates. However, as more complicated and diverse logic systems are getting required, a wealth of implementation techniques are proposed. There are three major categories in logic implementation technologies.

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

Again it is easy and simple to use standard gates such as NAND and AND, called random logic. Since a single gate is not good to sell and buy, a few or several gates are packed into a package. Right now, it is not widely used for economical reasons since there are a lot of IC packages in the catalog.

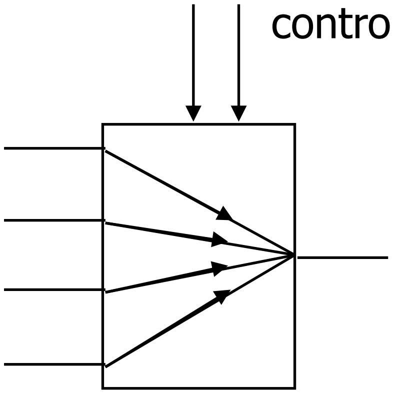
Regular logic

- **Need to make design faster**
- **Need to make engineering changes easier to make**
- **MUX and DEMUX**

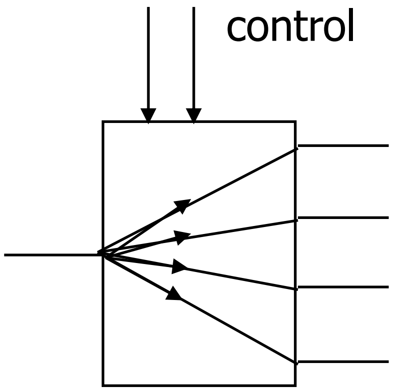
Unlike random logic, regular logic refers to a flexible component that performs a specific high-level function compared to primitive logic gates. Design becomes easier with these regular logic components since each component performs a specific function. Sometimes we can flexibly exploit the regular logic components for other purposes than its original one.

Making connections

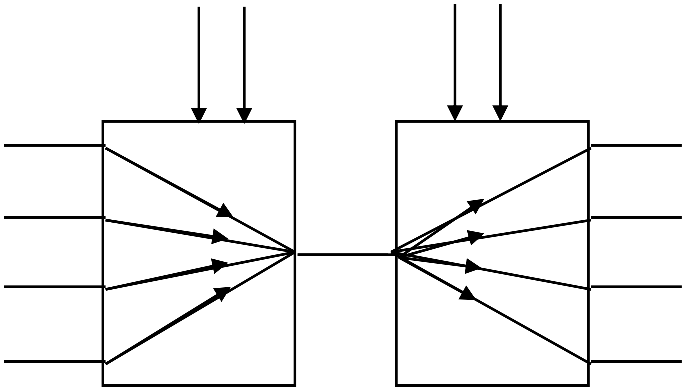
- **Direct point-to-point connections between gates**
  - wires we've seen so far
- **Route one of many inputs to a single output - multiplexer (MUX)**
- **Route a single input to one of many outputs - demultiplexer (DEMUX)**



multiplexer



demultiplexer

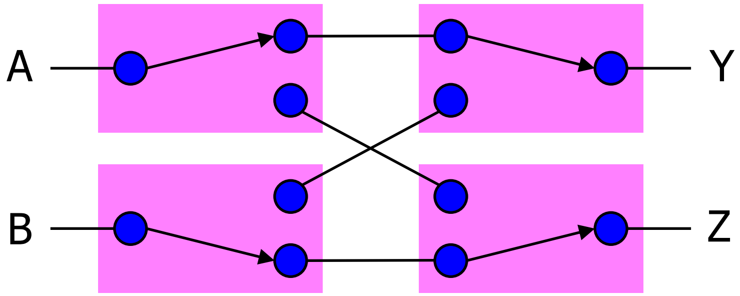


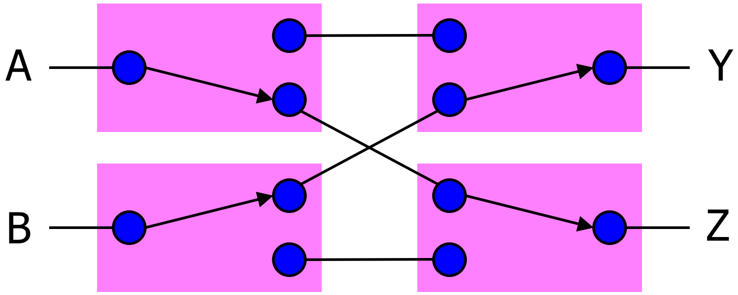
4x4 switch

Two popular regular logic components are MUX and DEMUX. A MUX selects one of its data inputs to the output by the control inputs; a MUX is also called a selector. The diagram on the left shows a 4-input MUX. Can you guess the relation between the number of data inputs and the number of control lines? A DEMUX performs the reverse function, often called a decoder. We can clearly see that these components perform higher-level functions compared to logic gates

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



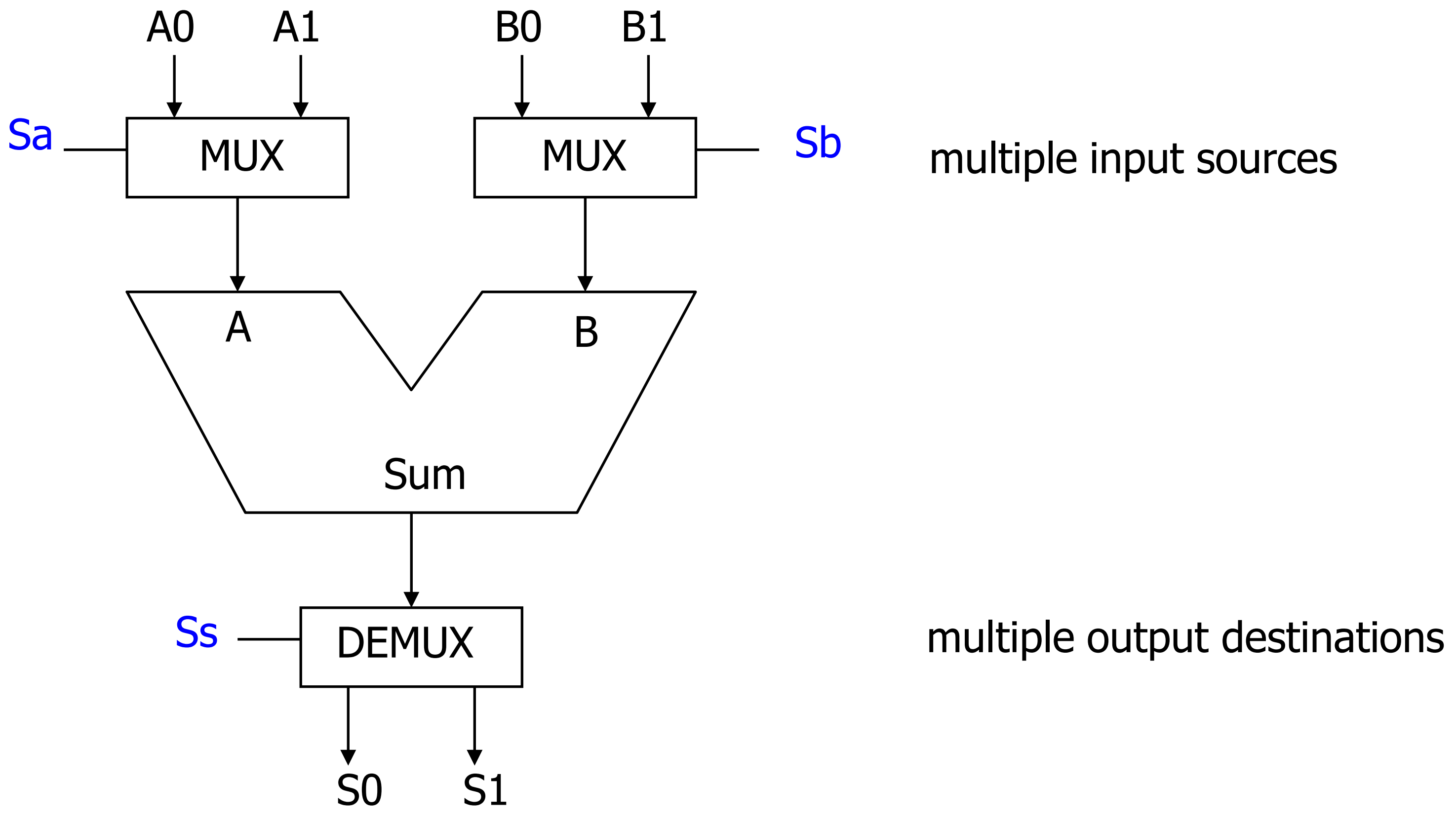


When we combine MUXs and DEMUXs, a switching network can be implemented. Note that control lines are skipped here.

In this slide, there are 2X2 switching networks. By using control variables (which are skipped), A and B can be routed to either Y or Z.

Mux and demux (cont'd)

- **Uses of multiplexers/demultiplexers in multi-point connections**



Let’s see how MUX and DEMUX can be used for a logic system design. Here is a 1-bit adder, a V-shape polygon. There are two sources for each input and two destinations for the resulting sum. So there are total three control variables.

Multiplexers (MUXs)/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

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

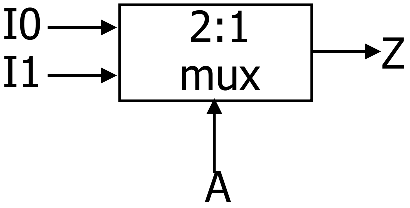
logical form

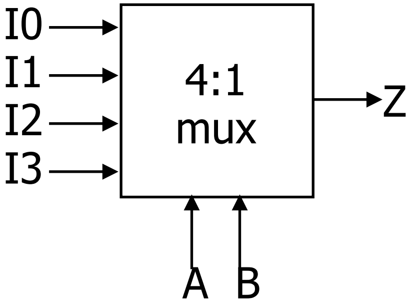
two alternative forms
for a 2:1 Mux truth table

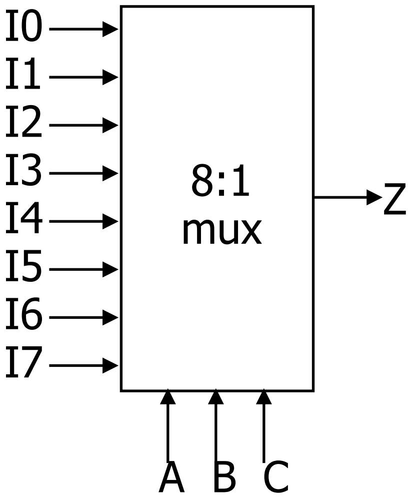
Let’s look at how a MUX function can be described as a boolean expression or a truth table. We will start with the simplest one, 2:1 MUX. There are two inputs I0 and I1 and the control input is A. Then Z will select I0 or I1 depending on A’s value. If we tabulate all the cases of I0 and I1, the final truth table is the one on the right.

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: $Z = \Sigma_{k=0}^{2^n-1}(m_k I_k)$
  - in minterm shorthand form for a 2^n:1 Mux



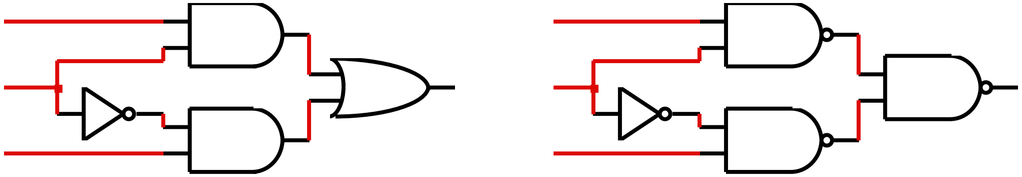




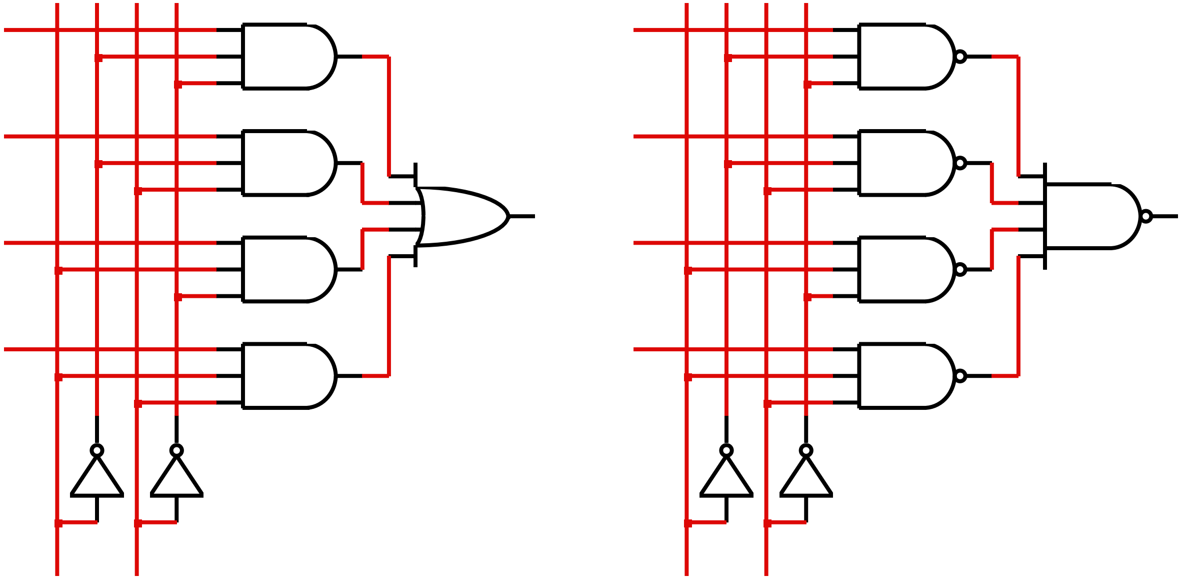
How can we express the output in a general form, regardless of the # of inputs? First of all, n is the # of control wires. So there are total 2**n inputs. Here m_k is the k-th minterm from the control variables.

Gate level implementation of muxes

- **2:1 mux**



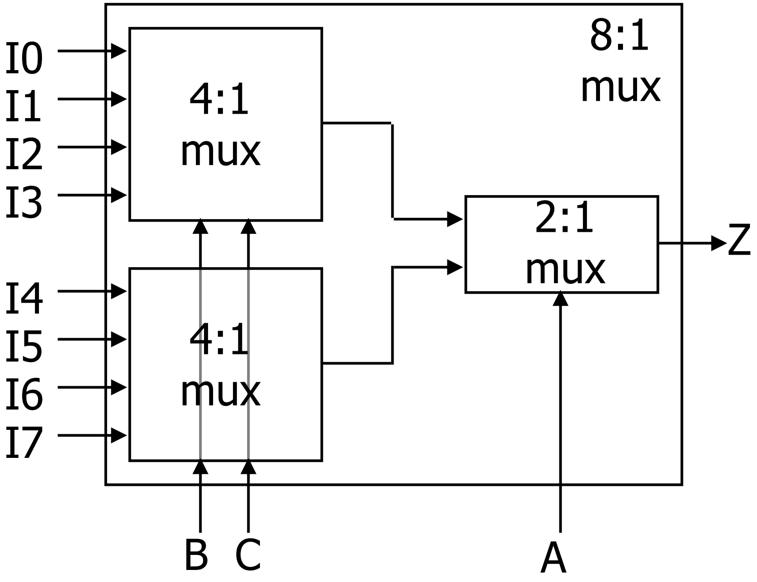
- **4:1 mux**



At the top left, an AND-OR realization of a 2:1 MUX is shown. What are the 2 data inputs and what is the control input?

Cascading multiplexers

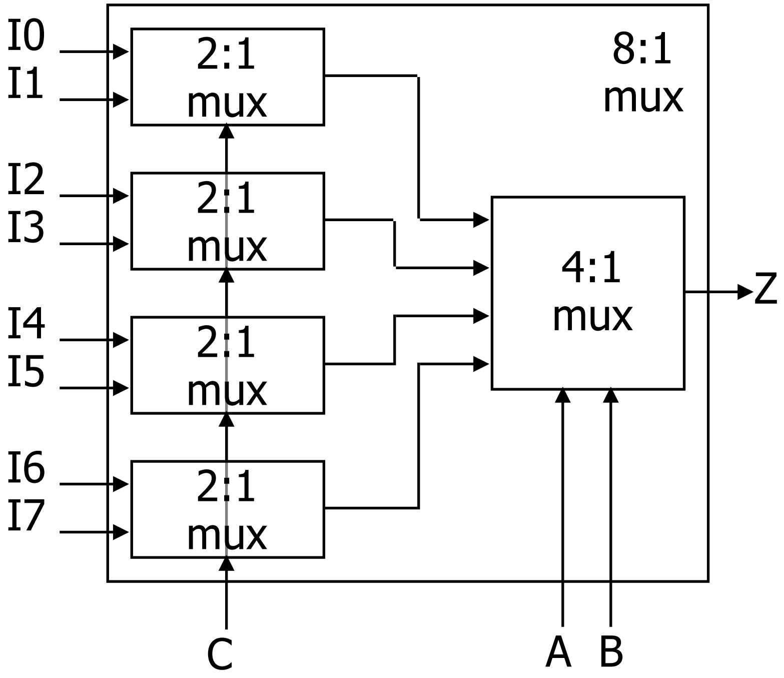
- **Large multiplexers can be made by cascading smaller ones**



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

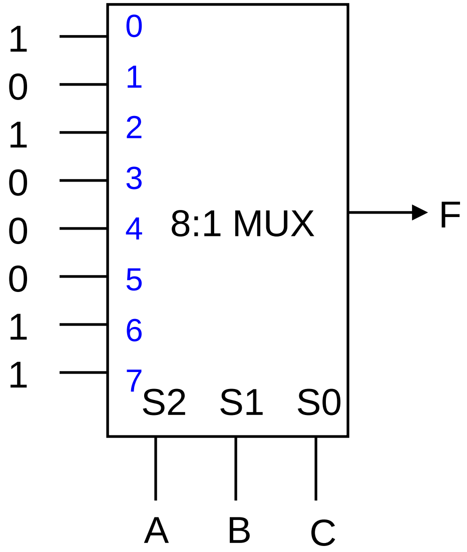
alternative implementation



You can build a large scale MUX in two ways. One option is just to use a single conventional MUX for 2**n inputs and n control lines. Or you can combine small scale MUXs. This slide shows two cases of building an 8:1 MUX from small scale MUXs.

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) = 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)



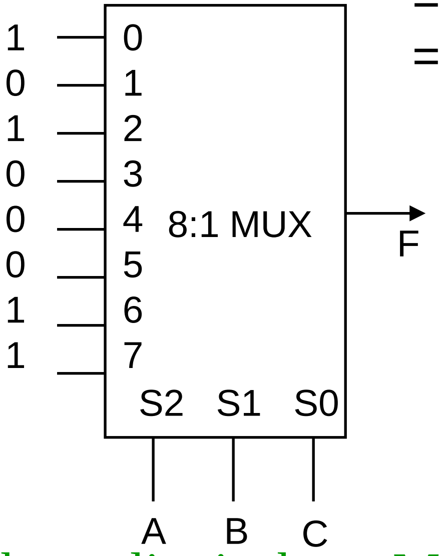
$$F = A'B'C'I_0 + A'B'CI_1 + A'BC'I_2 + A'BCI_3 + AB'C'I_4 + AB'CI_5 + ABC'I_6 + ABCI_7$$

This slide is very important. Actually a MUX can do more than just selection. Suppose each input wire (one of 2**n inputs) is fixed to either 0 or 1. Depending on the control inputs (here A,B,C), the corresponding bit will be popped up to F. This is kind of a lookup table.

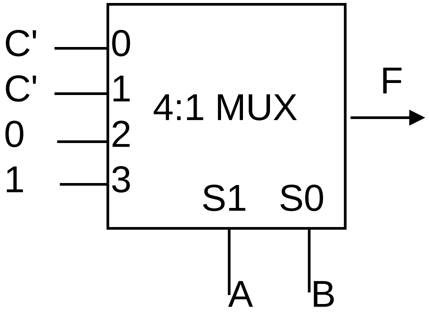
Now look at the system with a different viewpoint. Forget this is a MUX. Suppose A,B,C are the input variables of a logic function F. When will F be true?

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) = 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	



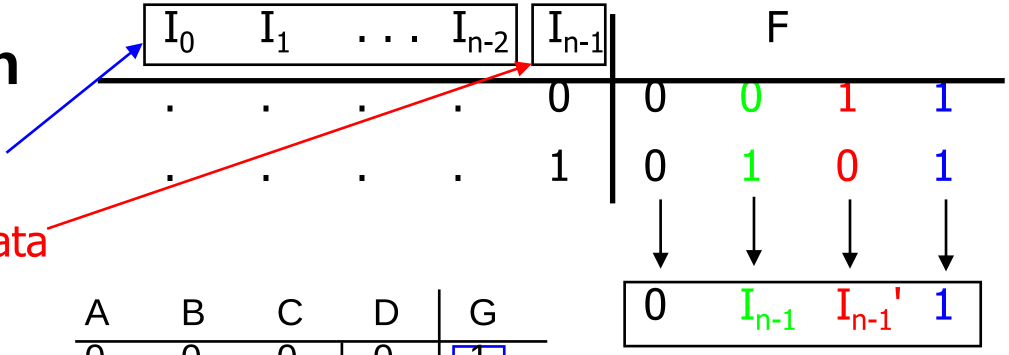
The reality is that a MUX can implement any function of n variables. Here is another example of implementing the same logic function in a simplified way. In this variation, one of the control variable is used as a data input of a 4:1 MUX. Now we have two control variables A and B. Meanwhile C becomes some of the data inputs. Overall, we have 4 cases instead 8 cases by considering F as a function of C

Multiplexers as general-purpose logic (cont’d)

- **Generalization**

n-1 mux control variables

single mux data variable



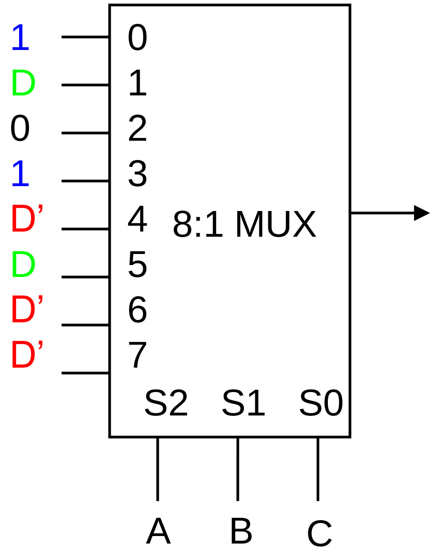
I_0	I_1	. . .	I_{n-2}	I_{n-1}	F			
.	.	.	.	0	0	0	1	1
.	.	.	.	1	0	1	0	1
					0	I_{n-1}	I_{n-1}'	1

four possible configurations of truth table rows can be expressed as a function of I_{n-1}

- **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	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	



Here is the generalized n-input logic function by a (n-1)-input MUX. Depending on the output of two cases of the singled-out variable, a different data input is attached to each minterm of the (n-1) input MUX. Reducing the MUX size is economical.

Demultiplexers (DEMUXs)/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 • S′
O1 = G • S

2:4 Decoder:

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

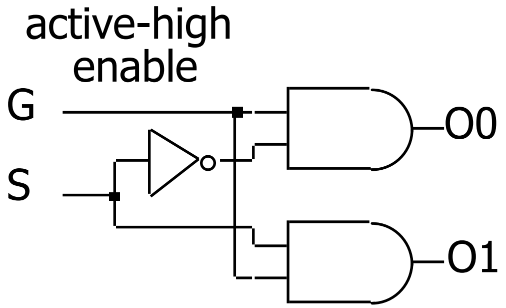
3:8 Decoder:

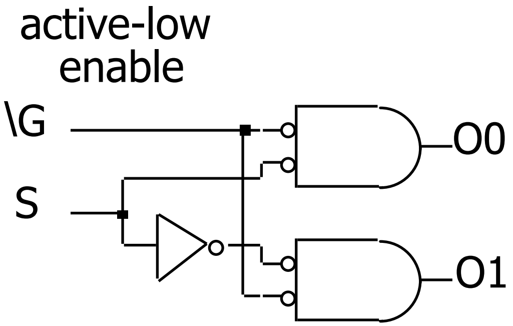
O0 = G • S2′ • S1′ • S0′
O1 = G • S2′ • S1′ • S0
O2 = G • S2′ • S1 • S0′
O3 = G • S2′ • S1 • S0
O4 = G • S2 • S1′ • S0′
O5 = G • S2 • S1′ • S0
O6 = G • S2 • S1 • S0′
O7 = G • S2 • S1 • S0

There is only one data input in DEMUXs, often denoted by G, which will be carried to one of the outputs. The control inputs are often denoted by S and the index of the control wires. Again, for n control inputs, we have 2**n outputs.

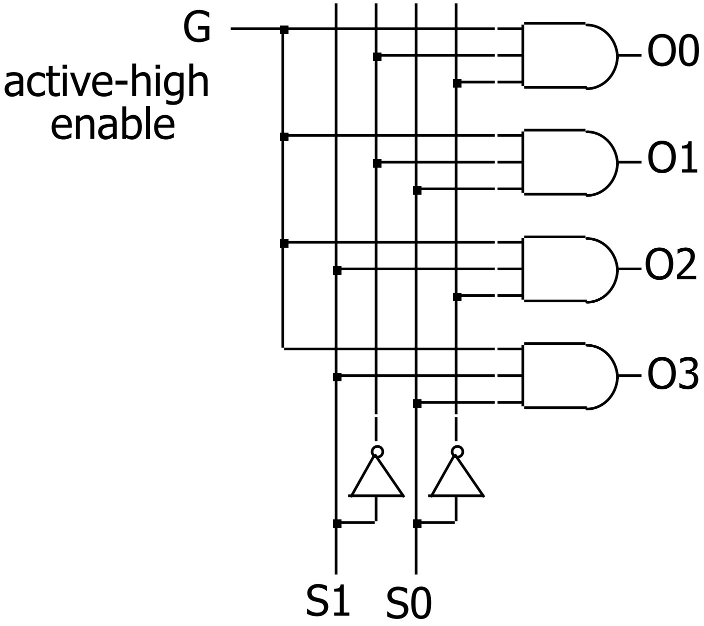
Gate level implementation of demultiplexers

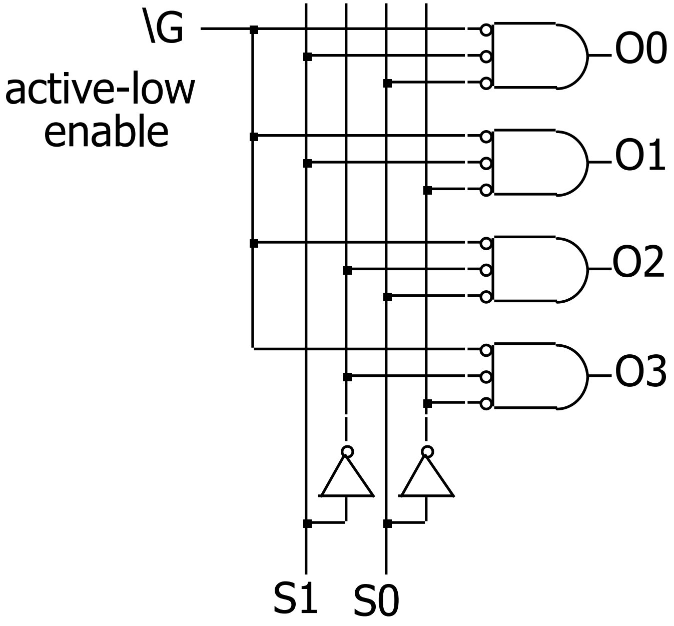
- **1:2 decoders**





- **2:4 decoders**

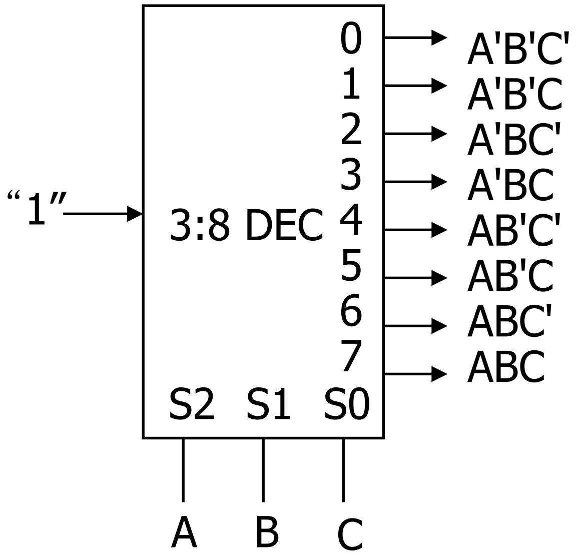




This slide shows a few DEMUXs implemented by logic gates. We can add two bubbles for each input. The reason for inserting two bubbles is to implement the DEMUX by NOR gates

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 input is tied to 1 and
  - the appropriate minterms summed to form the function

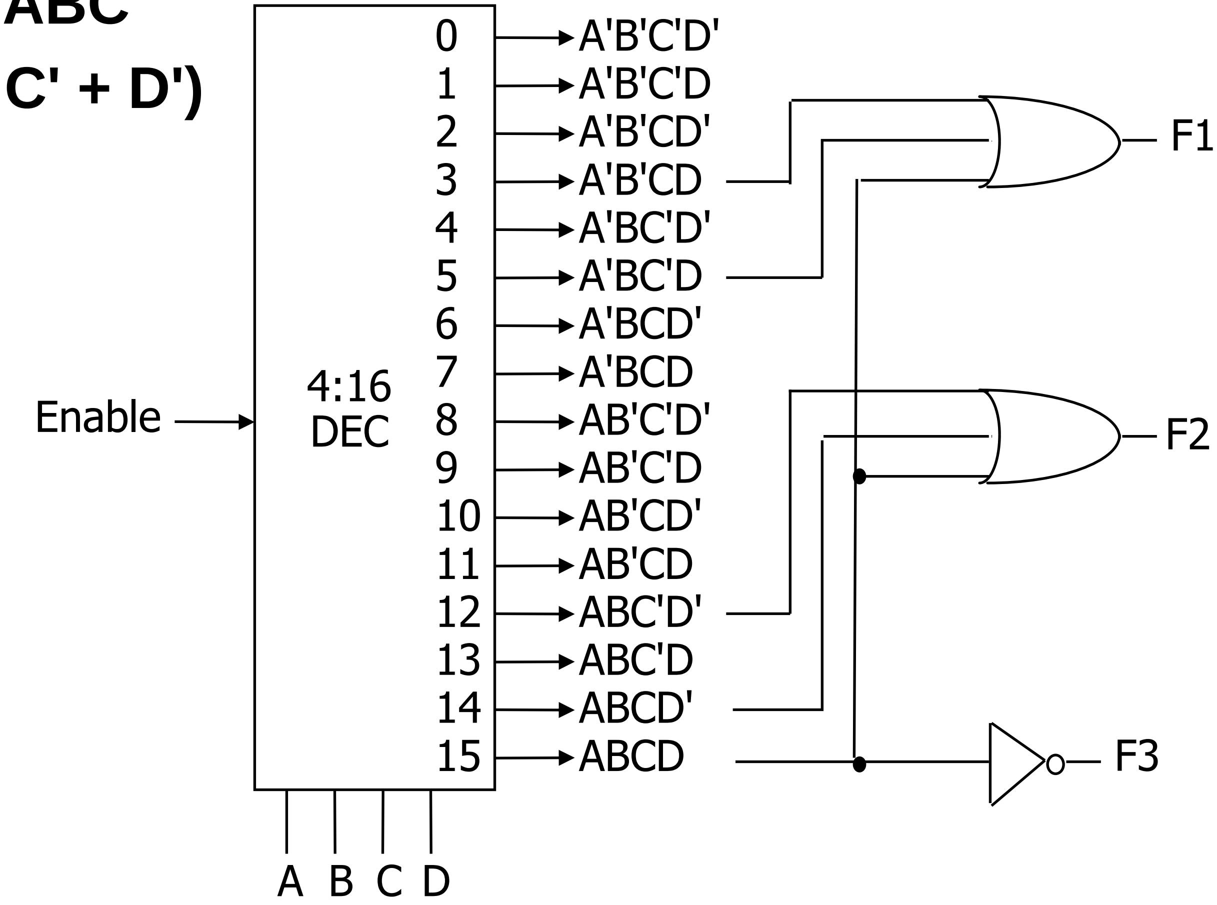


demultiplexer generates appropriate minterm based on control signals (it "decodes" control signals)

Like a MUX, a DEMUX can also perform a logic function of n variables. Here n variables are used for the control wires of the DEMUX. Depending on the values of control wires, a specific minterm will be asserted. Then what we need to do is ORing the relevant minterms for each output function F.

Demultiplexers as general-purpose logic (cont'd)

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

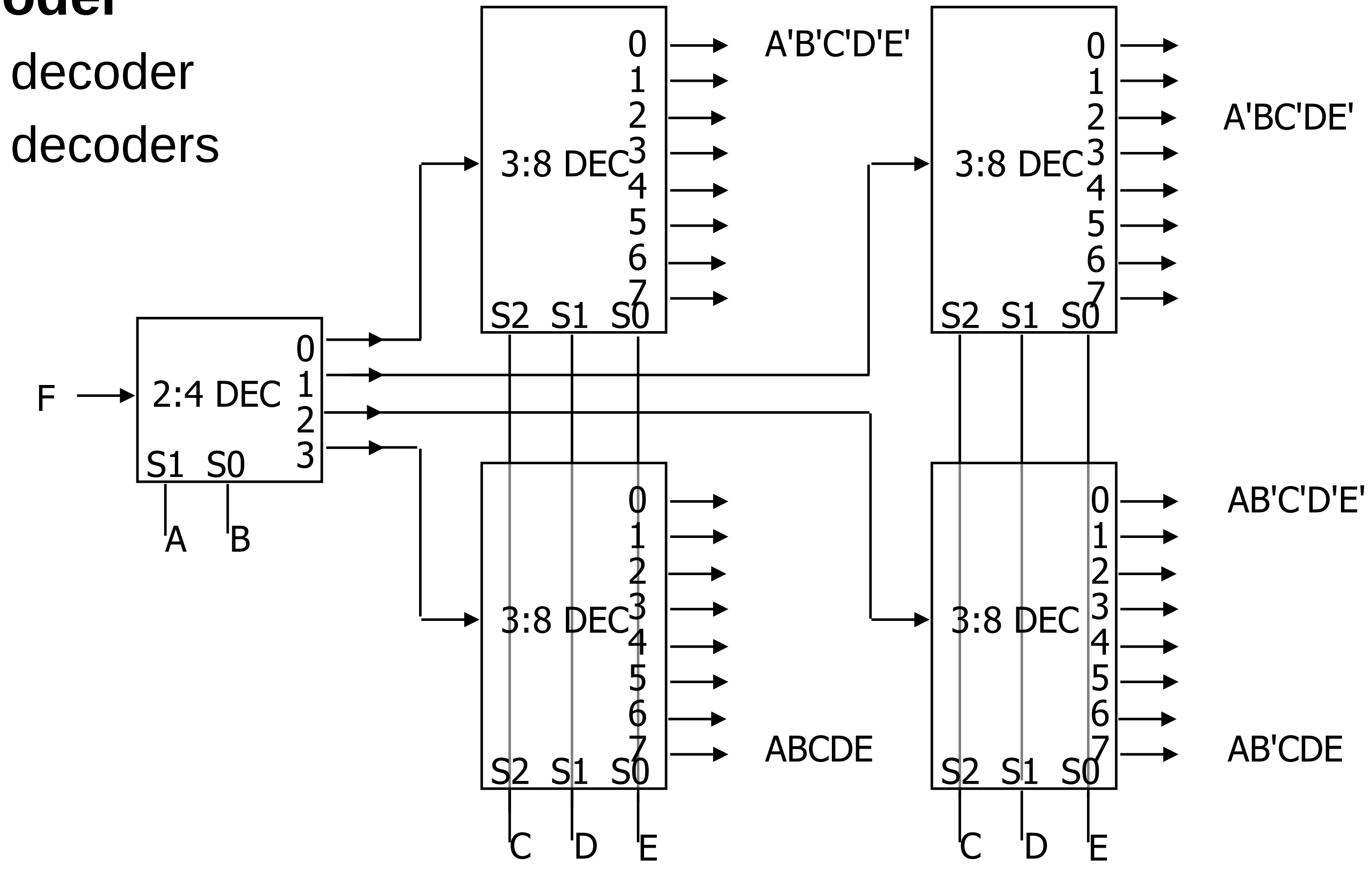


DEMUX is a minterm generator!

By using a single 4:16 DEMUX, we can implement three functions of 4 variables, with a few more gates.

Cascading decoders

- **5:32 decoder**
  - 1 x 2:4 decoder
  - 4 x 3:8 decoders

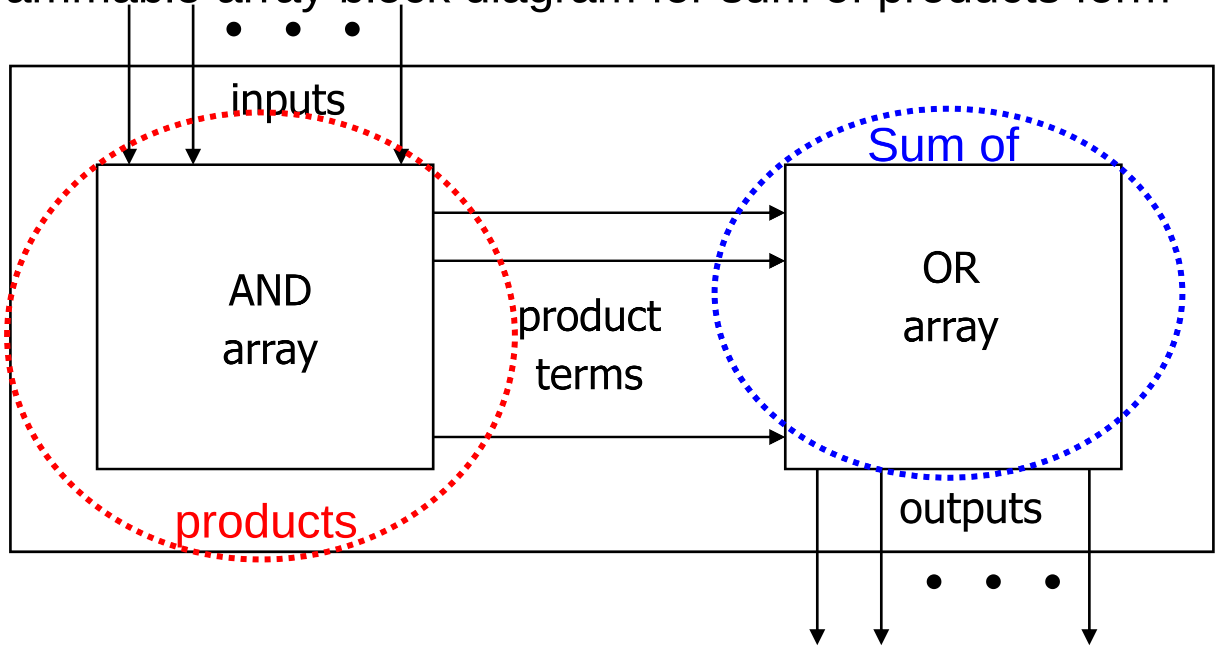


By combining small scale decoders or demuxes, we can build a larger-scale demux.

Here, we have 5 control lines (A,B,C,D,E) to route the enable line, F, to one of 32 output lines

Programmable logic arrays (PLAs)

- **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



A PLA is a general implementation of sum of products of a logic function. We first implement each product term (not necessarily minterm). Then the product terms will be Ored in the next stage. So we have two generic logic arrays. Programming means configuring connections in two arrays.

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	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:

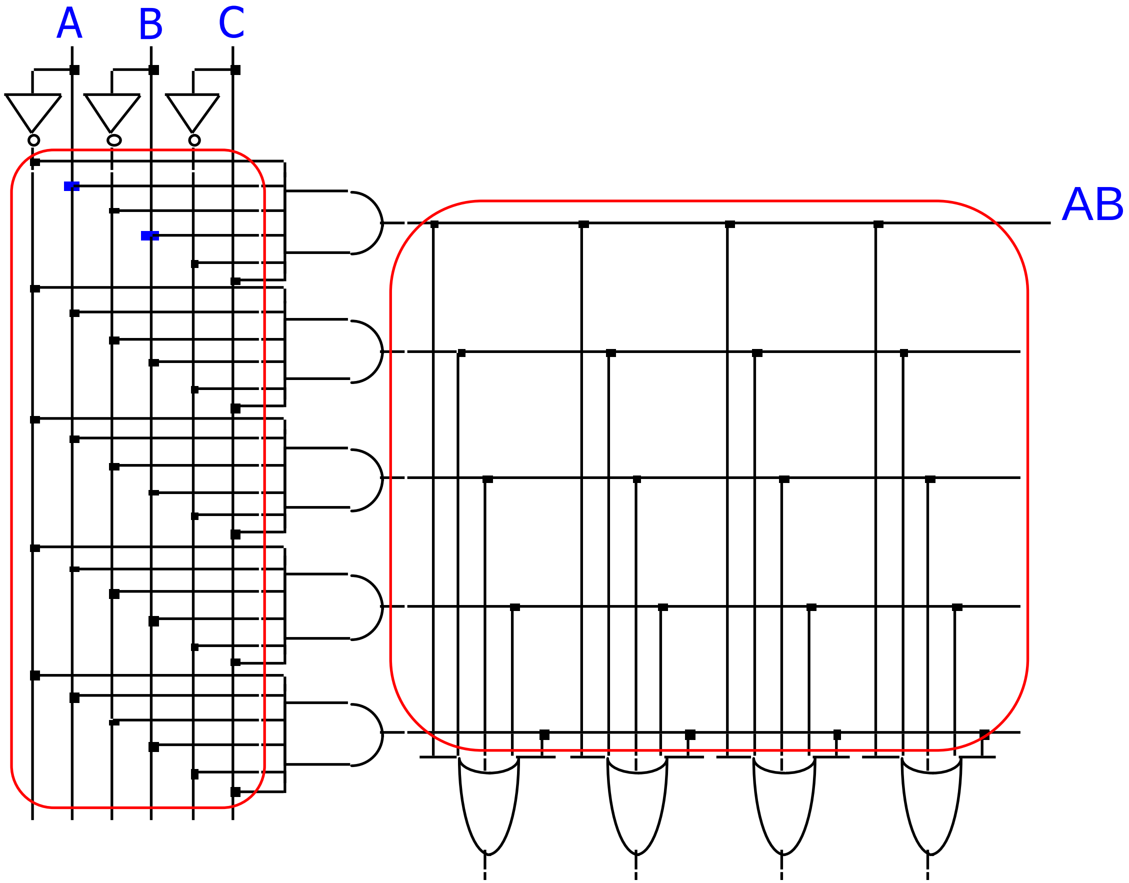
1 = term connected to output
0 = no connection to output

reuse of terms

Let’s take some logic functions for example to illustrate how we can use an PLA for logic implementation. There are 4 functions of three input variables. To use an PLA systematically, it is good to write a personality matrix. The middle section indicates how input variables are combined in the AND array and the right section shows how those product terms are combined in the OR array.

Before programming

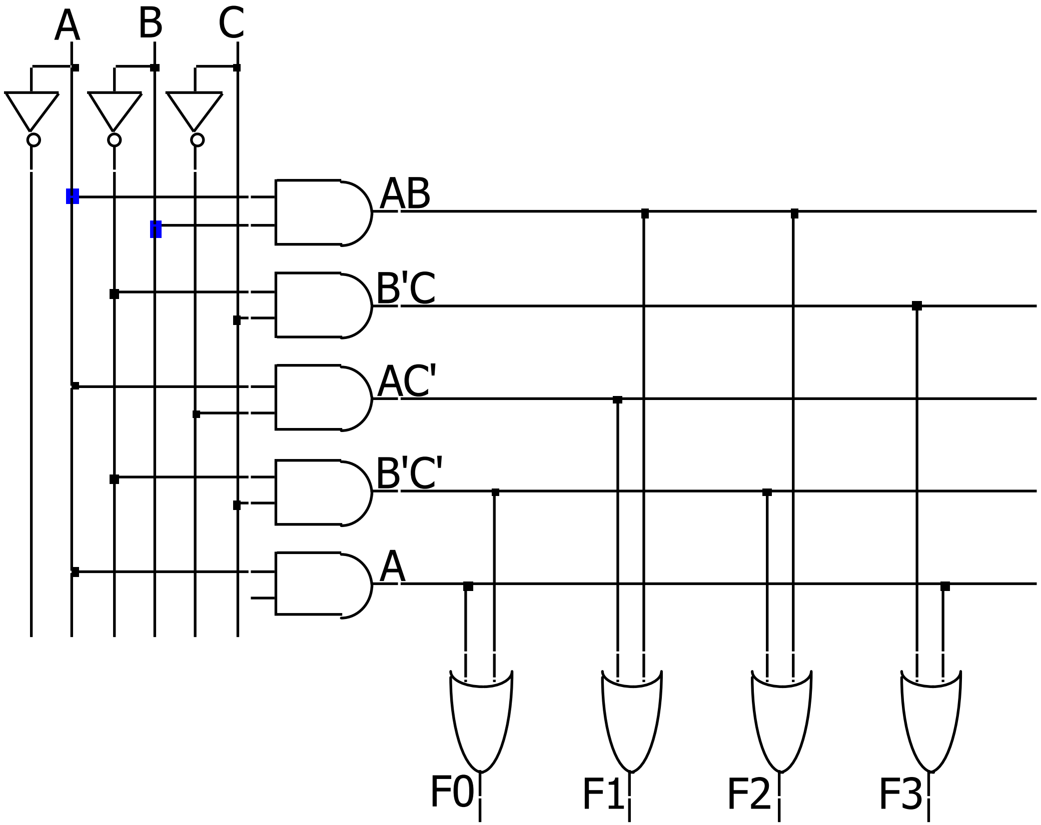
- **All possible connections are available before "programming"**
  - in reality, all AND and OR gates are NANDs



We have to look at two cross-connects: one is between inputs and AND gates and the other is between AND gates and OR gates. We can make or break connections in those two cross-connects.

After programming

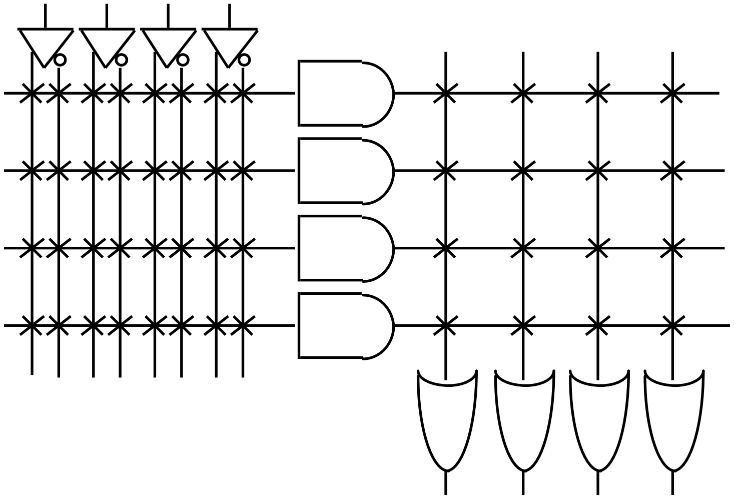
- **Unwanted connections are "blown"**
  - fuse (normally connected, break unwanted ones)
  - anti-fuse (normally disconnected, make wanted connections)



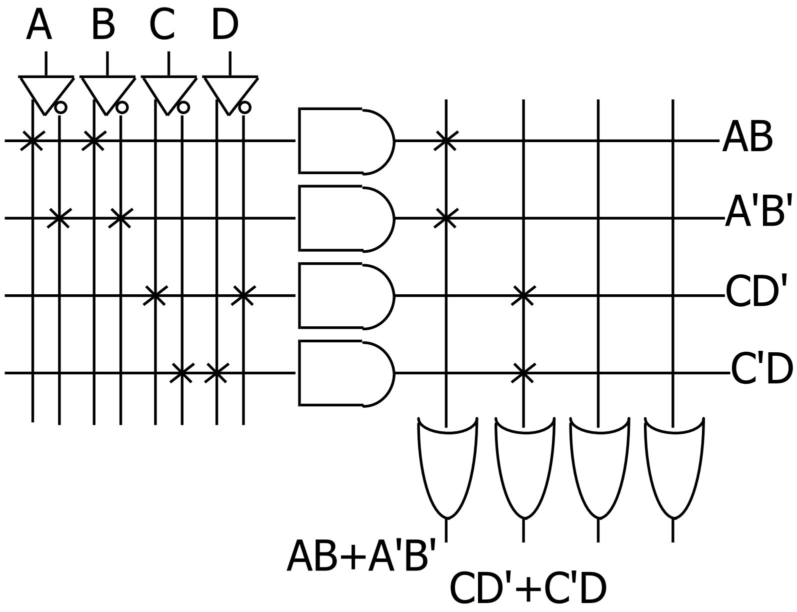
Then we have to do programming, which is the process of enabling or disabling each cross-point. If fuses are used for each cross point, we break unwanted ones. If anti-fuses are used, we enable the wanted ones.

Alternate representation for high fan-in structures

- **Short-hand notation so we don't have to draw all the wires**
  - *x* 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

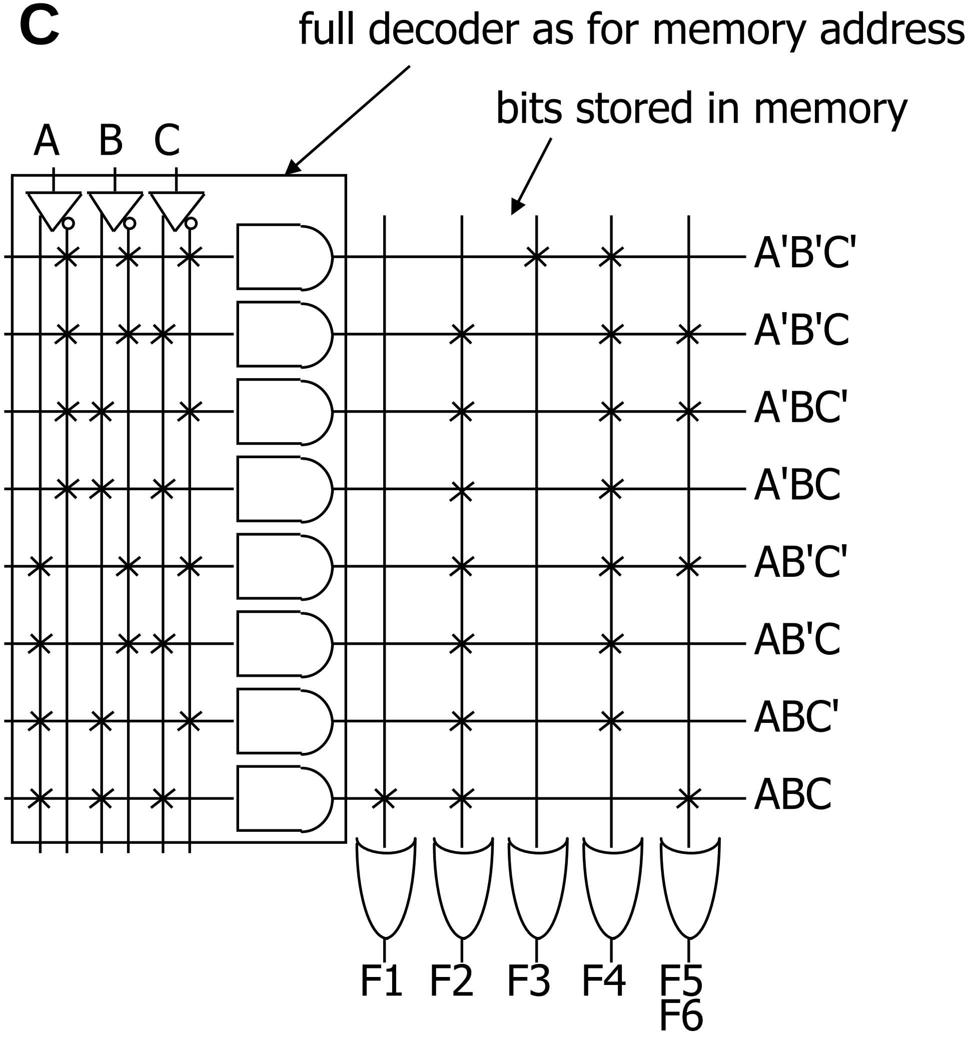


For simplicity, let’s assume fuses for each crosspoint, denoted by x. Before programming, the PLA will look like the one on the left. After programming for the required functions, the PLA will become the one on the right. All the unwanted crosspoints are broken.

PLA 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



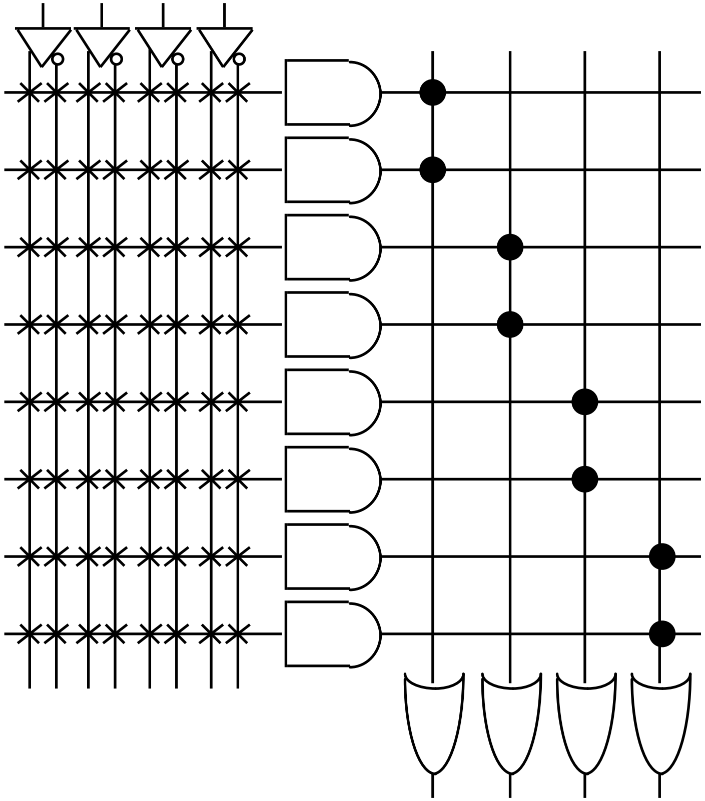
full decoder as for memory address

bits stored in memory

Here are the six functions of three variables.
As there are three variables, total 8 minterms exist. Then all the relevant minterms of each function will be connected to its OR gate.

PALs and PLAs

- **Programmable logic array (PLA)**
  - unconstrained fully-general AND and OR arrays
- **Programmable array logic (PAL)**
  - constrained topology of the OR array
  - faster and smaller OR plane



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

A little bit less programmable but faster version is PAL. In PALs, the cross-connects between AND gates and OR gates are already fixed; the transistor logic is much simpler. What we can control is the cross-connects between inputs and AND gates, denoted by x. So, there is less flexibility in PALs compared to PLAs.

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	–	–	–	–	–	–

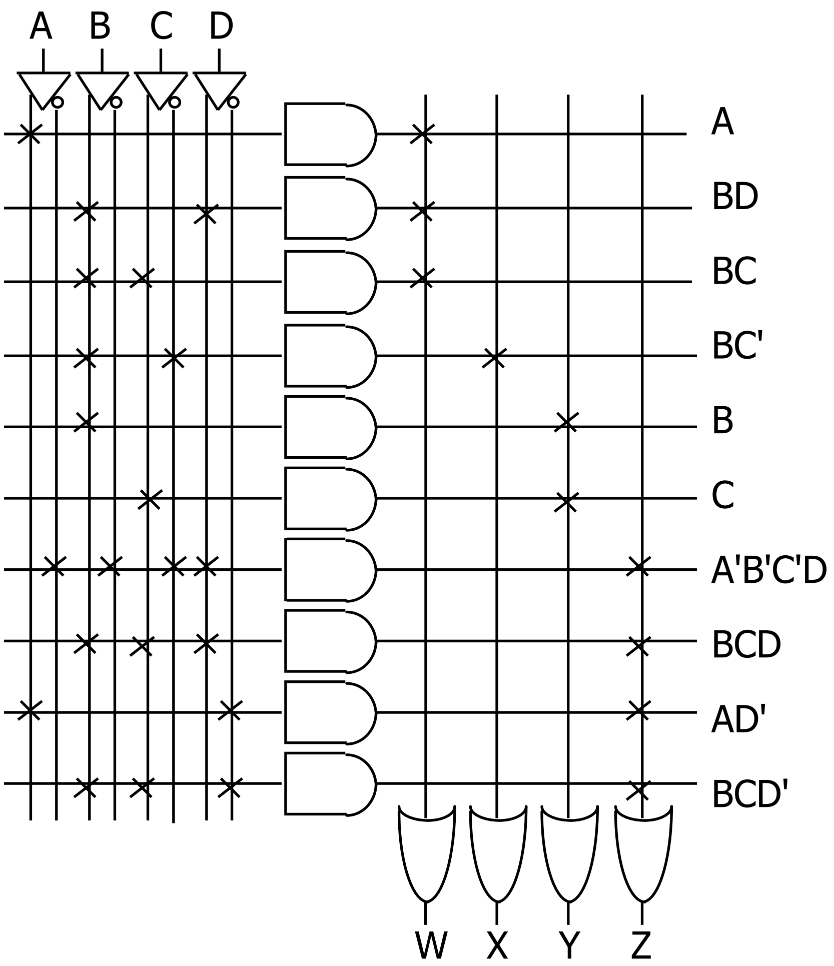
minimized functions:

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

Suppose we have to design a BCD to GRAY code converter. And simplified functions are shown in the above. Unfortunately, there are no common product terms among outputs.

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

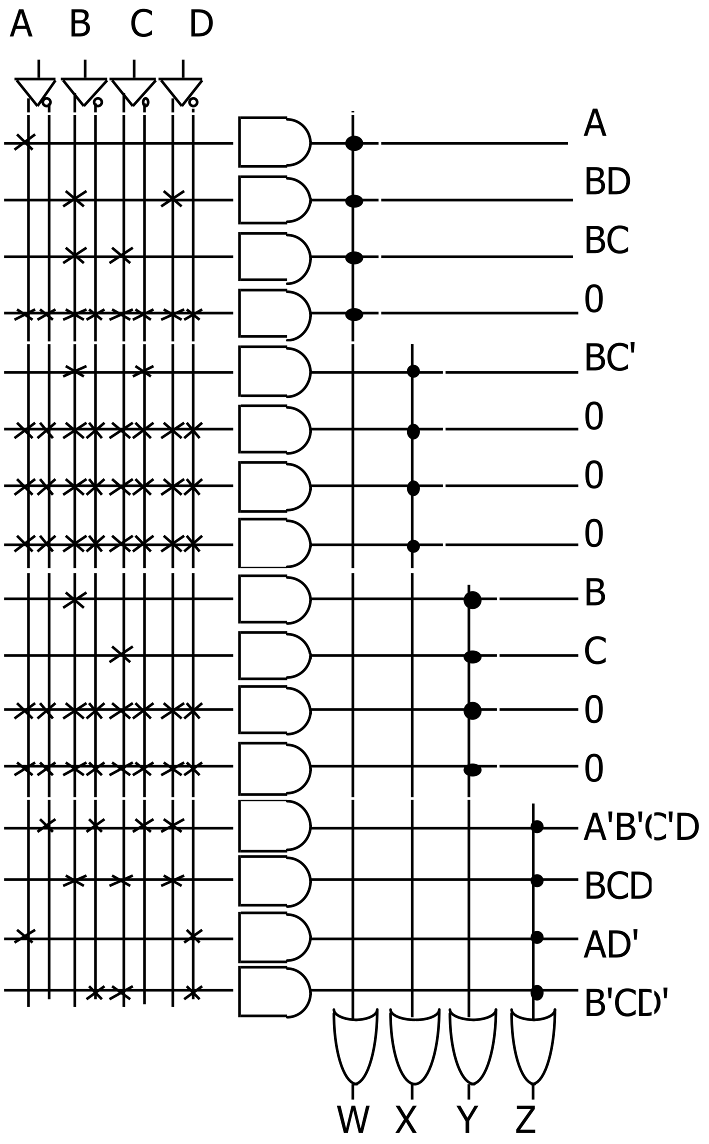
Here, 4 functions are implemented by a single PLA. Total 10 product terms are needed to be represented. In this case, there is no common product term that can be shared by multiple outputs, which means PLA is not an attractive option.

PALs and PLAs: (cont’d)

- **Code converter: programmed PAL**

Could be a limiting factor

4 product terms per each OR gate

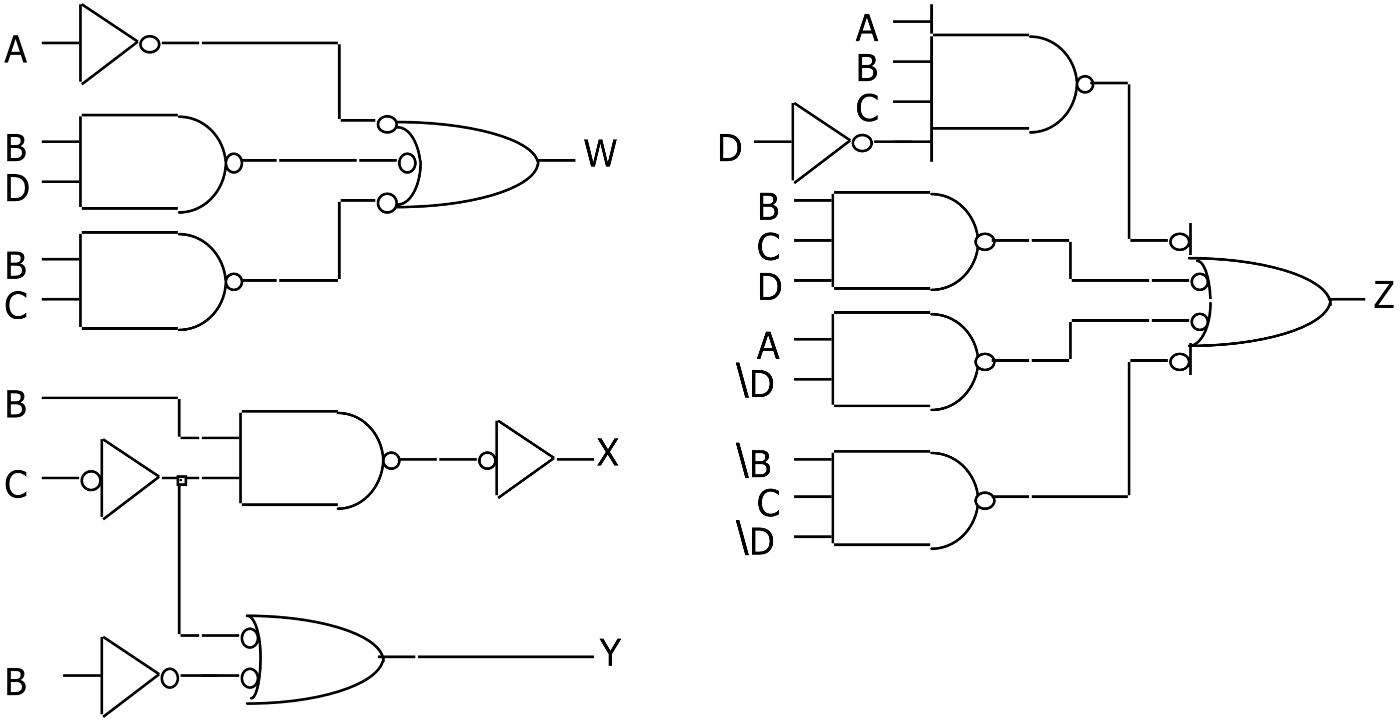


Let’s consider a PAL where exactly 4 AND gates are ORed for each function. Note that the maximum # of product terms of outputs is 4

As there are 4 functions, total 16 AND gates are required. For some functions, they don’t need up to 4 product terms. Then, a FALSE term is connected to surplus AND gates. To make a false term, just leave all the fuses intact.

PALs and PLAs: design example (cont'd)

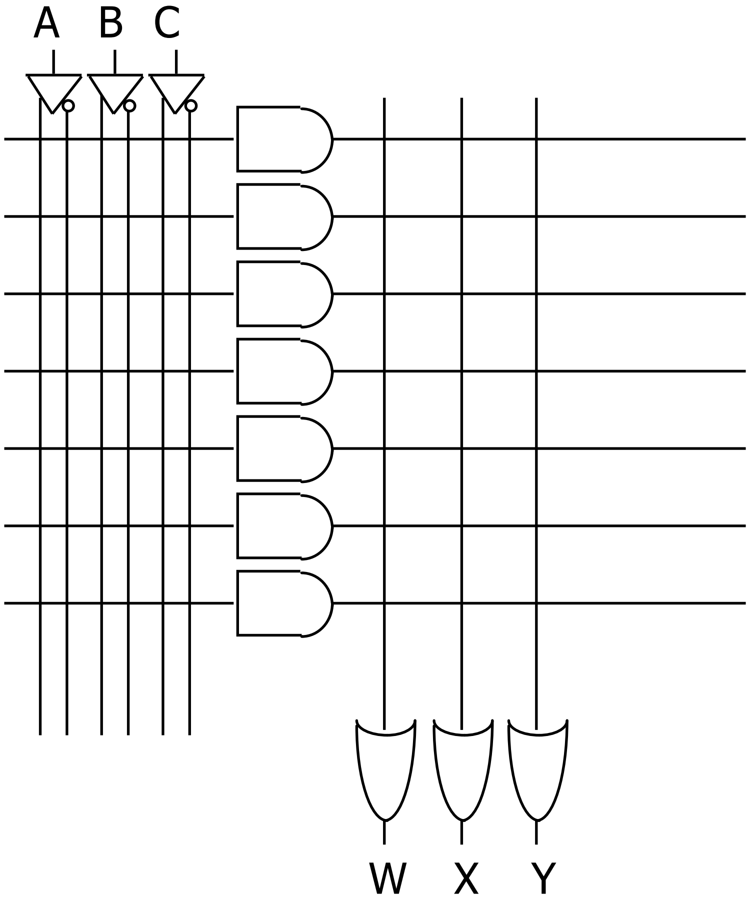
- **Code converter: NAND gate implementation of PAL and PLA**



As we have seen before, an AND-OR combination is easily converted to NAND logic. Just add two bubbles (inverters) in the middle and push them to the opposite directions. So PALs and PLAs are actually implemented by NAND gates.

Activity

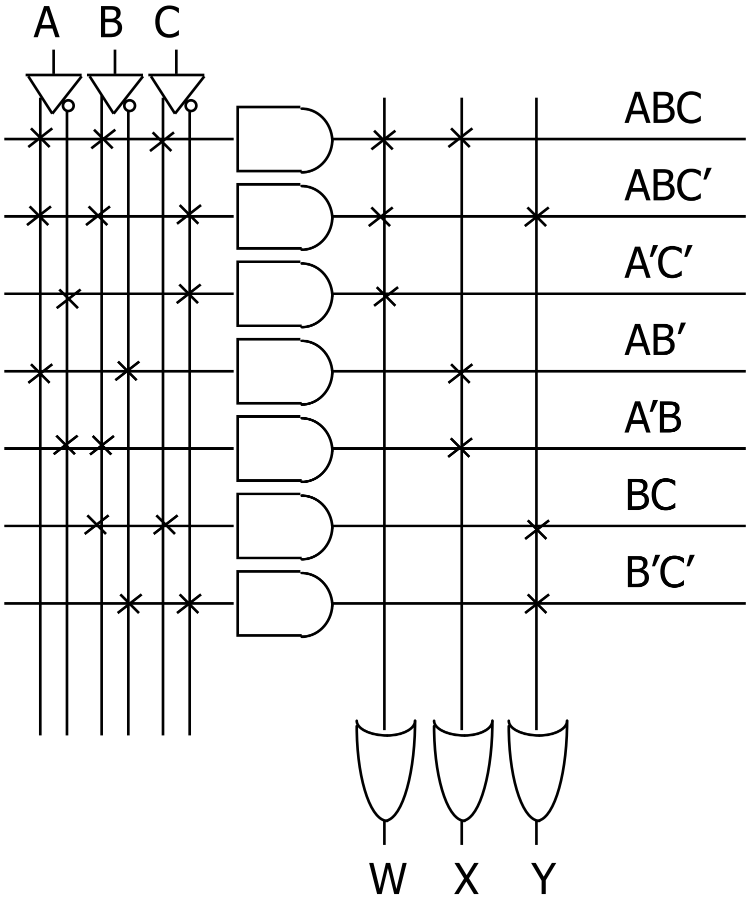
- **Map the following functions to the PLA below:**
  - W = AB + A’C’ + BC’
  - X = ABC + AB’ + A’B
  - Y = ABC’ + BC + B’C’



Activity (cont'd)

Activity (cont'd)

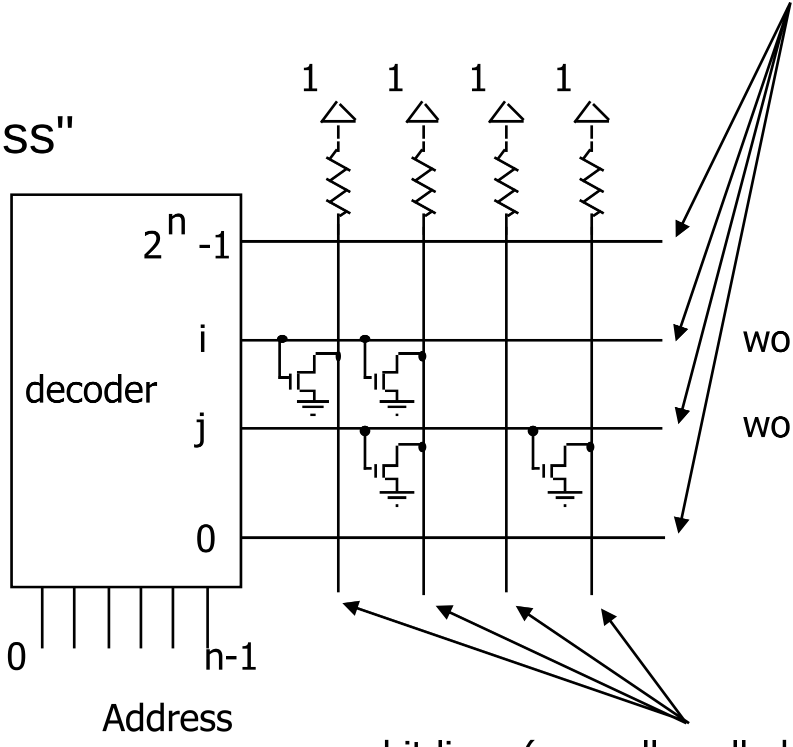
- **9 terms won't fit in a 7 term PLA**
  - can apply concensus theorem to W to simplify to: W = AB + A'C'
- **8 terms wont' fit in a 7 term PLA**
  - observe that AB = ABC + ABC'
  - can rewrite W to reuse terms: W = ABC + ABC' + A'C'
- **Now it fits**
  - W = ABC + ABC' + A'C'
  - X = ABC + AB' + A'B
  - Y = ABC' + BC + B'C'
- **This is called technology mapping**
  - manipulating logic functions so that they can use available resources



Read-only memories (ROMs)

- **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)

Let's look at the simplified structure of a ROM. A ROM is just like a look-up table whose structure is similar to that of a DEMUX. Actually, all the minterms are present in ROMs. Instead of the enable wire, the bits for outputs are programmed. When an address is coming in, its stored data (bits) should be brought up. An address to retrieve each stored word is equal to a minterm in the decoder.

ROMs and combinational logic

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

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

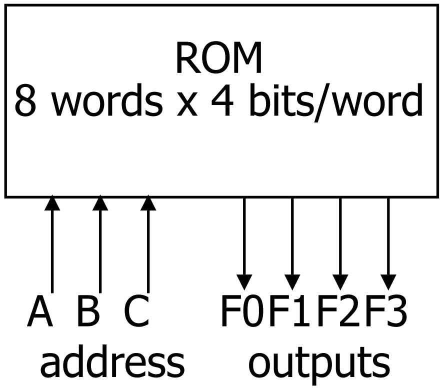
$$F1 = A'\,B'\,C \;+\; A'\,B\,C' \;+\; A\,B\,C$$

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

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

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



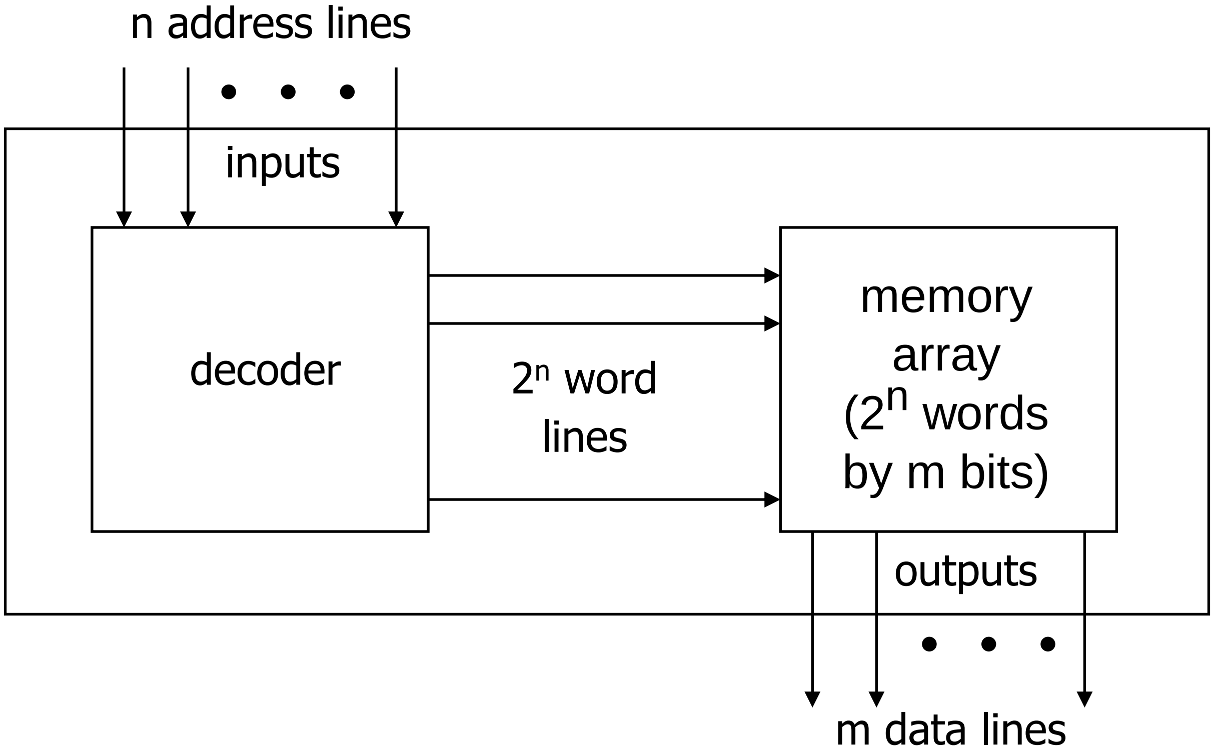
ROM
8 words x 4 bits/word

block diagram

So, a number of functions can be implemented together by using a single ROM. Here, total 32 bits are stored and 4 bits constitute a word. Actually, we don’t need those boolean expressions. We just need to fill in the ROM by the truth table and that’s it.

ROM structure

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



A ROM is similar to a PLA, but for n inputs, there are always 2**n AND gates. Depending on the output values, what we need to do is to control transistors of crosspoints (or bits)

ROM vs. PLA/PAL

- **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

By using ROMs, we can implement a number of functions quickly at the cost of large size (e.g. 2**n AND gates). Also, we cannot utilize DC terms. We can say that if we have to use many minterms, the ROM approach is the best. If there are many shared product terms among outputs, PLA may be good. If the number of product terms for each output is small, PAL may be the best approach

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)

This slide shows a pro-con list of ROM, PAL, PLA technologies. ROMs may be the cheapest due to mass production. In PALs, the OR array is fixed; it takes less time in the OR array. However, no shared product terms is supported in PALs. PLA is the most flexible and expensive option among logic implementation technologies.

Combinational logic technology summary

- **Random (fixed) 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

In chapter 4, we looked at a few programmable structures that facilitate the implementations of two-level logic functions. Each structure has its own pros and cons.