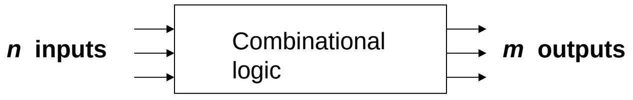
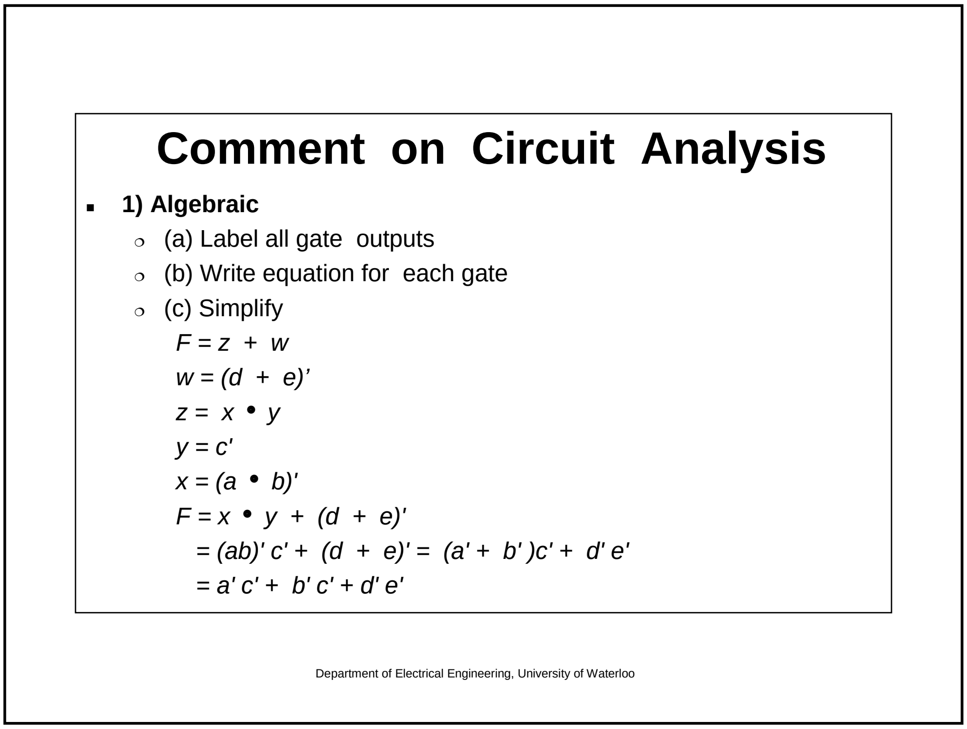
Section 3: Combinational Logic Design

- **Major Topics**
  - Design Procedure
  - Multilevel circuits
  - Design with XOR gates
  - Adders and Subtractors
  - Binary parallel adder
  - Decoders
  - Encoders
  - Multiplexers
  - Programmed Logic Devices

Combinational Logic

- The outputs are functions only of *current* values of the inputs (*no history)*





Comment on Circuit Analysis

- **1) Algebraic**
  - (a) Label all gate outputs
  - (b) Write equation for each gate
  - (c) Simplify

$F = z + w$

$w = (d + e)'$

$z = x \bullet y$

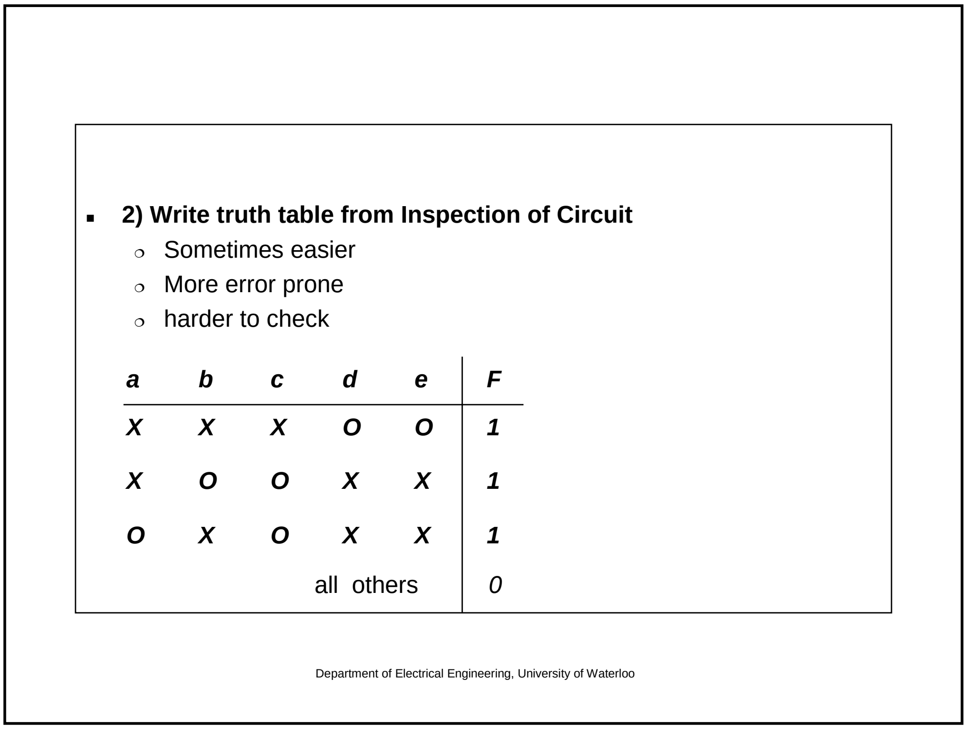
$y = c'$

$x = (a \bullet b)'$

$F = x \bullet y + (d + e)'$

$= (ab)' c' + (d + e)' = (a' + b')c' + d' e'$

$= a' c' + b' c' + d' e'$



- **2) Write truth table from Inspection of Circuit**
  - Sometimes easier
  - More error prone
  - harder to check

a	*b*	*c*	*d*	*e*	*F*
X	*X*	*X*	*O*	*O*	*1*
X	*O*	*O*	*X*	*X*	*1*
O	*X*	*O*	*X*	*X*	*1*
			all others		*0*

Design Procedure

- **Problem stated**
- **Input and Output variables determined**
- **Input and Output variables are assigned names**
- **Truth table developed for all Outputs**
- **A simplified Boolean function for each Output is obtained ****
  - ** constraints - minimum number of gates and Inputs to gate
    - minimum number of IC packages and interconnections
    - propagation times (delay, speed)
    - drive capacity of gates
    - **POWER !**
- **Logic Diagram drawn**
  - <u>Normally assume</u> complements of Inputs are available
  - If <u>not</u>, generate them with inverter

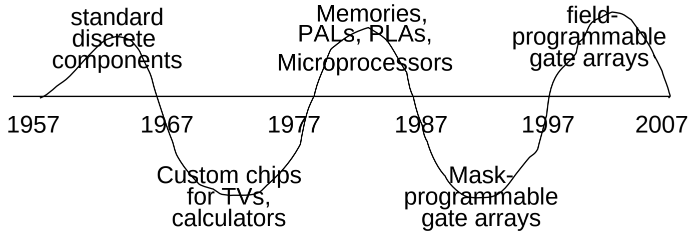
Evolution of Logic Design

- **Till the mid-1960's each gate in a logic circuit was a vacuum tube or transistor and <u>the design goal</u> was very simple**
  - Minimize the number of gates
- **With the development of Integrated Circuits (ICs) <u>two design goals</u> emerged**
  - For the *chip designer*
    - placing a complex function in a limited chip area
    - this requires that the number of gates and interconnections be minimized
  - For the *system designer*
    - minimize the number of IC packages required for the circuit

- **As time progressed the complexity of IC packages available increased**

SSI	**S**mall **S**cale **I**ntegration	≈ 10 gates	(4 NAND gates)
MSI	**M**edium **S**cale **I**ntegration	≈ 10^2 gates	(4 bit adder)
LSI	**L**arge **S**cale **I**ntegration	≈ 10^3 gates	(microprocessors, memory, PLD)
VLSI	**V**ery **L**arge **S**cale **I**ntegration	> 10^4 gates	(complex processors, large memories, gate arrays)

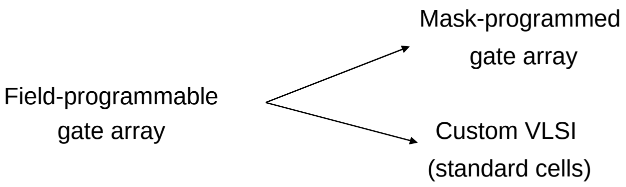
- **Large system logic design has gone in cycles between using standard components and developing customs circuits**
  - Standard Components


standard discrete components
Memories, PALs, PLAs, Microprocessors
field-programmable gate arrays
1957
1967
1977
1987
1997
2007
Custom chips for TVs, calculators
Mask-programmable gate arrays

- Custom Components

  (Adapted from “Makimoto’s Wave”, IEEE Spectrum, Jan 1992)

  - It should be noted that, for a given technology, custom circuits are faster and can provide greater functionality.
  - Typical large system evolution:



Digital Logic Families

- **The most widely used logic families are:**
  - TTL Transistor-transistor logic
    - For many years this was the standard
  - ECL Emitter-coupled logic
    - Used for high speed circuits
  - CMOS Complementary metal-oxide semiconductor
    - Very low power consumption, good speed
    - High packing density
    - Easy fabrication
- **Originally CMOS was slower than TTL, but progress in CMOS (smaller features) improved CMOS speed and it became the dominant technology around 1990.**

- **Typical gate characteristics (two input NAND)**

	Noise Margin (V)	**Fan-out**	**Power (mW)**	**Nominal Delay (ns)**
TTL	0.4	10	10	10
Schottky TTL	0.4	10	20	3
ECL	0.15	25	25	0.05
CMOS	0.5	4	0.001 (100 Mhz)	0.1

	MSI	**PLDs**	**Programmable Gate Arrays**	**Gate Arrays**	**Custom VLSI**
Integration in Gates	100	100-2k	1k-10M	1k-100k	1k-100M
Speed	Fast	Slow to Medium	Slow to Medium	Slow to Fast	Fast
Function defined by User	No	Yes	Yes	Yes	Yes
Time to Customize	-	Seconds	Seconds	Weeks	Months
User Programmable	No	Yes	Yes	No	No

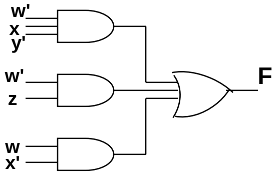
Design of Large Systems

- **FACTS:**
  - If inverters are ignored, minimum number of gates is normally given by a two level OR-AND or AND-OR circuits (single function)
  - Two level circuits have minimum delay (unless the number of inputs is large)
- **However:**
  - Cannot ignore inverters
  - Often generating multiple functions – Logic resuse
  - Two level circuits often require that gates have an excessive number of inputs
  - There are tricks in NAND and NOR circuits to eliminate inverters

Multi-Level Circuit Design

- **(1) Try to reduce the number of inputs to the gates by factoring**
  - Consider:

yz wx	00	01	11	10
00		1	1	
01	1	1	1	
11				
10	1	1	1	1



F = w'xy + w'z + wx'

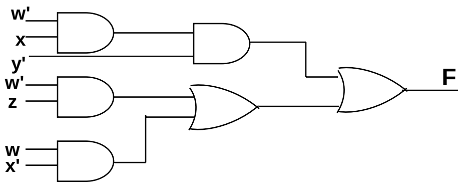
2 2-input gates

2 3-input gates

3 inverters

7 gates with 13 inputs

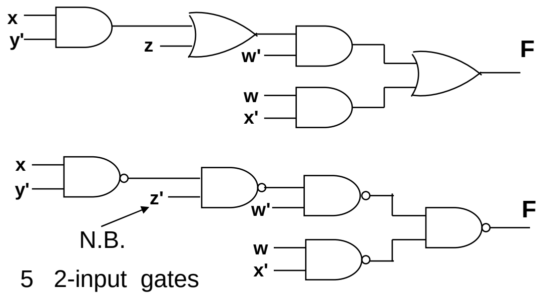
- If only two-input gates are permissible, splitting the three input gates yields:



Multi-Level NAND Circuit: Design and Analysis

- Given Boolean expression, draw the schematic with AND, OR and Inverter gates
- Convert all AND gates to NAND gates with AND-Invert symbols
- Convert all OR gates to NAND gates with Invert-OR symbols
- Check all inversion symbols (small circles) along signal paths, if needed add an inverter (1-input NAND gate)

F = w’(xy’ + z) + wx'



N.B.

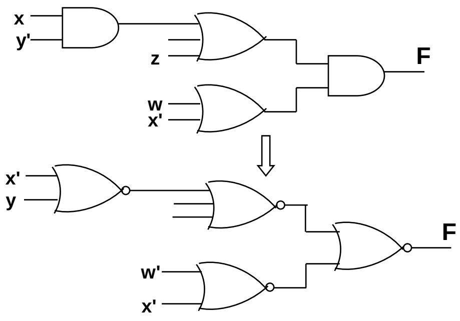
5 2-input gates

5 inverters

Multi-Level NOR Circuit: Design and Analysis

- **Given the algebraic expression, draw the AND-OR logic diagram**
- **Convert all OR gates to NOR gates with ON-Invert symbols**
- **Convert all AND gates to NOR gates with Invert-AND symbols**
- **Any inversion (small circle) that is not compensated by another small circle, needs an inversion (10input NOR gate)**

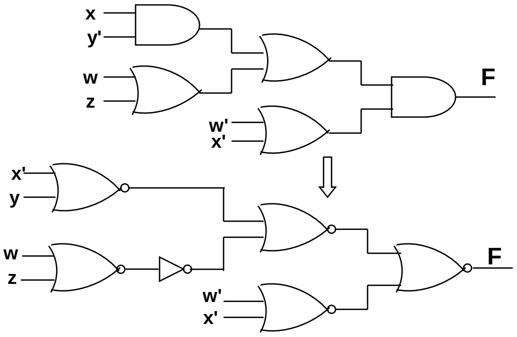
$$F = (w' + x')(w + x + z)(w + y' + z)$$
$$= (w' + x')(w + z + xy')$$



3 2- input NORs
1 3- input NORs
2 inverters

6 gates

- **If only two-input gates are available some gates must be split**



5 input NOR
3 inverters

8 gates

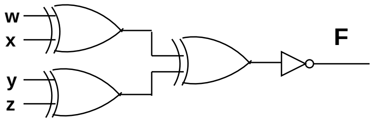
Circuit Design Using XOR Gates

- **Example**
  - Looks nasty!

    $F = (w \oplus x \oplus y \oplus z)'$

    $= (w \oplus x \oplus y \oplus z)'$

wx \ yz	00	01	11	10
00	1		1	
01		1		1
11	1		1	
10		1		1



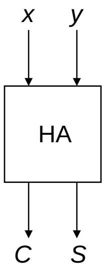
Adders and Subtractors

- **Adders and subtractors are important components in many logic circiuts**
- **3.6.1 Half- Adder**
  - $(CS)_2$ = x plus y

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = x'y + xy' = x \oplus y$

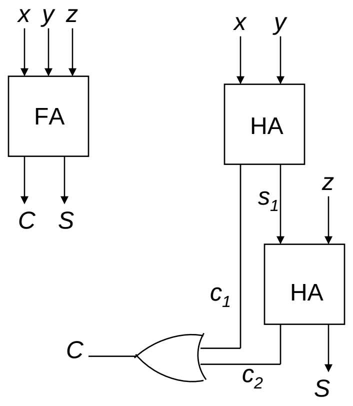
$C = xy$



Full - Adder

$(CS)_2$ = x plus y plus z

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



x \ yz	00	01	11	10
0		1		1
1	1		1	

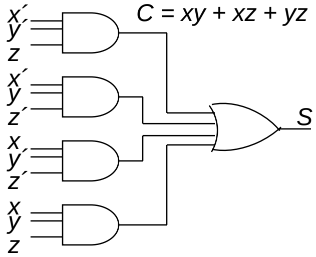
S

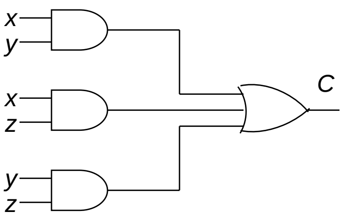
x \ yz	00	01	11	10
0			1	
1		1	1	1

C

$S = x'y'z + x'yz' + xy'z' + xyz = x \oplus y \oplus z$

$C = xy + xz + yz$





Note: The circuit inputs are symmetrical in *x*, *y* and *z*

Half - Subtractor

- **Generate *x - y.***
- **Let**
  - ○ *D* - Difference
  - ○ *B* - Borrow

x	*y*	*B*	*D*
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

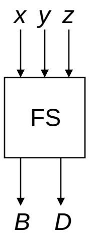
$D = x'y + x\,y' = S$ of Half- Adder

$B = x'y$

Full - Subtractor

- ***(x - y) -z* where (*z* represents a borrow)**

x	*y*	*z*	*B*	*D*
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



$D = x'y'z + x'yz' + x\,y'y' + xyz$

$= S$ of Full- Adder

$= x \oplus y \oplus z$

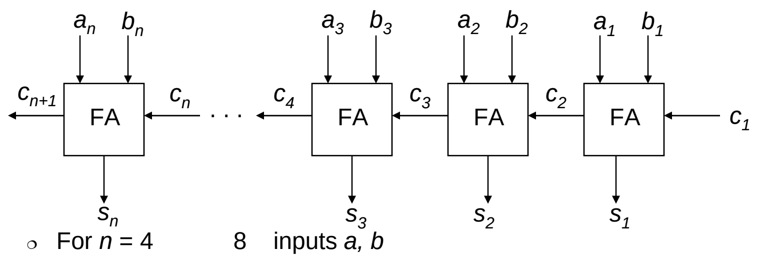
$B = x'y + x'z + yz$

(same as *C* of Full - Adder except *x* is inverted)

Binary Parallel Adder

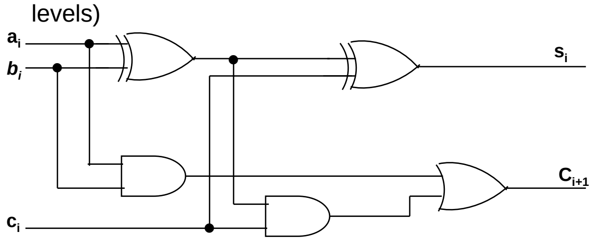
- **Required:** **Add two *n* - bit numbers plus carry**
- **(1) Classical Approach**
  - 2*n* + 1 inputs, *n* + 1 outputs
    Design a (*n* + 1) output, 2 level design
  - Problem:
    - Too many gates
    - Fan-in too large
    - Not practical for *n* > 3

- **(2) Use iterative circuit; reduces gate count and fan-in**



- For *n* = 4
  8 inputs *a, b*
  1 inputs c_1
  4 outputs s_i
  1 output c_5
  2 V_{CC} , ground
  16 pin package

- Problem:
    - Although fewer gates, slower than 2-level circuit because of carry propagation
    - Propagation delay = (average delay of gate) x (no. of gate levels)



    - 2 gate delays per Full - Adder for carry
    - For an *n* - bit adder:
      delay = 2*n* x gate delay

- Compare with the classical (and impractical) 2-level method:
  delay = 2 x gate delay

- To make faster
    - Faster gates
        - expense
        - heat
    - More complexity but less delay
    - Most common approach
        - Carry Look ahead

Carry Lookahead Logic

- **Define:**
  - Carry Propagate
    $P_i = a_i \oplus b_i$
  - Carry Generate
    $G_i = a_i\, b_i$
  - Now
    $s_i = a_i \oplus b_i \oplus c_i$
    $\quad = P_i \oplus c_i$
    $c_{i+1} = a_i\, b_i + a_i\, c_i + b_i\, c_i$
    $\quad = a_i\, b_i + a_i\, b_i'c_i + a_i'b_i\, c_i$
    $\quad = G_i + c_i\, P_i$

- **Observe:**
  - All carries can be generated simultaneously
    $c_2 = G_1 + P_1c_1$
    $c_3 = G_2 + P_2c_2 = G_2 + P_2G_1 + P_2P_1c_1$
    $c_4 = G_3 + P_3c_3 = G_3 + P_3G_2 + P_3P_2\, G_1 + P_3\, P_2P_1c_1$
- **Delay?**
  - P_i , G_i
    XOR and AND 2 gate delays
  - c_i →
    two-level AND - OR 2 gate delays
  - s_i →
    XOR 2 gate delays
  - 6 gate delays independent of n
  - n limited by connections and gate loading

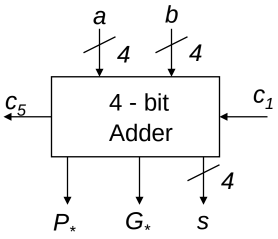
- **Note:**

$$\begin{aligned} c_5 &= G_4 + P_4 c_4 \\ &= G_* + P_* c_1 \end{aligned}$$

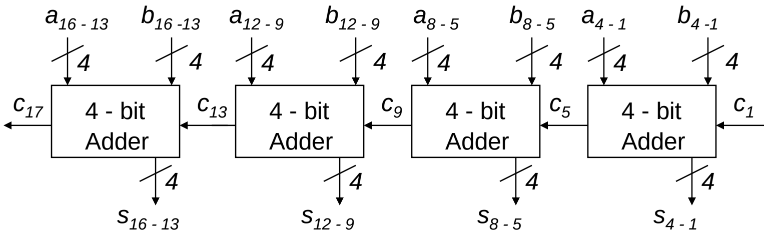
  - *Where*

$$G_* = G_4 + P_4 G_3 + P_4 P_3 G_2 + P_4 P_3 P_2 G_1$$
$$P_* = P_4 P_3 P_2 P_1$$

  - Typical Adder chip

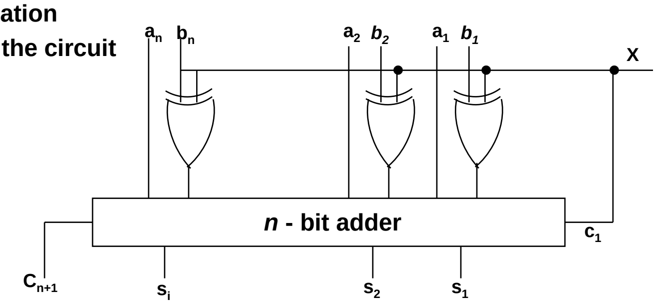


Ripple Carry Between Chips



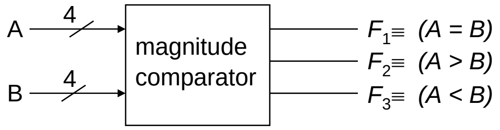
Twos Complement Adder/Subtractor

- **Let $a_n \ldots a_1$ and $b_n \ldots b_1$ be binary numbers in twos complement representation**
- **Consider the circuit**



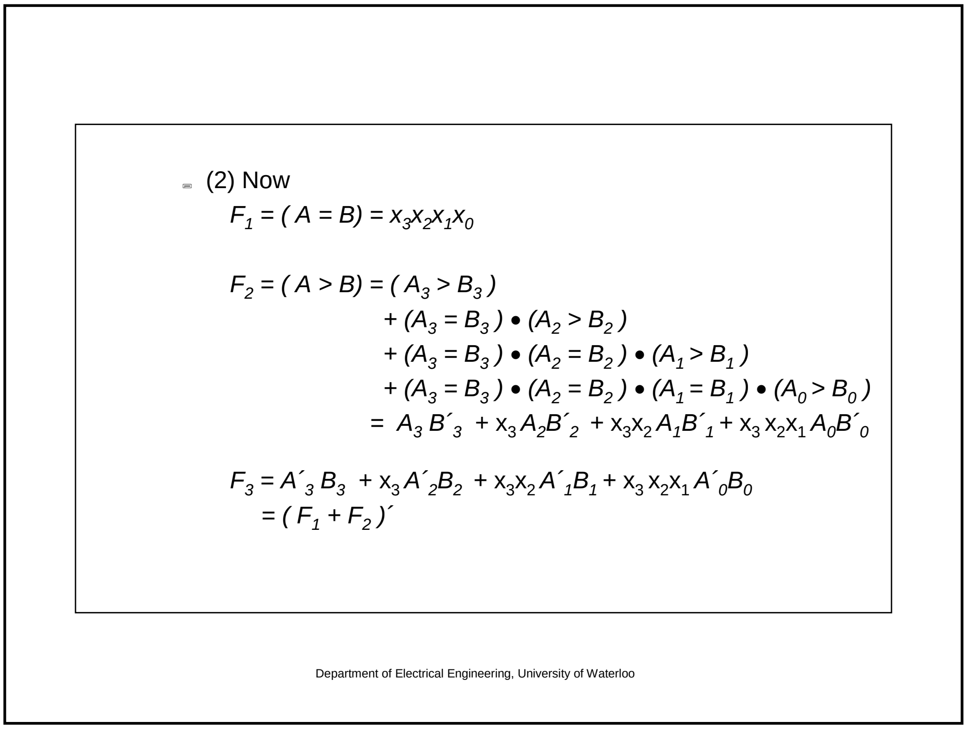
  - If $X = 0$ the circuit adds $a_n \ldots a_1$ and $b_n \ldots b_1$
  - If $X = 0$ the circuit subtracts $b_n \ldots b_1$ from $a_n \ldots a_1$
  - The same circuit will also add/subtract unsigned binary numbers

Magnitude Comparator



- **Classical Approach**
  - 3 outputs, 2*n* inputs
  - Almost impossible if *n* > 3
- **Approach**
  - Do in two steps
    - (1) Define

$$x_i \equiv (A_i \odot B_i) = A_i B_i + A_i' B_i' \text{ , } 0 \le i < 3$$

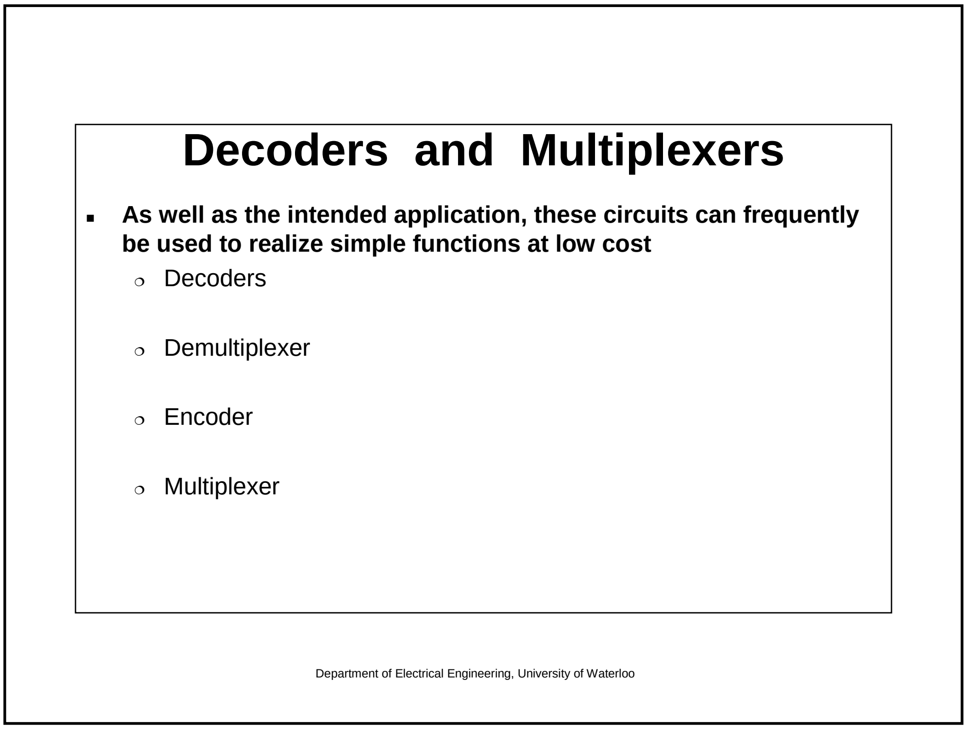


- (2) Now

$$F_1 = (A = B) = x_3x_2x_1x_0$$

$$\begin{aligned} F_2 = (A > B) &= (A_3 > B_3) \\ &+ (A_3 = B_3) \bullet (A_2 > B_2) \\ &+ (A_3 = B_3) \bullet (A_2 = B_2) \bullet (A_1 > B_1) \\ &+ (A_3 = B_3) \bullet (A_2 = B_2) \bullet (A_1 = B_1) \bullet (A_0 > B_0) \\ &= A_3 B'_3 + x_3 A_2 B'_2 + x_3 x_2 A_1 B'_1 + x_3 x_2 x_1 A_0 B'_0 \end{aligned}$$

$$\begin{aligned} F_3 &= A'_3 B_3 + x_3 A'_2 B_2 + x_3 x_2 A'_1 B_1 + x_3 x_2 x_1 A'_0 B_0 \\ &= (F_1 + F_2)' \end{aligned}$$



Decoders and Multiplexers

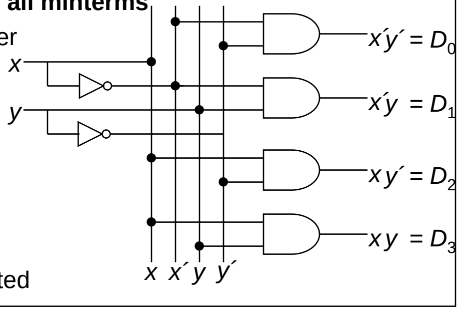
- **As well as the intended application, these circuits can frequently be used to realize simple functions at low cost**
  - Decoders
  - Demultiplexer
  - Encoder
  - Multiplexer

Decoders

- **Code of *n* bits can represent 2^n elements**
- ***n* - to - *m* line decoder converts an *n* bit input into *m* distinct outputs**
- **If $m = 2^n$, decoder produces all minterms**
  - Example: 2 - to - 4 decoder

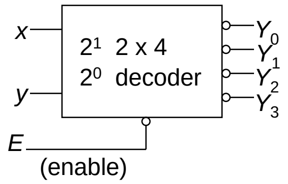
Truth Table

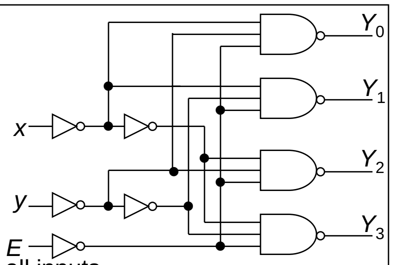
x	y	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



Note: Only one output asserted

- **Decoder with enable line**

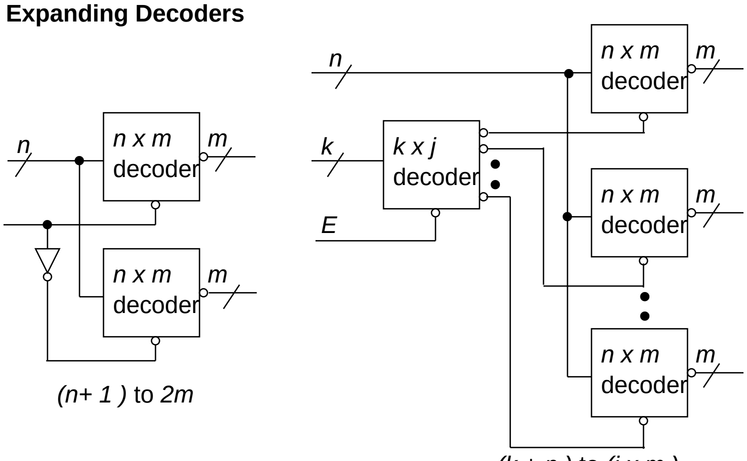




  - Note inverters
    - Uniform load of one gate on all inputs
- **Truth Table**

E	*x*	*y*	Y_0	Y_1	Y_2	Y_3
1	*X*	*X*	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

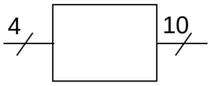
- **Expanding Decoders**

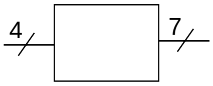


(n+ 1) to *2m*

(k + n) to *(j x m)*

- **Common Decoders (other than *n* -to- 2*ⁿ*)**
  - BCD to decimal



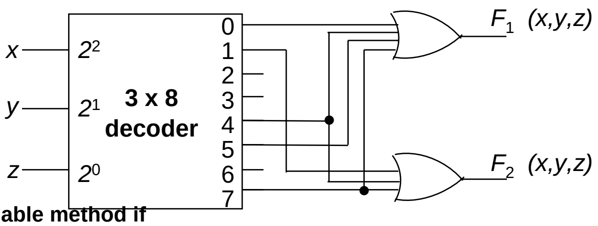


  - BCD to 7 segment
    - Normally organized so “don’t care” conditions are *0* outputs

- **Applications**
  - Address decoding (in components etc.)
  - Sequential circuits (state decoding)
  - Boolean function implementation

Function Implementation Using Decoders

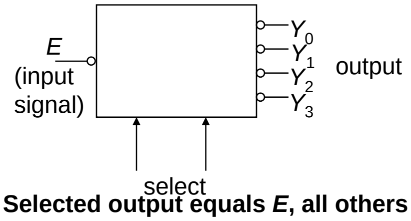
- **Example :** $F_1 = \sum(0,4,5,7)$ $F_2 = \sum(1,4,7)$



- **Reasonable method if**
  - many outputs
  - each output has only a few minterms
- **If any function requires more than half of the minterms generate *F´* and use a NOR gate**

Demultiplexer

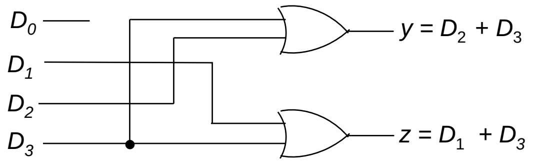
- **Put input on one of *m* output lines, according to values on select lines**
- **Just decoder with lines renamed**



- **Selected output equals *E*, all others *1***

Encoders

- **Reverse operation to decoder**
- **Assumes only one input line active**
- **Example : 4 lines to 2 (binary)**



binary number output is $(yz)_2$

Priority Encoders

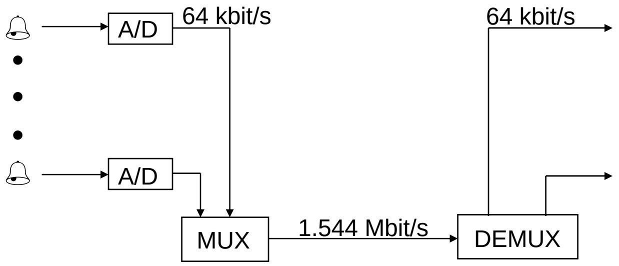
- **More than 1 active input line**
- **Output corresponds to input line with highest subscript**

D_0	D_1	D_2	D_3	x	y	z
X	X	X	1	1	1	1
X	X	1	0	1	1	0
X	1	0	0	1	0	1
1	0	0	0	1	0	0
0	0	0	0	0	0	0

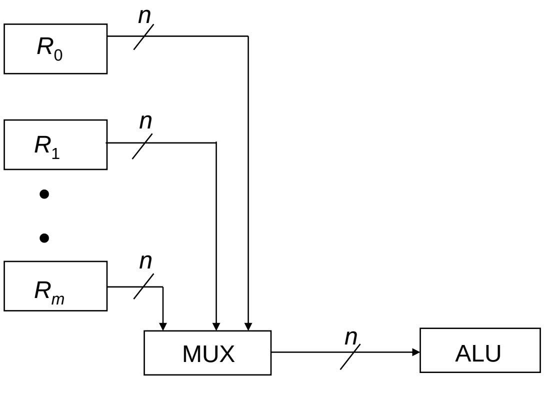
$x = D_0 + D_1 + D_2 + D_3$
$y = D_2 + D_3$
$z = D_3 + D_1D'_5$

Multiplexers

- **Multiplexing**
  - Concentrating information from a large number of lines onto a smaller number of lines
    - Example: **Telecommunications**

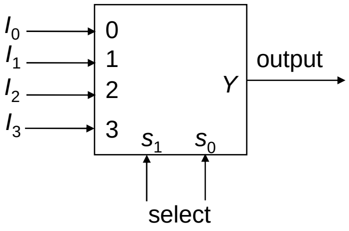


- Example: **Computers**

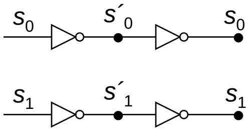


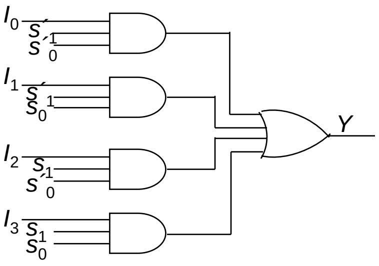
- **Key Component: Multiplexer (MUX)**
  - Example: 4 x 1 MUX

s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3



- Implementation : Obvious

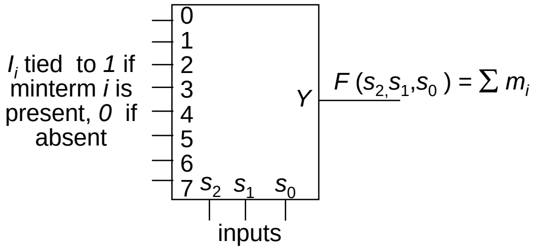




- **Note:** Looks like a decoder with extra input line to AND gate, and AND outputs Ored
- Usually, also on "enable" (or "strobe") input for expanding multiplexers.

Applications

- **1) Multiplexing Applications**
- **2) Sequential Circuits**
- **3) Boolean Function Implementation**
  - a) naïve approach, *n* select lines for function of *n* variables



I_i tied to *1* if minterm *i* is present, *0* if absent

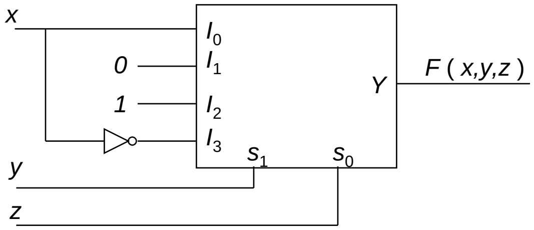
$F(s_2, s_1, s_0) = \sum m_i$

⟹ two level AND-OR circuit

- b) Better approach, (*n* - 1) select lines for functions of *n* variables
  - **How**: (*n* - 1) variables go to select lines, n^{th} variables, complement, *0* or *1* goes to MUX inputs
  - Example: $F(x,y,z) = \sum(2,3,4,6)$

select:	*yz*	*00*	*01*	*10*	*11*	
output:		I_0	I_1	I_2	I_3	
	x = 0	m_0	m_1	m_2	m_3	desired output
	1	m_4	m_5	m_6	m_7	
inputs:	(I_i)					

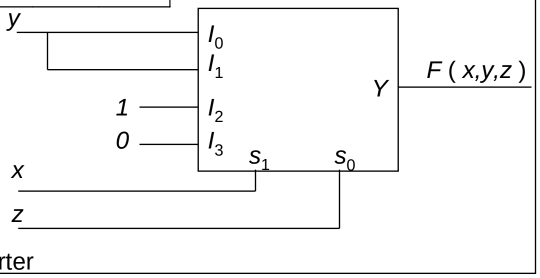
When $yz = 00$ Then $F = x$
$yz = 01$ $F = 0$
$yz = 10$ $F = 1$
$yz = 11$ $F = x'$



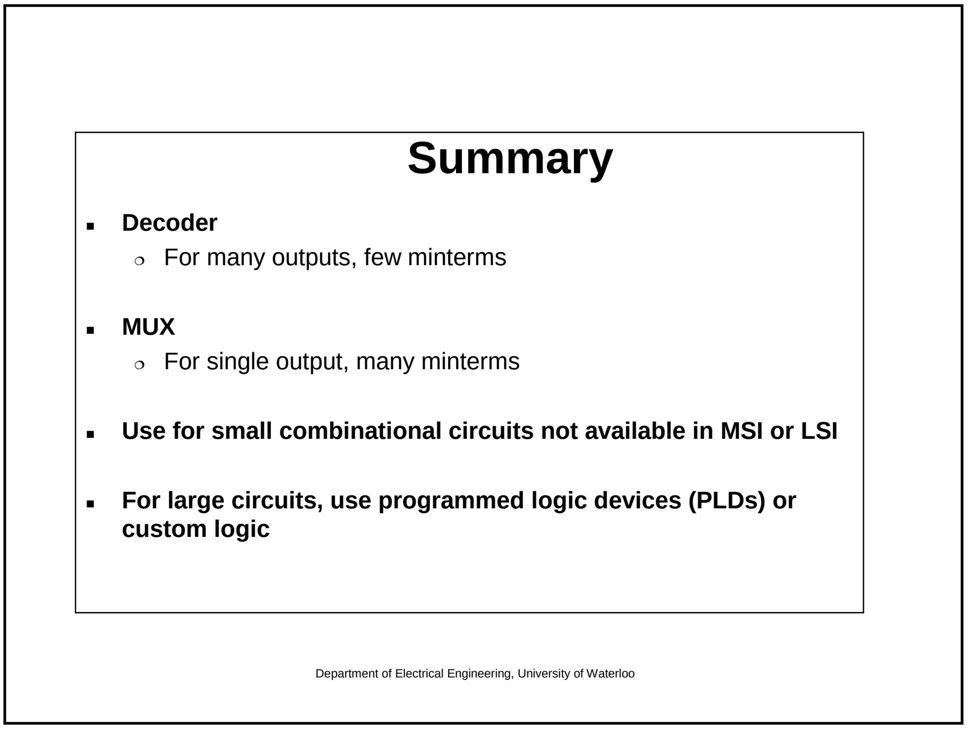
- **Note:** Should try several selections for MUX input

$F (x,y,z) = \sum (2,3,4,6)$

select:	xz	00	01	10	11
output:		I_0	I_1	I_2	I_3
	$y = 0$	m_0	m_1	m_2	m_3
	1	m_4	m_5	m_6	m_7
inputs:	(I_i)				



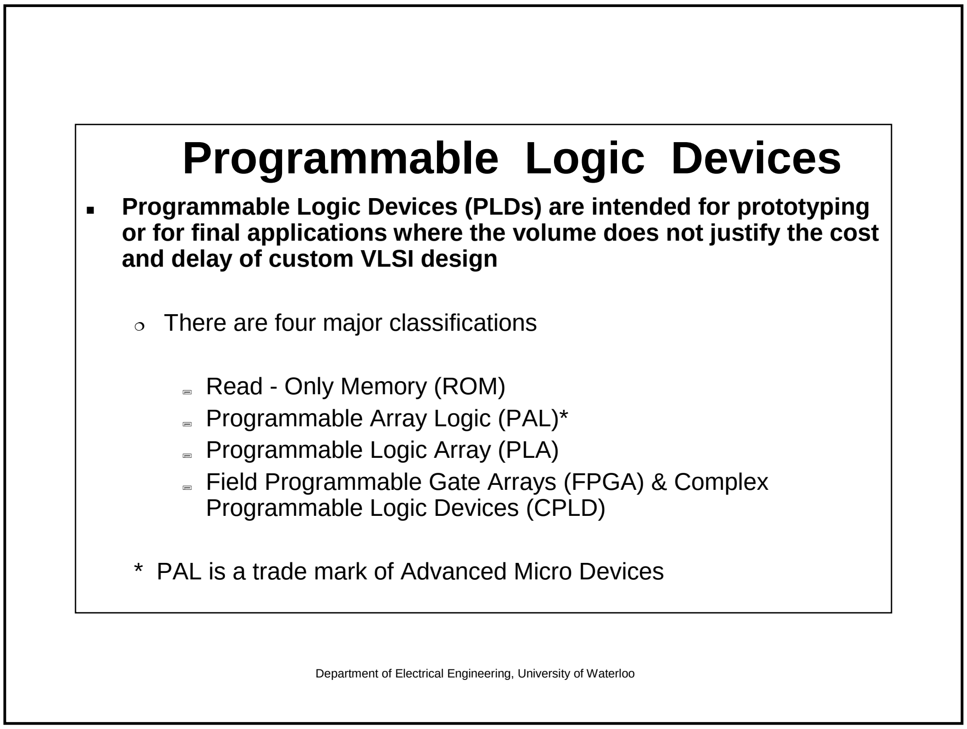
- saved an inverter



Summary

- **Decoder**
  - For many outputs, few minterms
- **MUX**
  - For single output, many minterms
- **Use for small combinational circuits not available in MSI or LSI**
- **For large circuits, use programmed logic devices (PLDs) or custom logic**

Department of Electrical Engineering, University of Waterloo



Programmable Logic Devices

- **Programmable Logic Devices (PLDs) are intended for prototyping or for final applications where the volume does not justify the cost and delay of custom VLSI design**
  - There are four major classifications
    - Read - Only Memory (ROM)
    - Programmable Array Logic (PAL)*
    - Programmable Logic Array (PLA)
    - Field Programmable Gate Arrays (FPGA) & Complex Programmable Logic Devices (CPLD)

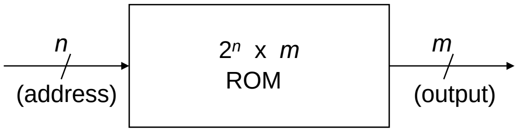
* PAL is a trade mark of Advanced Micro Devices

Department of Electrical Engineering, University of Waterloo

- All four come in many forms. Some key distinctions:
  - **Mask Programming**
    - The logic design is fixed during the last few steps of manufacturing
  - **Programmable**
    - With appropriate hardware the logic gates are “programmed” to have the desired configuration
  - **Erasable**
    - The pattern that was programmed can be erased
    - Some devices are erased by ultraviolet light, others are erased electrically

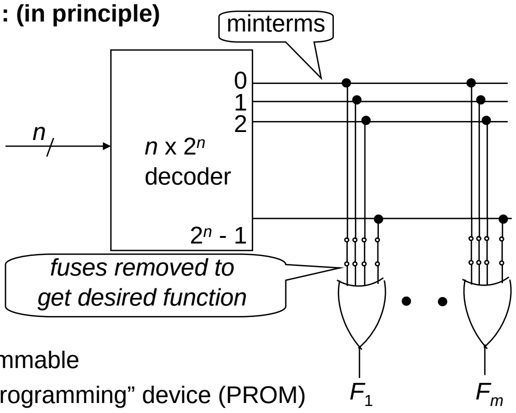
Read - Only Memory (ROM)

- **Use input variables as the address to a memory location**
  - Memory contents are function values



  - Can use to generate *m* functions of *n* variables

- **Internal Construction : (in principle)**



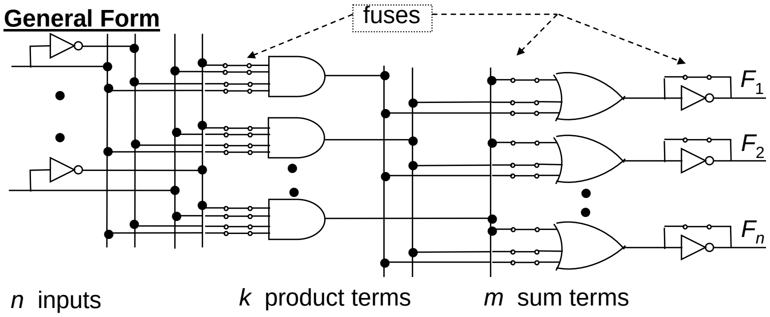
  - “Fuses” are
    - a) Mask Programmable
    - b) “Blown” by “programming” device (PROM)
      - Some PROM can be erased (EPROM or EEPROM)
  - Good for generating several functions of many variables
  - Good for prototype
  - Inefficient use of space if there are many “don’t care” cases

Types of ROM

- **Several technologies for ROM implementation**
- **ROM – Customized in manufacturing, one time programmable**
- **PROM (Programmable Read Only Memory)**
  - PROM contains fuses giving logic 1 or 0 to a particular bit. User blows the fuse for programming
  - One time programmable
- **EPROM (Electrically Programmable Read Only Memory)**
  - Can be “erased” by exposure to UV light; otherwise same as a PROM
  - Multiple time programmable
- **EEPROM (Electrically Erasable Programmable Read Only Memory)**
  - Can be “erased electrically” otherwise same as a PROM
  - Multiple time programmable
- **Extra pins in these devices for programming data (bit stream) is applied**

Programmable Logic Array (PLA)

- **Same idea as ROM except “don’t cares” can be eliminated**
- **Generalized AND-OR and AND-OR-INVERT which is mask or field programmed by removing unwanted fuses**
- **General Form**



- **One set of fuses determines variables input into AND gates**
- **Second set of fuses determines product terms input into OR gates**
- **Third set of fuses selects AND-OR or AND-OR-INVERT realization**
  - not all PLAs have the AND-OR-INVERT choice

- **Example:**
  - Note: Unrealistically small

    $F_1 = \sum (0,4,5,7,9)$ $F_2 = \sum (0,1,2,8,10,11,12,13,14,15)$

wx \ yz	00	01	11	10
00	1			
01	1	1	1	
11				
10		1		

F_1

wx \ yz	00	01	11	10
00	1	1		1
01				
11	1	1	1	1
10	1		1	1

F_2

wx \ yz	00	01	11	10
00	1			
01	1	1	1	
11				
10		1		

F'_1

wx \ yz	00	01	11	10
00	1	1		1
01				
11	1	1	1	1
10	1		1	1

F'_2

Department of Electrical Engineering, University of Waterloo

- $F_1 = w'y'z' + w'xz + wx'y'z$ $F_2 = w'x'y' + wx + wy + x'z'$

  $F'_1 = w'x'z' + yz' + wx + wz' + wy$ $F'_2 = w'x + wx'y'z$

  $or = w'x'z' + yz' + wx + wz' + x'y$
- **Select whichever of**
  - $F_1 F_2$, $F_1 F'_2$, $F'_1 F_2$ or $F'_1 F'_2$

    minimizes the number of product terms

    (Number of product terms is the limiting factor)
- **PLA specification**

	product term	**inputs**				**outputs**	
		w	***x***	***y***	***z***	F_1	F_2
w´y´z´	1	0	-	0	0	1	-
w´xz	2	0	1	-	1	1	-
wx´y´z	3	1	0	0	1	1	1
w´x	4	0	1	-	-	-	1
w´yz	5	0	-	1	1	-	1
	true / complement →					T	C

Department of Electrical Engineering, University of Waterloo

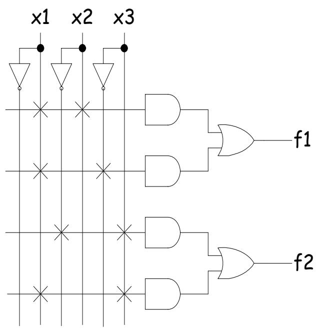
Programmed Array Logic Devices

- **A Programmed Array Logic (PAL)* device is an alternative to a PLA**
  - Only one level of programming
    - Programmable AND
    - Fixed OR
    - Opposite of ROM
  - Easier to program but less flexible
  - Less expensive to manufacture and somewhat faster due to only having one level of configurable logic
  - Many PALs have some bi-directional input/output pins
  - Many PALs have flip-flops

*** PAL is a trademark of Advanced Micro Devices**

Programmable Array Logic (PAL)

- **Example of a PAL:**

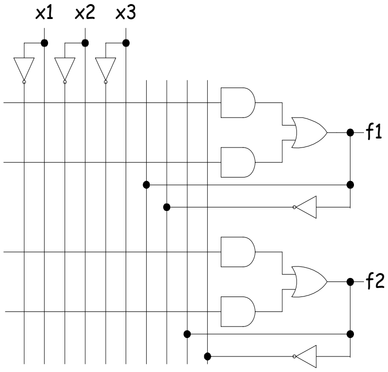


$$f_1 = x_1x_2 + x_1\overline{x}_3$$

$$f_2 = \overline{x}_2x_3 + x_1x_3$$

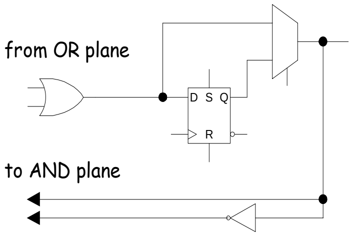
Programmable Array Logic (PAL)

- Sometimes the outputs are fed back internally and can be used to create product terms.



Simple Programmable Logic Device (SPLD)

- To implement sequential circuits, take a PAL and add some flip-flops at the output of the OR plane.
- For example…



- Above circuit (plus SOP from the AND plane and OR gate) form a MacroCell.
- Several MacroCells together in the same IC is called an SPLD.

Complex Programmable Logic Device (CPLD)

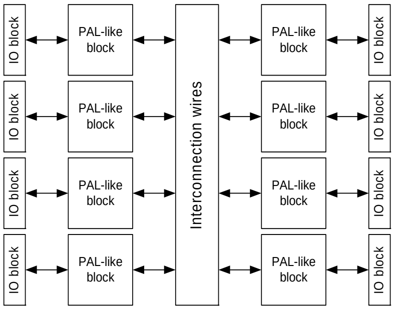
- **PLA, PAL and SPLD typically contain small number of outputs (e.g., 16 outputs) with many inputs (e.g., 36 inputs) and a fair number of product terms.**
  - Therefore only good for simple circuits where each equation has a wide fanin.

- **Using a** Complex Programmable Logic Device (CPLD) **is the “next step” if we have a large complicated circuit…**

- **CLPD consists of many SPLD connected together by a** Programmable Routing Fabric **all in the** same **IC.**

Complex Programmable Logic Device (CPLD)

- **Typical architecture (each PAL-like block has many inputs – e.g., 36 - , many product terms – e.g., 80 – and several outputs – e.g., 16).**

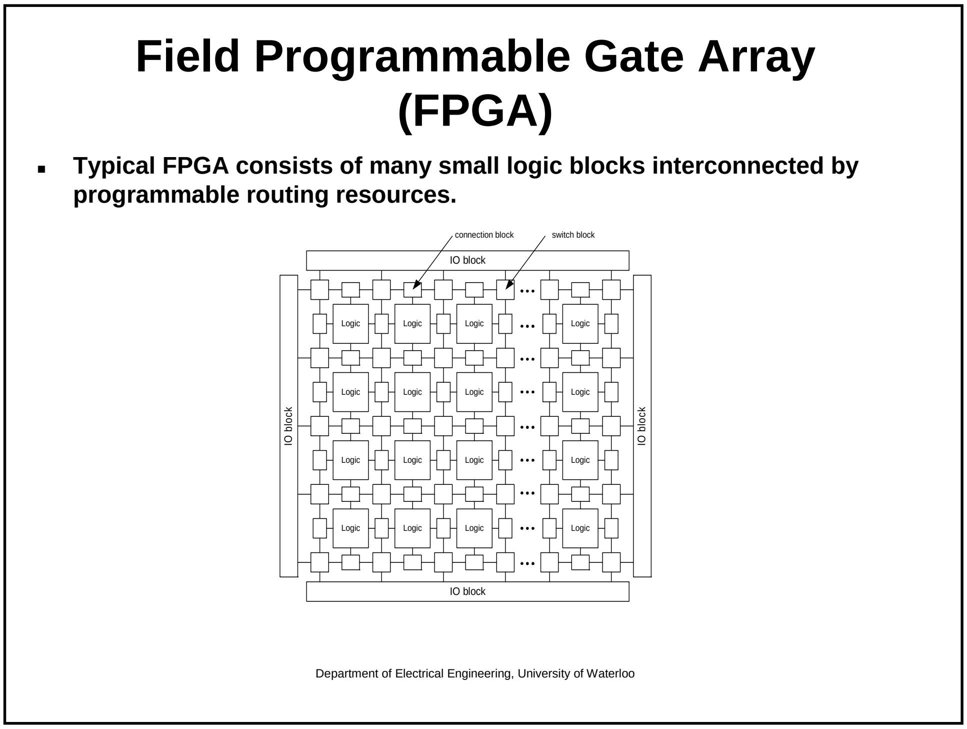


Types of PLA, PAL, SPLD and CPLD

- **Programming of these devices is similar to ROM; i.e., these devices are typically either PROM, EPROM or EEPROM.**
- **Programming info is generated (perhaps with a software tool), and the bit stream of program info is provided to one (or a few) additional pins on the device.**
- **Also possible (these days) to have SRAM-based PLDs…**
  - In SRAM devices, the programming info is **lost** when power is **turned off**.
  - Necessary to re-program device every time the system is powered up.
  - Often to see a **configuration EPROM** beside an SRAM based PLD on a circuit board.
- **Two chip solution… The EPROM holds the program that gets applied to the PLD upon power up.**

Field Programmable Gate Array (FPGA)

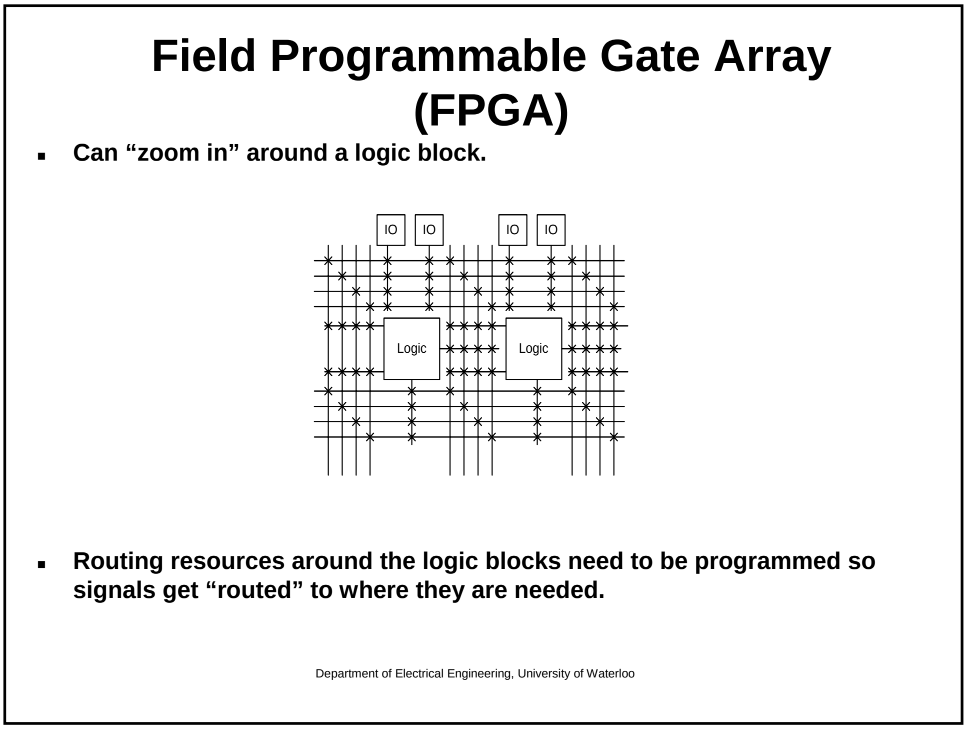
- **Another type of programmable device capable of handling large circuits.**
- **Different from a CPLD:**
  - Logic is not implemented in terms of Product Terms/MacroCells
  - Implemented using Lookup Table (LUT) which are like little memories



Field Programmable Gate Array (FPGA)

- Typical FPGA consists of many small logic blocks interconnected by programmable routing resources.

Department of Electrical Engineering, University of Waterloo



Field Programmable Gate Array (FPGA)

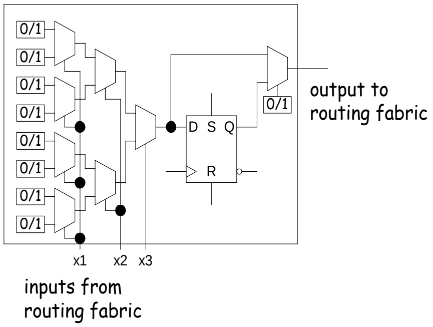
- Can “zoom in” around a logic block.

- Routing resources around the logic blocks need to be programmed so signals get “routed” to where they are needed.

Department of Electrical Engineering, University of Waterloo

Field Programmable Gate Array (FPGA)

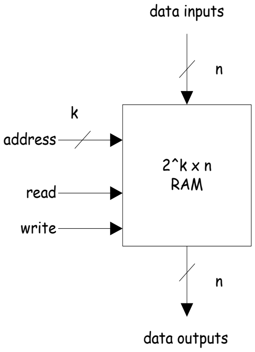
- Can “zoom in” inside a logic block (e.g., 3-input logic block):

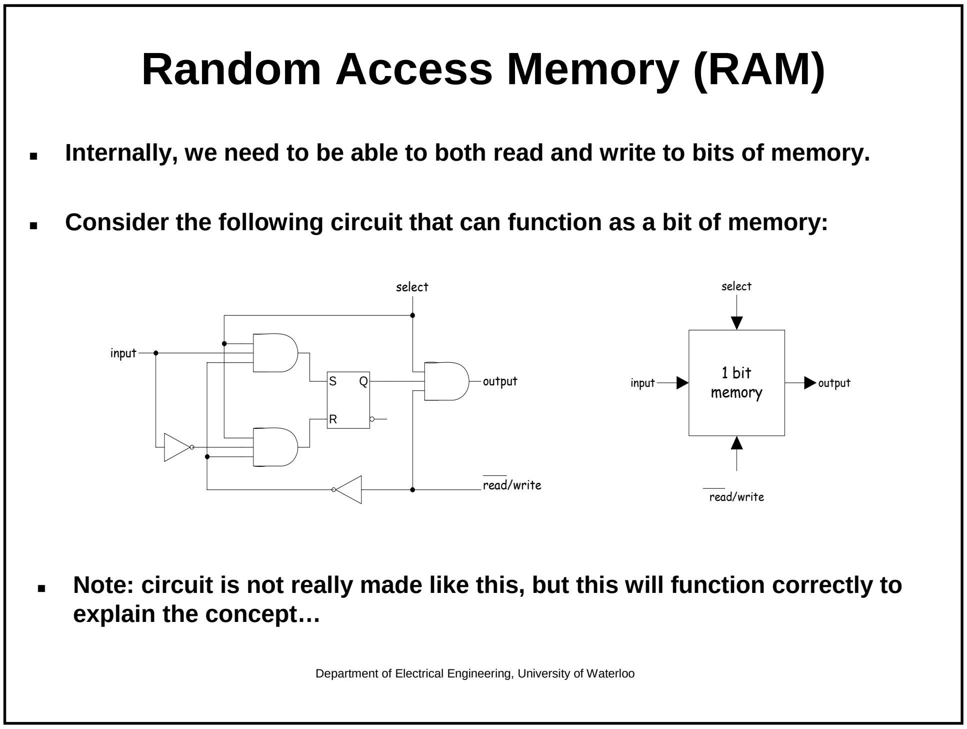


- Can implement any 3-input function by properly programming the configuration bits.

Random Access Memory (RAM)

- Storage device to which we can both read and write information.





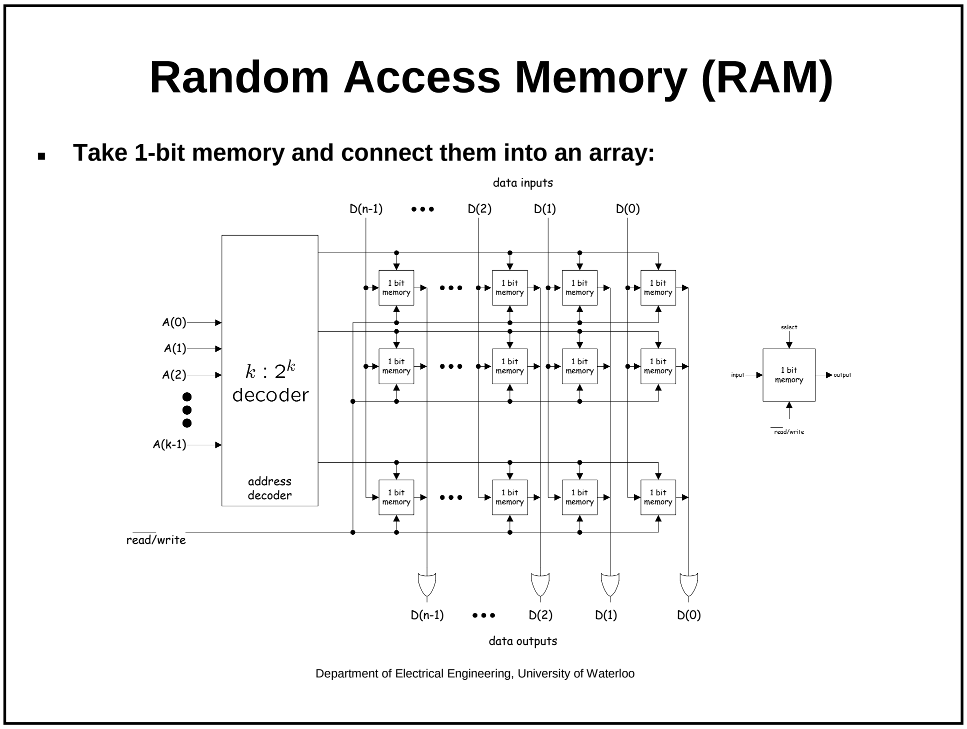
Random Access Memory (RAM)

- Internally, we need to be able to both read and write to bits of memory.
- Consider the following circuit that can function as a bit of memory:

select
input
S
Q
R
output
$\overline{\text{read}}$/write
select
input
1 bit memory
output
$\overline{\text{read}}$/write

- Note: circuit is not really made like this, but this will function correctly to explain the concept…

Department of Electrical Engineering, University of Waterloo



Random Access Memory (RAM)

- Take 1-bit memory and connect them into an array:

data inputs
D(n-1) ••• D(2) D(1) D(0)
A(0)
A(1)
A(2)
A(k-1)
$k : 2^k$ decoder
address decoder
$\overline{\text{read}}$/write
1 bit memory
D(n-1) ••• D(2) D(1) D(0)
data outputs
select
input
output
$\overline{\text{read}}$/write

Department of Electrical Engineering, University of Waterloo