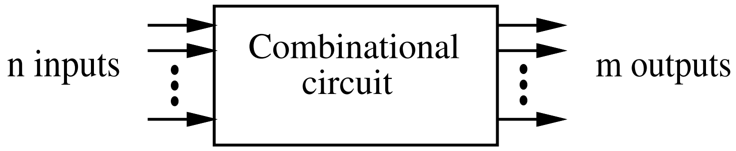
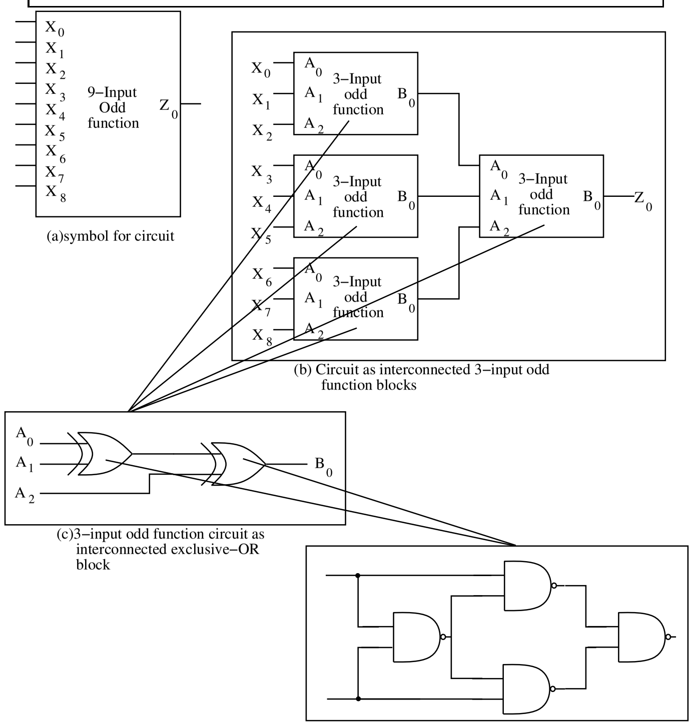
Combinational (Combinatorial) Logic

- A **combinational** logic circuit has
  - A set of n Boolean **inputs**,
  - A set of m Boolean **outputs**,
  - A **function** mapping inputs to outputs such that the outputs depend **only** on the inputs.



- The outputs of a **sequential** logic circuit depend on the inputs and **the present state** (see next chapter).

Example: 9-input Odd Function



(a)symbol for circuit

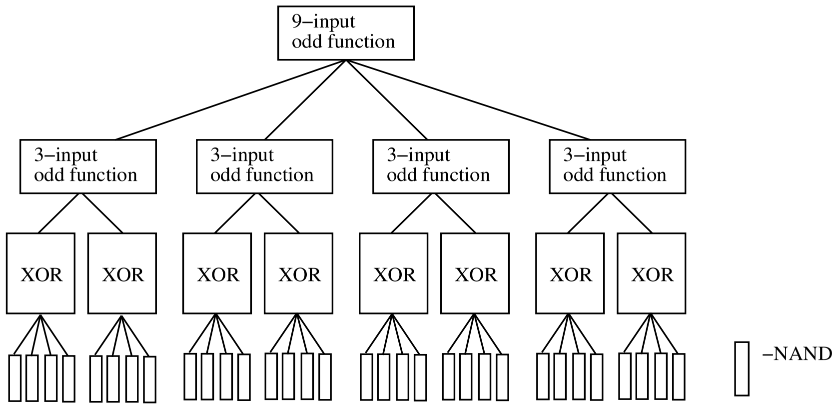
(b) Circuit as interconnected 3−input odd function blocks

(c)3−input odd function circuit as interconnected exclusive−OR block

(c) Exclusive−OR block as interconnect−ed NANDs

Hierarchical Design

- How to design a complex function that maps inputs to outputs?
  - Decompose the function into smaller **blocks**.
  - Subdivide blocks into finer blocks until **primitive blocks** are obtained.
- Exp: 9-input odd function.
  - Top level: 9 inputs, one output odd function $\longrightarrow$ 2nd level: 4 3-input odd function in two levels $\longrightarrow$ 3rd level: 2 2-input XOR $\longrightarrow$ Primitive level: 4 2-input NAND gates.
  - Needs $4 \times 2 \times 4 = 32$ two-input NAND gates.



Design Methodology

- Whenever possible, decompose a complex design into common, **reusable functional blocks**.
- These blocks are tested and well documented.
- **C**omputer-**A**ided **D**esign (**CAD**) tools might include them in libraries.
  - Example CAD tools: Schematic capture, Logic simulators, Timing verifiers, High-level synthesis tools: **H**ardware **D**escription **L**anguages (**HDLs**)
- **C**omputer-**A**ided **M**anufacturing (**CAM**) tools might know how to manufacture and test them.

Top-down vs. Bottom-up Design

- **Top-down design:** proceeds from an abstract, high-level specification to a more and more detailed design by decomposition and successive refinement.

- **Bottom-up design:** starts with detailed primitive elements and combines them into larger and larger functions.

- Designs usually proceed from **both directions** simultaneously.

- Top-down design answers: **What are we building?**

- Bottom-up design answers: **How do we build it?**

- Top-down controls **complexity** while bottom-up handles **detailed**.

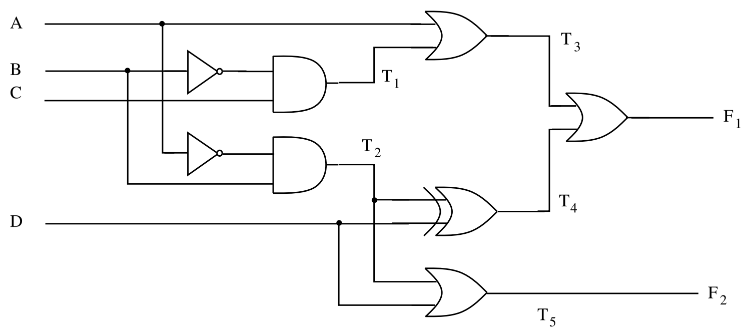
- Before design, we first need to cope with circuit **analysis**.

Analysis Procedures

- **Analysis:** From a **design** to a **specification** of the behavior.
  - Logic diagram to Boolean equations.
  - Logic diagram to function table.

- Boolean functions from logic diagrams: First determine that the circuit is combinational (no feedback loops), then
  1. Identify and label all gate outputs that a function of the input variables. Obtain the Boolean functions for these labeled gate outputs.
  2. Identify and label all gate outputs that a function of inputs or previously labeled gates. Obtain the Boolean functions for them.
  3. Repeat Step 2 until all outputs are completed.
  4. Back substitute until all functions are specified in terms of inputs only.

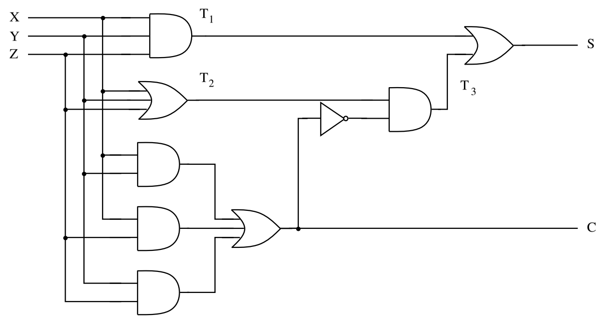
Analysis Example



- $T_1 = \bar{B}C; \quad T_2 = \bar{A}B.$
- $T_3 = A + T_1 = A + \bar{B}C;$
  $T_4 = T_2 \oplus D = (\bar{A}B) \oplus D = \bar{A}B\bar{D} + AD + \bar{B}D;$
  $T_5 = T_2 + D = \bar{A}B + D.$
- $F_2 = T_5 = \bar{A}B + D;$
  $F_1 = T_3 + T_4 = A + \bar{B}C + \bar{A}B\bar{D} + AD + \bar{B}D = A + \bar{B}C + B\bar{D} + \bar{B}D.$

Truth Tables from Logic Diagram

- For each of 2^n input vectors, determine the outputs of the circuit.
- $S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ; \quad C = XY + YZ + XZ.$

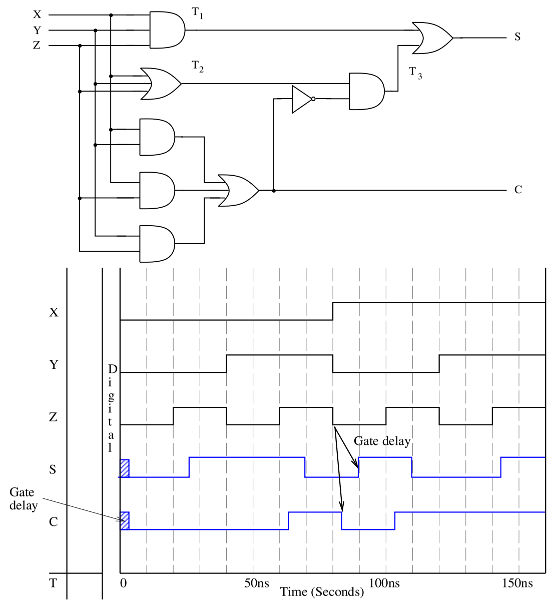


X	Y	Z	C	$\overline{C}$	T_1	T_2	T_3	S
0	0	0	0	1	0	0	0	0
0	0	1	0	1	0	1	1	1
0	1	0	0	1	0	1	1	1
0	1	1	1	0	0	1	0	0
1	0	0	0	1	0	1	1	1
1	0	1	1	0	0	1	0	0
1	1	0	1	0	0	1	0	0
1	1	1	1	0	1	1	0	1

Logic Simulation

- **Logic simulation** is a fast, accurate (should be) method of analyzing a circuit to see its logic waveforms or a truth table is correct.

- Assumptions: gate delay = 2 ns; inputs changed every 20 ns.



Design Synthesis Procedure

- **Synthesis:** From **specification** to design **implementation**.
  1. Define the problem (specification).
  2. Identify the inputs and outputs.
  3. Derive function tables or equations.
  4. Minimize the Boolean function.
  5. Implement the network (draw the logic diagram, write HDL code, etc).
  6. Verify the circuit.

- Will study the design of a code converter to see these steps.

Code Converter Design

- A **code converter** transforms one internal representation of data to another.
- **Problem:** Design a BCD to Excess-3 code converter.
- **Specification: BCD code:** 4-bit patterns "0000" to "1001 for digits 0 to 9; **Excess-3 code:** BCD code + "0011" for digits 0 to 9.
- Truth table for code converter:

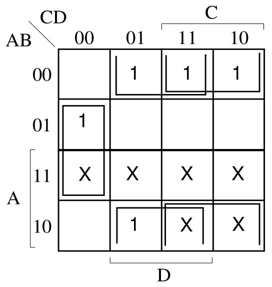
Decimal Digit	Input BCD				Output Excess–3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

Code Converter

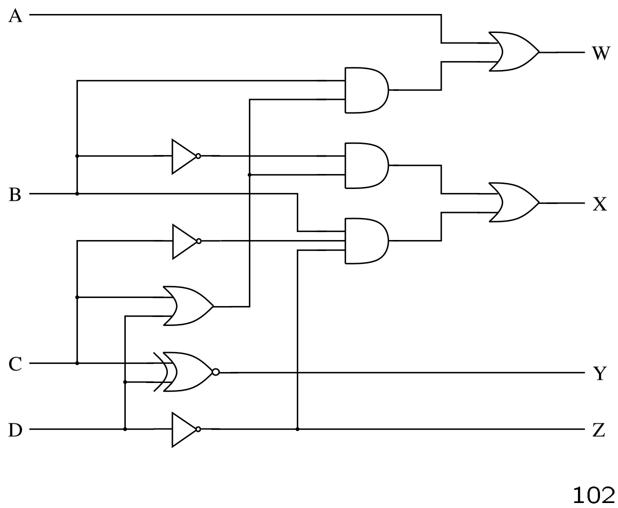
- Equations: (share terms to minimize the implementation cost)

$$\begin{aligned} W &= A + BC + BD = A + B(C+D) \\ X &= \bar{B}C + \bar{B}D + B\bar{C}\bar{D} = \bar{B}(C+D) + B\bar{C}\bar{D} \\ Y &= CD + \bar{C}\bar{D} = \overline{C \oplus D} \\ Z &= \bar{D} \end{aligned}$$

- Implement the network:



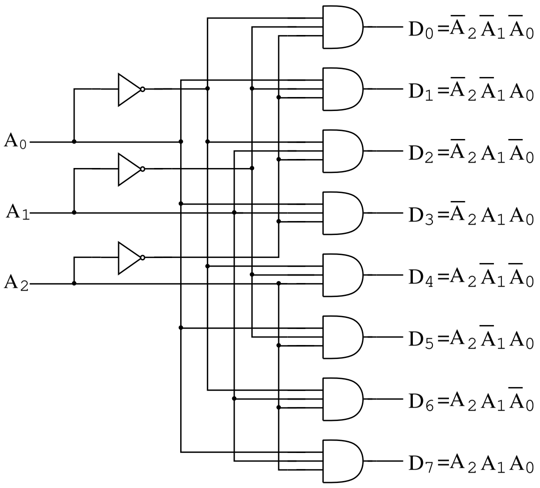
X=$\bar{B}$C+$\bar{B}$D+B $\bar{C}$ $\bar{D}$



Decoder

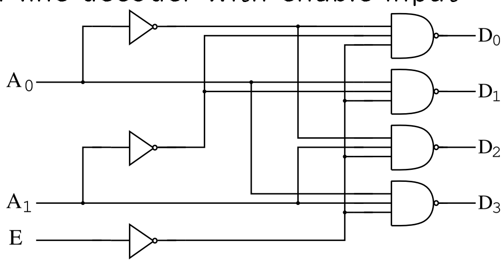
- A **decoder** converts n binary bits to a maximum of 2^n unique output lines.
- Decoders implement **minterms** directly.

Inputs			Outputs							
A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0



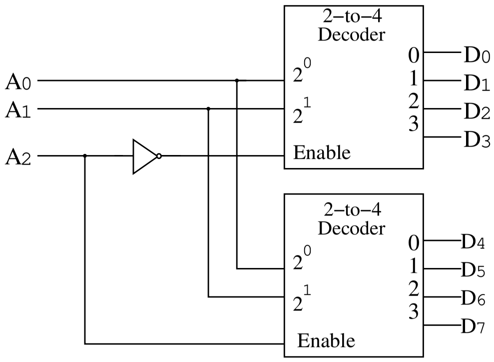
Decoder with Enable Input

- A 2-to-4 line decoder with enable input



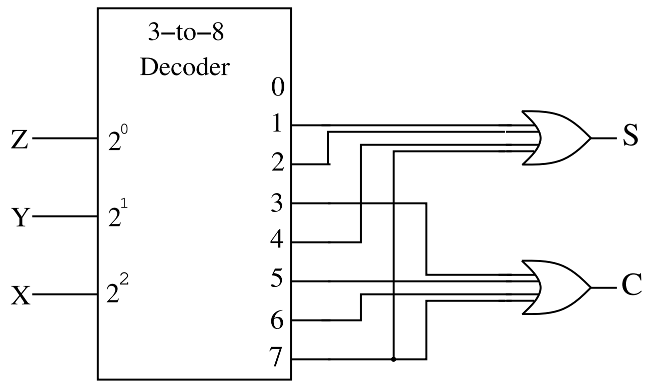
E	A_1	A_0	D_0	D_1	D_2	D_3
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	X	X	1	1	1	1

- Construct a 3-to-8 decoder from 2-to-4 decoders



Circuit Implemented by a Decoder

- Decoders implement **minterms** directly.
- Use a decoder and an **OR** gate to form sum-of-minterms directly.
- Exp: $S(X,Y,Z) = \sum_m(1,2,4,7)$; $C(X,Y,Z) = \sum_m(3,5,6,7)$



Encoder

- **Encoders** perform the "inverse" operation of decoders, taking a code in one format and encoding it into another format.
- Exp: Octal-to-binary encoder

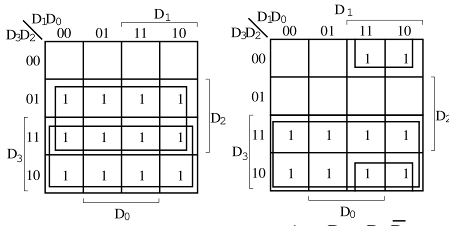
$$\begin{aligned} A_0 &= D_1 + D_3 + D_5 + D_7 \\ A_1 &= D_2 + D_3 + D_6 + D_7 \\ A_2 &= D_4 + D_5 + D_6 + D_7 \end{aligned}$$

Inputs								Outputs		
D7	D6	D5	D4	D3	D2	D1	D0	A2	A1	A0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Priority Encoder

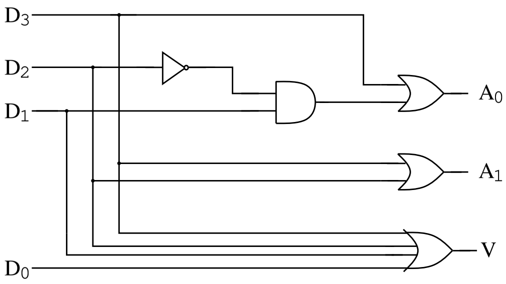
- **Priority encoders**: If two or more inputs are equal to 1 at the same time, the input having the highest priority takes precedence.

Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1



$A_1 = D_2 + D_3$

$A_0 = D_3 + D_1\bar{D}_2$



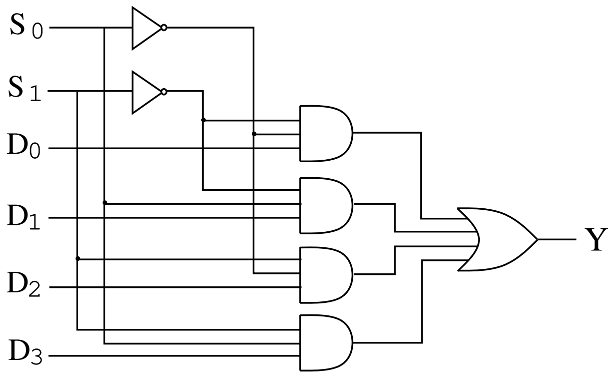
Multiplexer

- A **multiplexer** uses n binary select bits to choose from a maximum of 2^n unique input lines.

- Like a decoder, it decodes **minterms** internally; unlike a decoder, it has only **one output line**.

- The output of the multiplexer is the data input whose index is specified by the n-bit code.

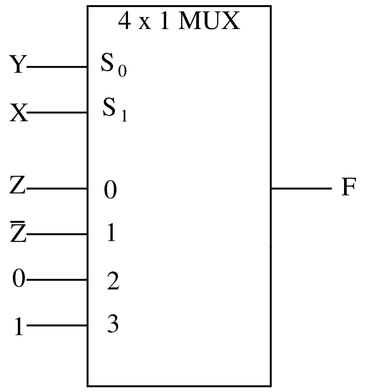
S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3



Combinational Circuit Implemented with Multiplexers

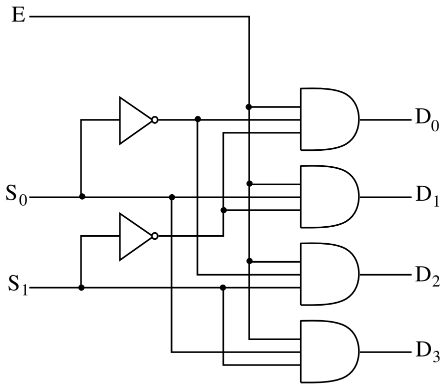
- Can implement any Boolean function of n variables with an n selection input, 2^n data input multiplexer.
  - Simply tie each input to the "0" or "1" line.
- Also possible to implement any $(n+1)$-variable function with a 2^n-to-1 multiplexer.
  - Use the $(n+1)$-st variable in true or complement form, depending on what the truth table requires.

X	Y	Z		
0	0	0	0	F = Z
0	0	1	1	
0	1	0	1	F = $\overline{Z}$
0	1	1	0	
1	0	0	0	F = 0
1	0	1	0	
1	1	0	1	F = 1
1	1	1	1	



Demultiplexer

- A **demultiplexer** is a digital function that performs the inverse of the multiplexing operation.
  - Receives info from a single line and transmits it to one of 2^n possible ouput lines.
- A 1-to-2^n demultiplexer is identical to an n-to-2^n decoder with enable input.
  - Exp: Decoder: S_0, S_1 are data inputs, E is the enable; Demultiplexer: E is the data input, the other inputs accept the selection variables.

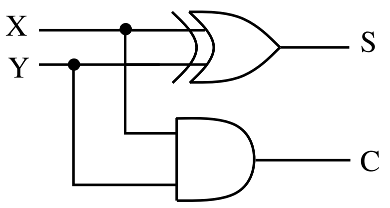


Half Adder

- A **half adder** adds two bits, X and Y, to produce a 1-bit **sum**, S, and a 1-bit **carry**, C.

- $S = (\bar{X}Y + X\bar{Y}) = X \oplus Y$;
  $C = XY$.

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

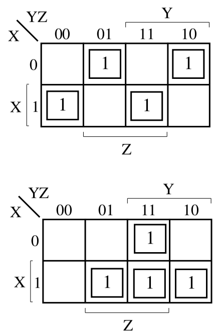


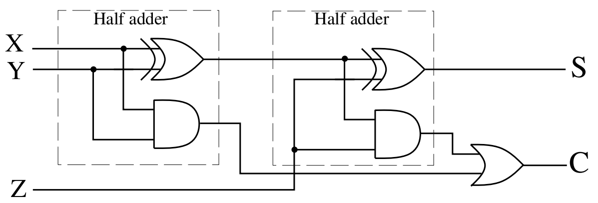
Full Adder

- A **full adder** adds two bits, X and Y, and one **carry input**, Z, to produce a 1-bit **sum**, S, and a 1-bit **carry**, C.

- $S = (\bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ) = X \oplus Y \oplus Z$;
  $C = XY + XZ + YZ = XY + Z(X \oplus Y)$.

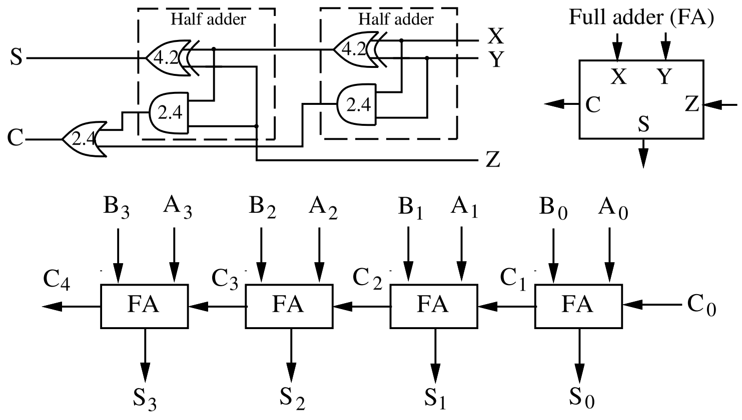
Inputs			Outputs	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1





Ripple Carry Adder

- 4-bit ripple carry adder example: Inputs: C_0, $A = A_{n-1} \dots A_0$, $B = B_{n-1} \dots B_0$; Outputs: $s_{n-1} \dots s_0$ and $c_n \dots c_1$.



- Propagation delays: $C_0(A_0, B_0) \to C_1$: 4.8(9.0) ns; $C_0(A_0, B_0) \to C_2$: 9.6(13.8) ns; $C_0(A_0, B_0) \to C_3$: 14.4(18.6) ns; $C_0(A_0, B_0) \to C_4$: 19.2(23.4) ns.
- **Question:** Can we improve the delays?
  - The carries are the bottleneck!

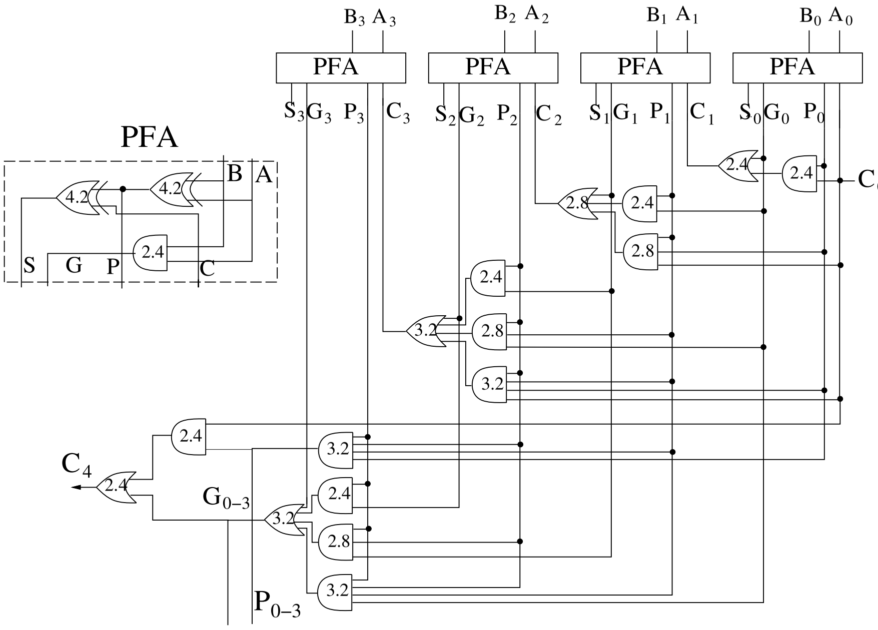
Carry Lookahead Adder

- **Observation:** Major part of the expressions for each carry in the chain can be **precomputed**.
  - Exp: Add $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$.
  - **Carry-generate function:** $G_i = A_iB_i$; **carry-propagate function:** $P_i = A_i \oplus B_i$. Then, the 4 carries C_1, C_2, C_3, C_4 can be expressed as follows:

$$\begin{aligned}
C_1 &= G_0 + P_0C_0 && (1)\\
C_2 &= G_1 + P_1C_1 && (2)\\
&= G_1 + P_1G_0 + P_1P_0C_0 \\
C_3 &= G_2 + P_2C_2 && (3)\\
&= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \\
C_4 &= G_3 + P_3C_3 && (4)\\
&= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0
\end{aligned}$$

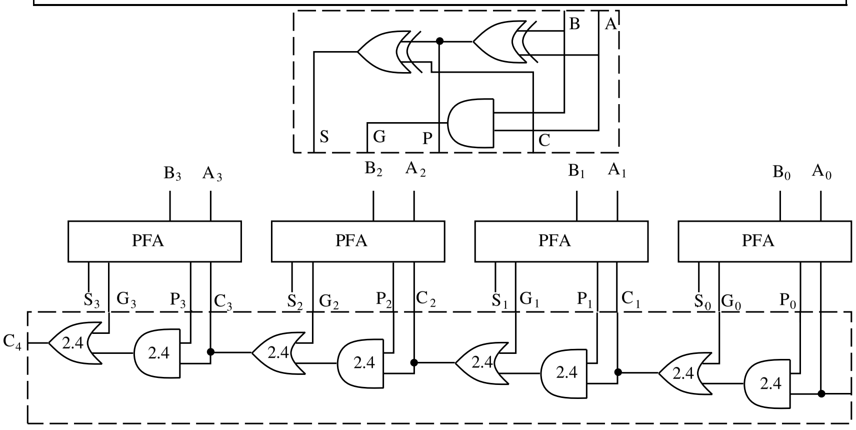
- Carries can be computed directly from input bits A_i, B_i and input carry C_0. No ripple effect!
- Extend to n-bit adders?
  - **Problem: Fan-in limitation?**

Carry Lookahead Adder



- **Group propagate function:** $P_{0-3} = P_3P_2P_1P_0$;
  Group generate function: $G_{0-3} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$;
  $C_4 = G_{0-3} + P_{0-3}C_0$.

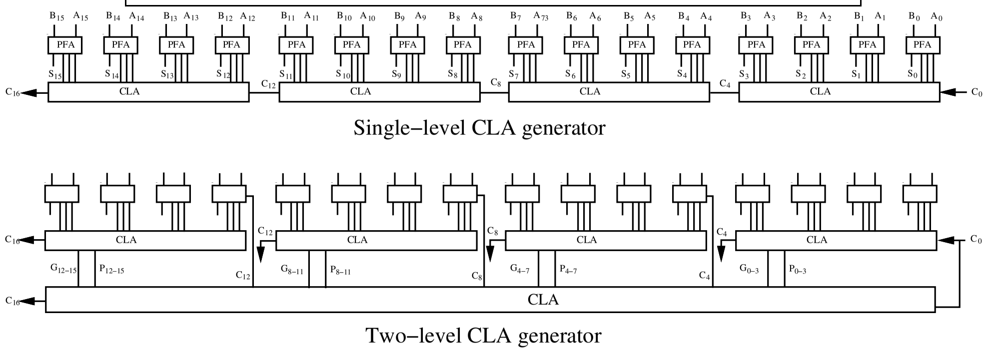
Carry Lookahead vs. Ripple Carry Adders



Ripple carry adder

Carry chain	Ripple delay	CLA delay
$C_0(A_0, B_0)$ to C_1	4.8 (9.0)	4.8 (9.0)
$C_0(A_0, B_0)$ to C_2	9.6 (13.8)	5.6 (9.8)
$C_0(A_0, B_0)$ to C_3	14.4 (18.6)	6.4 (10.6)
$C_0(A_0, B_0)$ to C_4	19.2 (23.4)	4.8 (13.0)

Carry Lookahead Adder Construction



- Carry chain delays for 16-bit adders:

Carry chain	Ripple delay	One-level CLA	Two-level CLA
$C_0(A_0, B_0)$ to C_4	19.2 (23.4)	4.8 (13.0)	4.8 (13.0)
$C_0(A_0, B_0)$ to C_8	38.4 (42.6)	9.6 (17.8)	5.6 (16.2)
$C_0(A_0, B_0)$ to C_{12}	57.6 (61.8)	14.4 (22.6)	6.4 (17.0)
$C_0(A_0, B_0)$ to C_{16}	76.8 (81.0)	19.2 (27.4)	4.8 (19.4)

- n-bit ripple-carry adder has $2n$ gate delays; n-bit adder with $\lceil \log_4 n \rceil$ levels of CLA needs $2\lceil \log_4 n \rceil$ gate delays to generate top-level carries.

Diminished Radix Complements

- **Diminished radix $(r-1)$'s complement:** Given a number N in based r with n digits, the $(r-1)$'s complement is defined as

$$(r^n - 1) - N. \tag{5}$$

- Exp1: $r = 10, N = 1234_{10}, n = 4 \Longrightarrow$ 9's complement of 1234_{10} is $9999_{10} - 1234_{10} = 8765_{10}$.

- Exp2: $r = 2, N = 01110011_2, n = 8 \Longrightarrow$ 1's complement of 01110011_2 is $11111111_2 - 01110011_2 = 10001100_2$.

- Finding one's complement $\Longrightarrow$ complementing each individual bit.

Radix Complements

- **Radix r's Complement:** Given a number N in based r with n digits, the r's complement is defined as

$$\begin{array}{rl} r^n - N & for \quad N \neq 0; \\ 0 & for \quad N = 0. \end{array}$$

- Exp1: $r = 10, N = 1234_{10}, n = 4 \Longrightarrow$ 10's complement of 1234_{10} is $10000_{10} - 1234_{10} = 8766_{10}$.
- Exp2: $r = 2, N = 01110011_2, n = 8 \Longrightarrow$ 2's complement of 01110011_2 is $100000000_2 - 01110011_2 = 10001101_2$.
- r's complement $= (r-1)$'s complement $+1$.
- The 2's complement of 0 is zero!

Subtraction with Radix Complements

- Subtract two n-digit, unsigned numbers $M - N$ in base r:
  1. Add the minuend M to the r's complement of the subtrahend N:
  $$M + (r^n - N) \quad = \quad M - N + r^n.$$
  2. If $M \geq N$, the sum will produce an end carry C_n which is discarded; what is left is the result, $M - N$.
  3. If $M < N$, the sum does not produce an end carry C_n and is equal to $r^n - (N - M)$, which is the r's complement of $(N - M)$. We shall take the r's complement of the sum and place a negative sign in front.

Find $543_{10} - 123_{10}$. Step 1: 1000 − 123 877 Step 2: 543 + 877 1420 Step 3: Since M >= N, discard the carry. Ans: 420	Find $123_{10} - 543_{10}$. Step 1: 1000 − 543 457 Step 2: 123 + 457 580 Step 3: M < N 1000 − 580 420 Ans: −420	Find 1010100 − 1000011 Step 1: 2's complement of 1000011 is 0111101. Step 2: 1010100 + 0111101 1 0010001 Step 3: Since M >= N, discard the carry. Ans: 0010001	Find 1000011 − 1010100 Step 1: 2's complement of 1010100 is 0101100. Step 2: 1000011 + 0101100 1101111 Step 3: Since M < N, complement the result. Ans: (−) 0010001

Subtraction Using 1's Complement

- Use addition of the 1's complement to subtract two numbers:
  1. **When an end carry occurs:** Since $(r-1)$'s complement is one less than the r's complement, the result is one less than the correct sum. $\Longrightarrow$ Discard the end carry and add one to the sum (**end-around carry**).
  2. **When no end carry occurs:** The result is negative. $\Longrightarrow$ Use 1's complement to represent the negative result.

Find 1010100 – 1000011	Find 1000011 – 1010100
Step 1: 1's complement of 1000011 is 0111100.	Step 1: 1's complement of 1010100 is 0101011.
Step 2: 1010100 + 0111100 = 1 0010000	Step 2: 1000011 + 0101011 = 1101110
Step 3: A carry occurs, add carry, end around.	Step 3: No carry occurs, 1's complement the result.
Ans: 0010001	Ans: (–) 0010001

Signed Binary Numbers

- Use the **most significant bit** as a **sign bit**: $sa_{n-2}\ldots a_0$.
  - $s=0$: positive numbers; $s=1$: negative numbers.
- Three popular ways to interpret $a_{n-2}\ldots a_0$:
  - **Signed magnitude:** The $n-1$ digits are interpreted as a positive magnitude.
  - **Signed 1's complement:** Use the 1's complement to compute the **negative** numbers.
  - **Signed 2's complement:** Use the 2's complement to compute the **negative** numbers.
- Exp: Signed-magnitude: 10001001 (-9); signed 1's complement: 11110110 (-9); signed 2's complement: 11110111 (-9).
- For positive numbers, the three ways give the same representation.

Three Signed-Number Representations

Decimal	Signed 2's Compement	Signed 1's Complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
–0	–	1111	1000
–1	1111	1110	1001
–2	1110	1101	1010
–3	1101	1100	1011
–4	1100	1011	1100
–5	1011	1010	1101
–6	1010	1001	1110
–7	1001	1000	1111
–8	1000	–	–

Signed-Magnitude Arithmetic

- Addition
  - **If signs are the same:**
    1. Add the magnitudes.
    2. Check for **overflow** (a carry into the sign bit).
    3. The sign of the result is the same.
  - **If signs differ:**
    1. Subtract the magnitude of the smaller from the magnitude of the larger.
    2. Use the sign of the larger magnitude for the sign of the result.
    3. **Overflow will never occur.**
- Subtraction: Complement the sign bit of the number you are subtracting and follow the rules for addition.
- Same signs: $000 + 011 = 011$; $010 + 010 = x00$ (**overflow!**); $101 + 101 = 110$; $110 + 110 = x00$ (**overflow!**).
- Different signs: $001 + 110 = 101$ ($010 - 001$, take $-$ sign); $111 + 010 = 101$; $100 + 000 = ?00$ ($+$ or $-$ zero?).

Signed-Complement Arithmetic

- Addition
  1. Add the numbers (including the sign bits),
     - 2's complement: discarding a carry out of the sign bit.
     - 1's complement: using an end-around carry.
  2. If the sign bits were the same for both numbers and the result is different, an overflow has occurred.
  - Boolean equation for overflow:

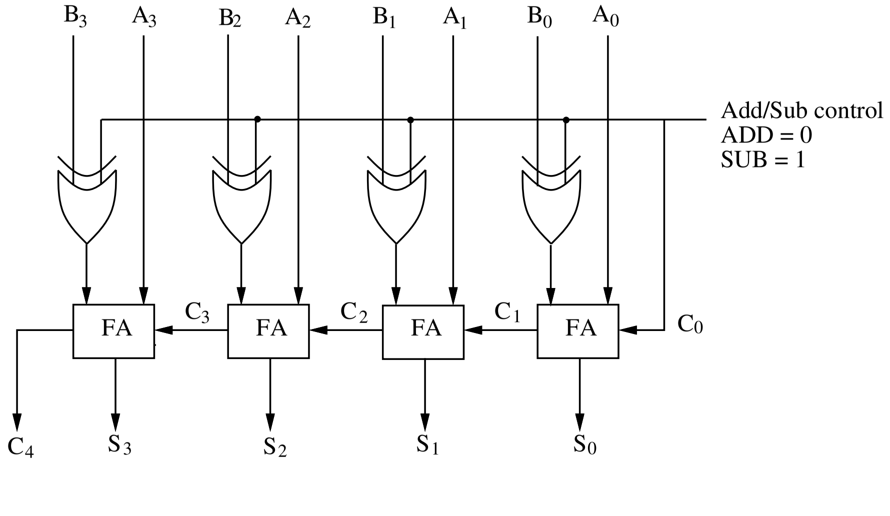
$$Overflow = A_{n-1}B_{n-1}\overline{S_{n-1}} + \overline{A_{n-1}}\ \overline{B_{n-1}}S_{n-1}.$$

+70	0	1000110	−70	1	0111010
+80	0	1010000	−80	1	0110000
+150	1	0010110	−150	0	1101010

- Subtraction: Form the complement of the number you are subtracting and follow the rules for addition.

Binary Adder/Subtractor

- Subtraction can be achieved by adding 2's complement.
  - Complement each bit (1's complement) and add 1 to the result.
- $C_0 = 0$: addition; $C_0 = 1$: Form the 1's complement of B using EXOR gates and add one.



Binary Multiplication

- **Compute partial products**, and **justify and sum** the partial products.
  - $m \times n$ digit multiply generates up to an $(m+n)$-digit result.
- **Compute partial products:**
  - n-digit, base-2 numbers require adding up to n digits (with carries) in a column.
  - Multiply the row of multiplicand digits by each multiplier digit, one at a time.
  - If the multiplier digit is 0 $\Longrightarrow$ all zero; if the multiplier digit is 1 $\Longrightarrow$ the same as the multiplicand.

$$\begin{array}{rrrrrr} & & & 1 & 0 & 1 \\ & & \text{x} & 0 & 1 & 1 \\ \hline & & & 1 & 0 & 1 \\ & & 1 & 0 & 1 & \\ & 0 & 0 & 0 & & \\ \hline 0 & 0 & 1 & 1 & 1 & 1 \end{array}$$

2×2 Multiplier

- For $m \times n$ digit multiply, we need $m \times n$ AND gates and $(n-1)$ m-bit adders to produce a product of $m+n$ bits.

$$\begin{array}{cccc} & & B_1 & B_0 \\ & & A_1 & A_0 \\ \cline{3-4} & & A_0B_1 & A_0B_0 \\ & A_1B_1 & A_1B_0 & \\ \hline P_3 & P_2 & P_1 & P_0 \end{array}$$

$P_0 = A_0 B_0$

$P_1 = A_0B_1 \oplus A_1B_0$

$P_2 = A_1B_1 \oplus (A_0B_1 \bullet A_1B_0)$

$P_3 = A_1B_1 \bullet (A_0B_1 \bullet A_1B_0)$