Systemy Komputerowe

(a właściwie Algorytmy w Systemach Komputerowych)
Edycja lato 2024/25

Program wykładu

1. Sprzętowa implementacja operacji arytmetycznych: szybkie algorytmy dodawania (sumatory carry-lookahead i Ling'a), mnożenia (drzewa Wallace'a i kodowanie Booth'a) i dzielenia (non-restoring division, SRT division).
2. Niskopoziomowa reprezentacja programów (kod trójadresowy, graf przepływu sterowania), analiza przepływu danych (obliczenia stałopunktowe).
3. Architektura współczesnego procesora (przetwarzanie potokowe i *out-of-order*). Graf przepływu danych i algorytm Tomasulo.
4. Spekulatywne wykonywanie instrukcji, przewidywanie skoków (perceptron, TAGE), atak Spectre.
5. Instrumentacja kodu, profilowanie (graf wywołań funkcji) i optymalizacja kodu (algorytm *local value numbering*).
6. Struktury pamięci podręcznych, algorytmy *cache-oblivious*.
7. Algorytmy szeregowania wątków i balansowania obciążenia procesorów (algorytm ULE, kolejka kalendarzowa).

Program wykładu (cd)

8. Translacja adresów, wielopoziomowa i odwrócona tablica stron.
9. Pamięć wirtualna, algorytmy zastępowania stron (algorytm WSCLOCK, aging, Clock-Pro).
10. Algorytmy dynamicznego przydziału pamięci (buddy systems, boundary tags). Algorytmy odśmiecania (mark and sweep, algorytm kopiujący).
11. Struktury danych systemów plików (grupy bloków, i-node, B-drzewo, journaling). Planowanie operacji I/O (budget-fair queueing).
12. Przetwarzanie współbieżne. Implementacja środków synchronizacji (synchronizacja pamięci podręcznych, exponential backoff).
13. Kontrola przepływności w protokole TCP. Obliczanie sum kontrolnych i kodów korekcyjnych.
14. Sieci w układach scalonych: topologia i trasowanie pakietów.

Organizacja zajęć

Wykład:

- materiały pomocnicze (slajdy + odnośniki do literatury) w SKOSie

Ćwiczenia:

- Zadania tablicowe
- HackMD
- deklaracje
- Progi zaliczeń: 50% — 3.0, 90% – 5.0

Egzamin:

- (prawdopodobnie) test złożony z pytań otwartych i zamkniętych
- Kiedy?
- Nie wykluczam innych form weryfikacji wiedzy (kolokwia?, quizy?)

Wykład 1

Sprzętowa implementacja operacji arytmetycznych: szybkie algorytmy dodawania (sumatory carry-lookahead i Ling'a), mnożenia (drzewa Wallace'a i kodowanie Booth'a) i dzielenia (non-restoring division, SRT division)

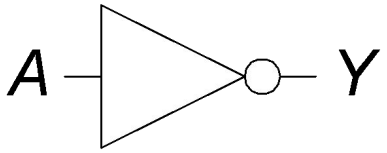
Bramki

Logic Gates

- **Perform logic functions:**
  - inversion (NOT), AND, OR, NAND, NOR, etc.
- **Single-input:**
  - NOT gate, buffer
- **Two-input:**
  - AND, OR, XOR, NAND, NOR, XNOR
- **Multiple-input**

Single-Input Logic Gates

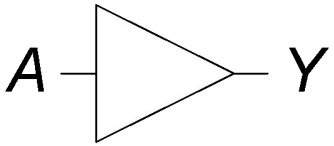
NOT



$Y = \overline{A}$

A	Y
0	
1	

BUF

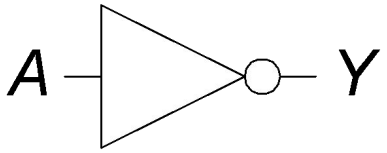


$Y = A$

A	Y
0	
1	

Single-Input Logic Gates

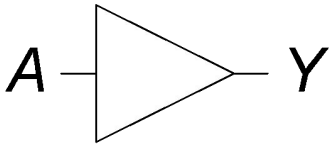
NOT



$Y = \overline{A}$

A	Y
0	1
1	0

BUF

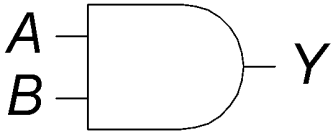


$Y = A$

A	Y
0	0
1	1

Two-Input Logic Gates

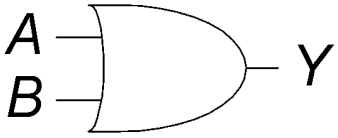
AND



$Y = AB$

A	B	Y
0	0	
0	1	
1	0	
1	1	

OR

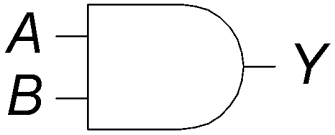


$Y = A + B$

A	B	Y
0	0	
0	1	
1	0	
1	1	

Two-Input Logic Gates

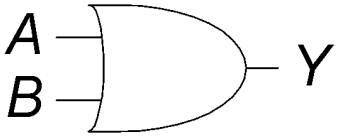
AND



$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR

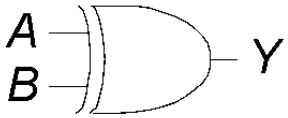


$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

More Two-Input Logic Gates

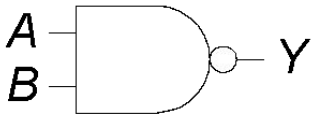
XOR



$Y = A \oplus B$

A	B	Y
0	0	
0	1	
1	0	
1	1	

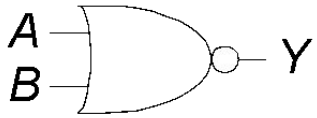
NAND



$Y = \overline{AB}$

A	B	Y
0	0	
0	1	
1	0	
1	1	

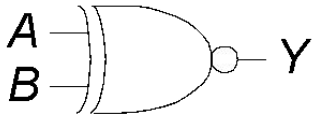
NOR



$Y = \overline{A + B}$

A	B	Y
0	0	
0	1	
1	0	
1	1	

XNOR

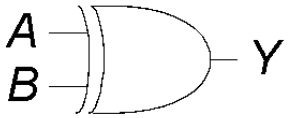


$Y = \overline{A \oplus B}$

A	B	Y
0	0	
0	1	
1	0	
1	1	

More Two-Input Logic Gates

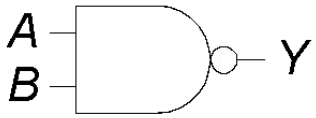
XOR



$Y = A \oplus B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

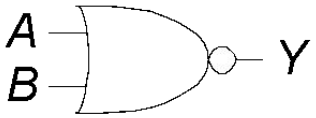
NAND



$Y = \overline{AB}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

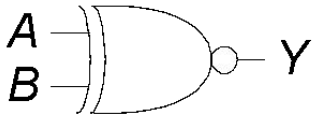
NOR



$Y = \overline{A + B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

XNOR

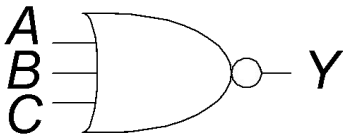


$Y = \overline{A \oplus B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Multiple-Input Logic Gates

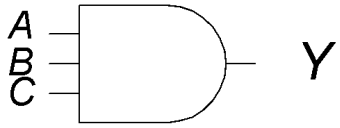
NOR3



$$Y = \overline{A+B+C}$$

A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

AND3

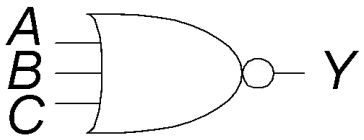


$$Y = ABC$$

A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Multiple-Input Logic Gates

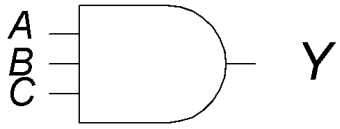
NOR3



$$Y = \overline{A+B+C}$$

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

AND3



$$Y = ABC$$

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

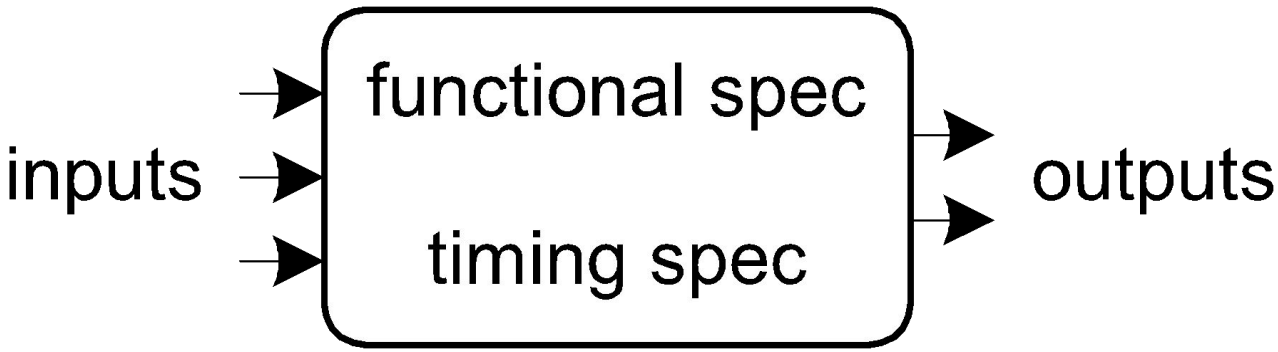
- Multi-input XOR: Odd parity

Bramki

Introduction

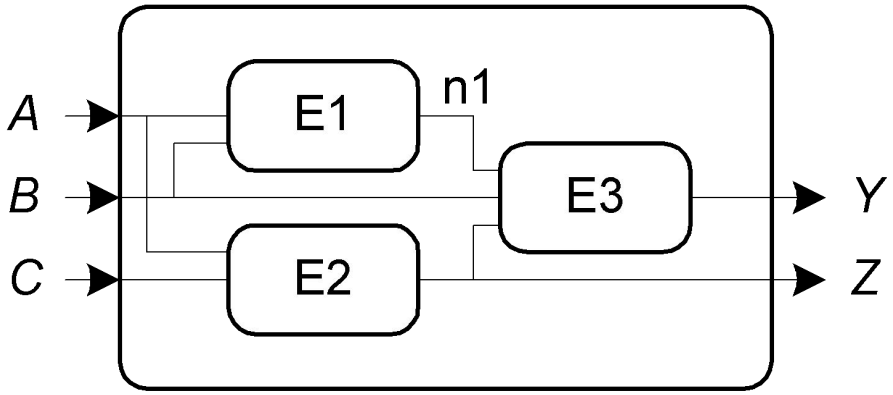
A logic circuit is composed of:

- Inputs
- Outputs
- Functional specification
- Timing specification



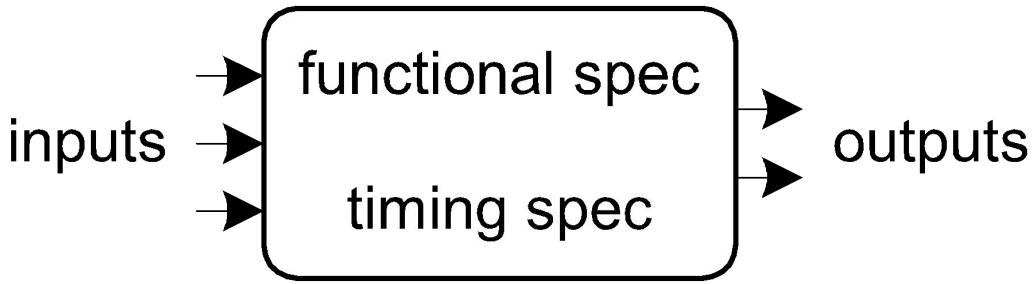
Circuits

- Nodes
  - Inputs: *A*, *B*, *C*
  - Outputs: *Y*, *Z*
  - Internal: n1
- Circuit elements
  - E1, E2, E3
  - Each a circuit



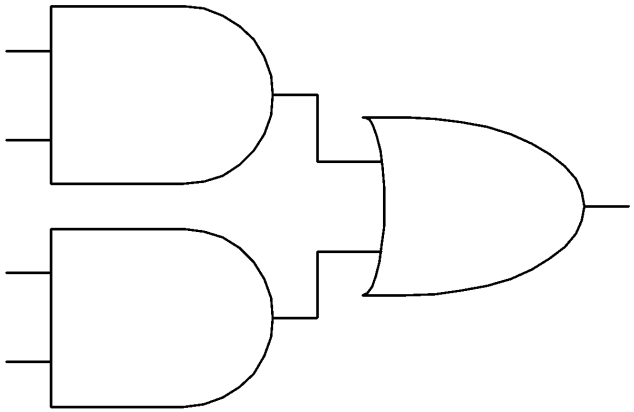
Types of Logic Circuits

- **Combinational Logic**
  - Memoryless
  - Outputs determined by current values of inputs
- **Sequential Logic**
  - Has memory
  - Outputs determined by previous and current values of inputs



Rules of Combinational Composition

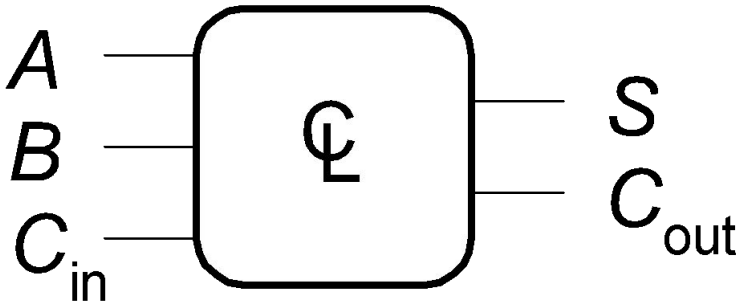
- Every element is combinational
- Every node is either an input or connects to *exactly one* output
- The circuit contains no cyclic paths
- **Example:**



Boolean Equations

- Functional specification of outputs in terms of inputs
- **Example:** $S = F(A, B, C_{in})$

  $C_{out} = F(A, B, C_{in})$



$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

Some Definitions

- Complement: variable with a bar over it
  $\bar{A}, \bar{B}, \bar{C}$
- Literal: variable or its complement
  $A, \bar{A}, B, \bar{B}, C, \bar{C}$
- Implicant: product of literals
  $AB\bar{C}, \bar{A}C, BC$
- Minterm: product that includes all input variables
  $AB\bar{C}, A\bar{B}C, ABC$
- Maxterm: sum that includes all input variables
  $(A+\bar{B}+C), (\bar{A}+B+\bar{C}), (\bar{A}+\bar{B}+C)$

Sum-of-Products (SOP) Form

- All equations can be written in SOP form
- Each row has a **minterm**
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)
- Form function by ORing minterms where the output is TRUE
- Thus, a sum (OR) of products (AND terms)

A	*B*	*Y*	minterm	minterm name
0	0	0	$\overline{A}\ \overline{B}$	m_0
0	1	1	$\overline{A}\ B$	m_1
1	0	0	$A\ \overline{B}$	m_2
1	1	1	$A\ B$	m_3

$$Y = F(A, B) =$$

Sum-of-Products (SOP) Form

- All equations can be written in SOP form
- Each row has a **minterm**
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)
- Form function by ORing minterms where the output is TRUE
- Thus, a sum (OR) of products (AND terms)

A	*B*	*Y*	minterm	minterm name
0	0	0	$\overline{A}\ \overline{B}$	m_0
0	1	1	$\overline{A}\ B$	m_1
1	0	0	$A\ \overline{B}$	m_2
1	1	1	$A\ B$	m_3

$$Y = F(A, B) =$$

Sum-of-Products (SOP) Form

- All equations can be written in SOP form
- Each row has a **minterm**
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)
- Form function by ORing minterms where the output is TRUE
- Thus, a sum (OR) of products (AND terms)

A	B	Y	minterm	minterm name
0	0	0	$\overline{A}\,\overline{B}$	m_0
0	1	1	$\overline{A}\,B$	m_1
1	0	0	$A\,\overline{B}$	m_2
1	1	1	$A\,B$	m_3

$$Y = F(A, B) = \overline{A}B + AB = \Sigma(1, 3)$$

Product-of-Sums (POS) Form

- All Boolean equations can be written in POS form
- Each row has a **maxterm**
- A maxterm is a sum (OR) of literals
- Each maxterm is FALSE for that row (and only that row)
- Form function by ANDing the maxterms for which the output is FALSE
- Thus, a product (AND) of sums (OR terms)

A	B	Y	maxterm	maxterm name
0	0	0	$A + B$	M_0
0	1	1	$A + \overline{B}$	M_1
1	0	0	$\overline{A} + B$	M_2
1	1	1	$\overline{A} + \overline{B}$	M_3

$$Y = F(A, B) = (A + B)(A + \overline{B}) = \Pi(0, 2)$$

Boolean Equations Example

- You are going to the cafeteria for lunch
  - You won't eat lunch ($\overline{E}$)
  - If it's not open ($\overline{O}$) or
  - If they only serve corndogs (C)
- Write a truth table for determining if you will eat lunch (E).

O	C	E
0	0	
0	1	
1	0	
1	1	

Boolean Equations Example

- You are going to the cafeteria for lunch
  - You won't eat lunch ($\overline{E}$)
  - If it's not open ($\overline{O}$) or
  - If they only serve corndogs (C)
- Write a truth table for determining if you will eat lunch (E).

O	C	E
0	0	0
0	1	0
1	0	1
1	1	0

SOP & POS Form

- SOP – sum-of-products

O	C	E	minterm
0	0		$\overline{O}\,\overline{C}$
0	1		$\overline{O}\,C$
1	0		$O\,\overline{C}$
1	1		$O\,C$

- POS – product-of-sums

O	C	E	maxterm
0	0		$O + C$
0	1		$O + \overline{C}$
1	0		$\overline{O} + C$
1	1		$\overline{O} + \overline{C}$

SOP & POS Form

- SOP – sum-of-products

O	C	E	minterm
0	0	0	$\overline{O}\ \overline{C}$
0	1	0	$\overline{O}\ C$
1	0	1	$O\ \overline{C}$
1	1	0	$O\ C$

$$E = O\overline{C}$$
$$= \Sigma(2)$$

- POS – product-of-sums

O	C	E	maxterm
0	0	0	$O + C$
0	1	0	$O + \overline{C}$
1	0	1	$\overline{O} + C$
1	1	0	$\overline{O} + \overline{C}$

$$E = (O + C)(O + \overline{C})(\overline{O} + \overline{C})$$
$$= \Pi(0, 1, 3)$$

Boolean Algebra

- Axioms and theorems to **simplify** Boolean equations
- Like regular algebra, but simpler: variables have only two values (1 or 0)
- **Duality** in axioms and theorems:
  - ANDs and ORs, 0's and 1's interchanged

Boolean Axioms

	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

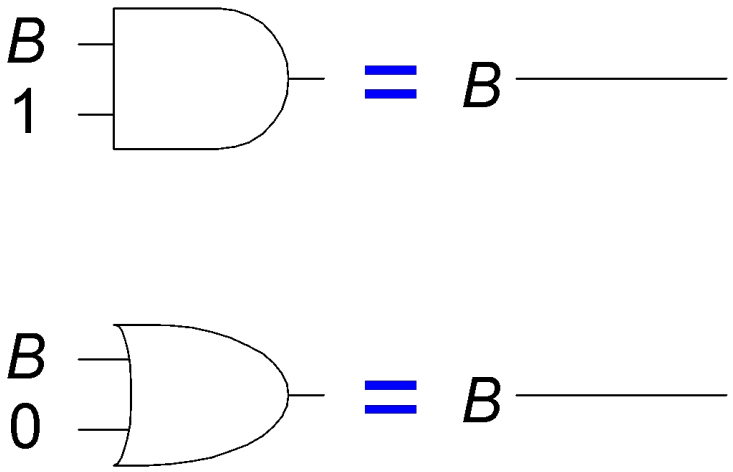
	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

T1: Identity Theorem

- $B \cdot 1 = B$
- $B + 0 = B$

T1: Identity Theorem

- $B \cdot 1 = B$
- $B + 0 = B$

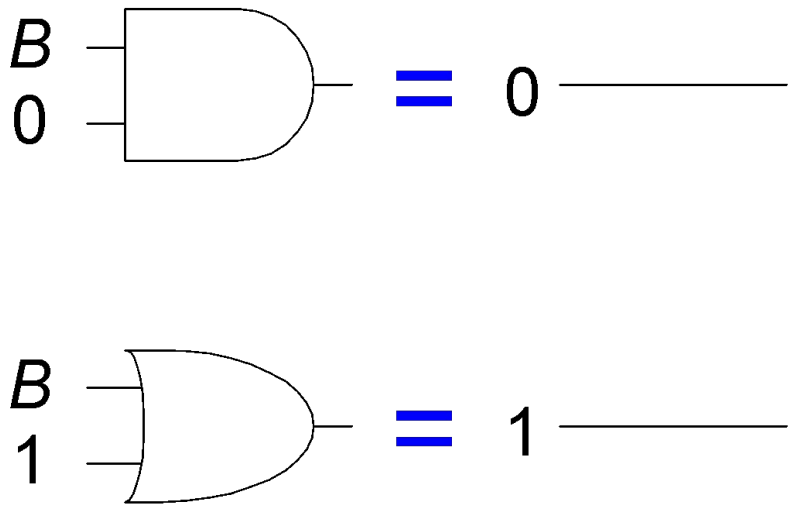


T2: Null Element Theorem

- $B \cdot 0 = 0$
- $B + 1 = 1$

T2: Null Element Theorem

- $B \cdot 0 = 0$
- $B + 1 = 1$

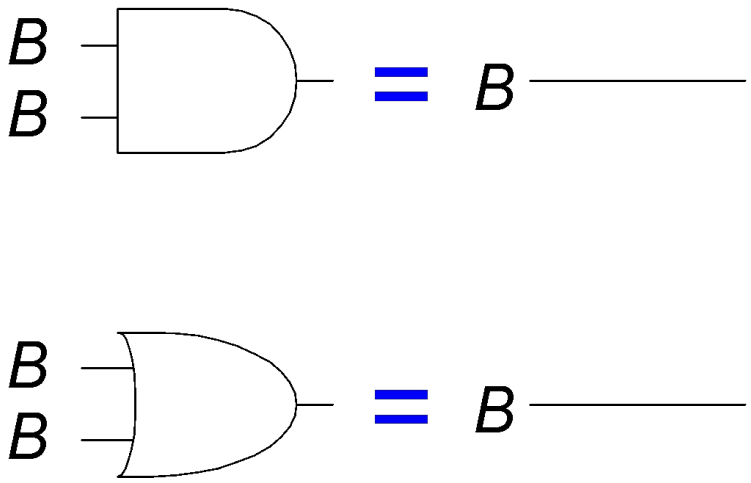


T3: Idempotency Theorem

- $B \cdot B = B$
- $B + B = B$

T3: Idempotency Theorem

- $B \cdot B = B$
- $B + B = B$

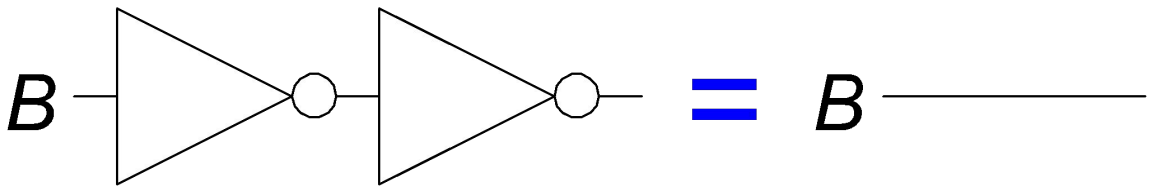


T4: Identity Theorem

- $\overline{\overline{B}} = B$

T4: Identity Theorem

- $\overline{\overline{B}} = B$

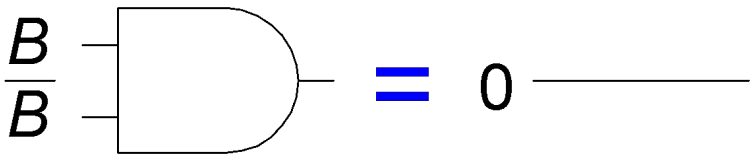


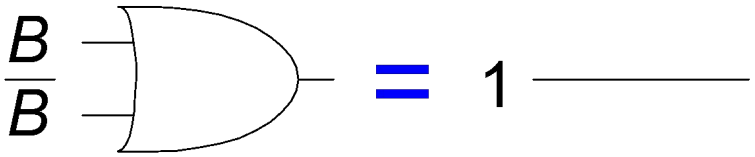
T5: Complement Theorem

- $B \cdot \overline{B} = 0$
- $B + \overline{B} = 1$

T5: Complement Theorem

- $B \cdot \overline{B} = 0$
- $B + \overline{B} = 1$





Boolean Theorems Summary

	Theorem		Dual	Name
T1	$B \cdot 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \cdot 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \cdot B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \cdot \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

Boolean Theorems of Several Vars

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D) = B \bullet C + \overline{B} \bullet D$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D) = (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots} = (\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots} = (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2})$	De Morgan's Theorem

Note: T8' differs from traditional algebra: OR (+) distributes over AND (•)

Simplifying Boolean Equations

Example 1:

$$Y = AB + \overline{A}B$$

Simplifying Boolean Equations

Example 1:

$$Y = AB + A\overline{B}$$

$$= B(A + \overline{A}) \quad \text{T8}$$

$$= B(1) \quad \text{T5'}$$

$$= B \quad \text{T1}$$

Simplifying Boolean Equations

Example 2:

$Y = A(AB + ABC)$

Simplifying Boolean Equations

Example 2:

$Y = A(AB + ABC)$

$= A(AB(1 + C))$ T8

$= A(AB(1))$ T2’

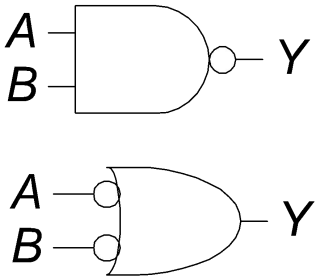
$= A(AB)$ T1

$= (AA)B$ T7

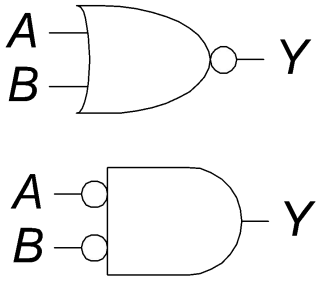
$= AB$ T3

DeMorgan's Theorem

- $Y = \overline{AB} = \overline{A} + \overline{B}$

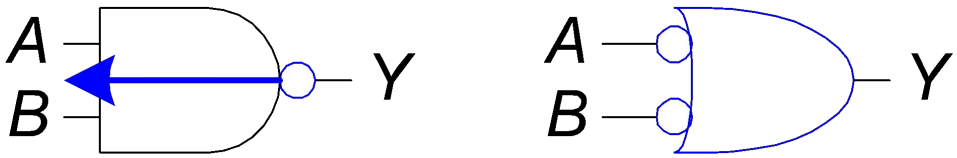


- $Y = \overline{A + B} = \overline{A} \cdot \overline{B}$

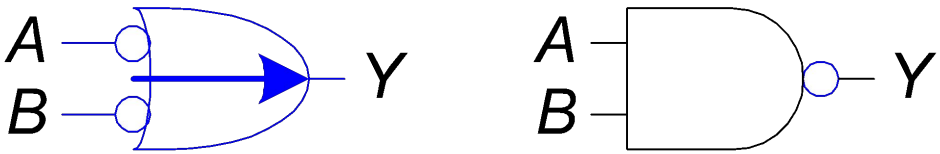


Bubble Pushing

- **Backward:**
  - Body changes
  - Adds bubbles to inputs

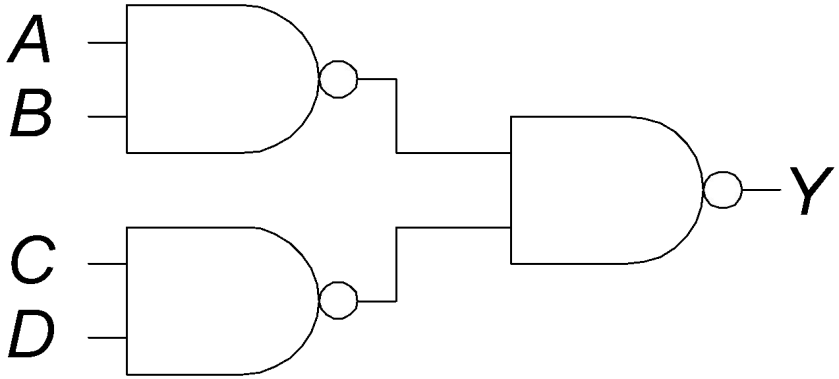


- **Forward:**
  - Body changes
  - Adds bubble to output



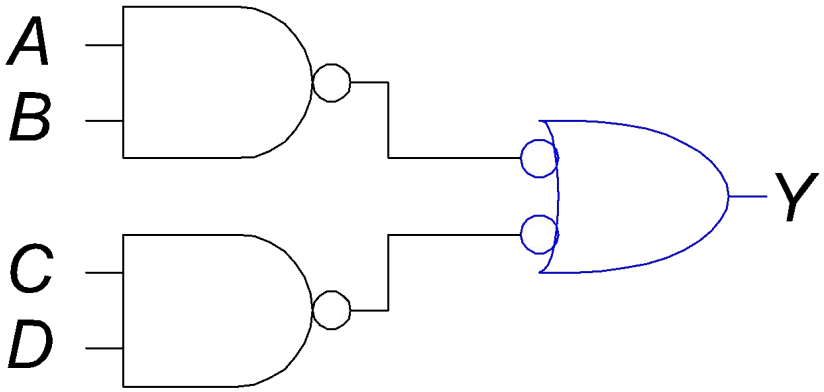
Bubble Pushing

- What is the Boolean expression for this circuit?



Bubble Pushing

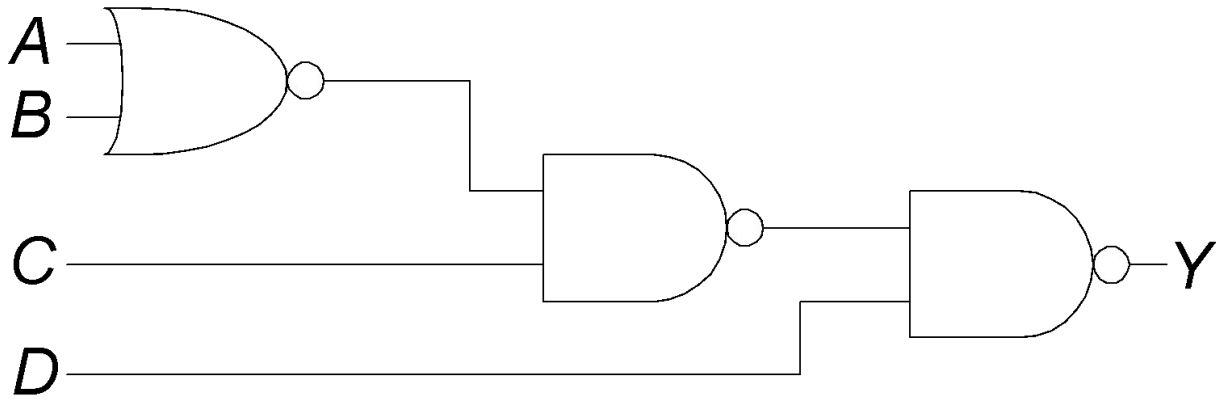
- What is the Boolean expression for this circuit?



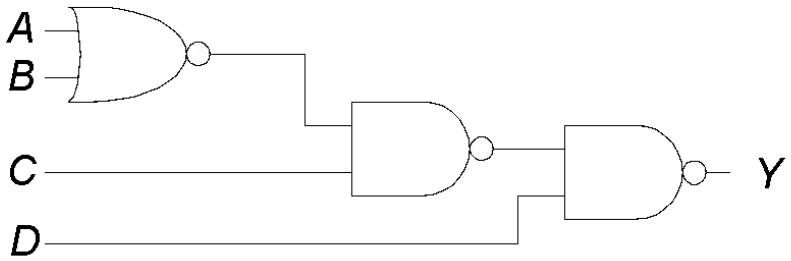
$$Y = AB + CD$$

Bubble Pushing Rules

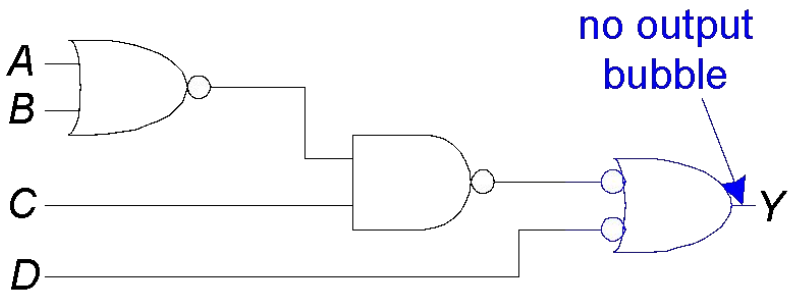
- Begin at output, then work toward inputs
- Push bubbles on final output back
- Draw gates in a form so bubbles cancel



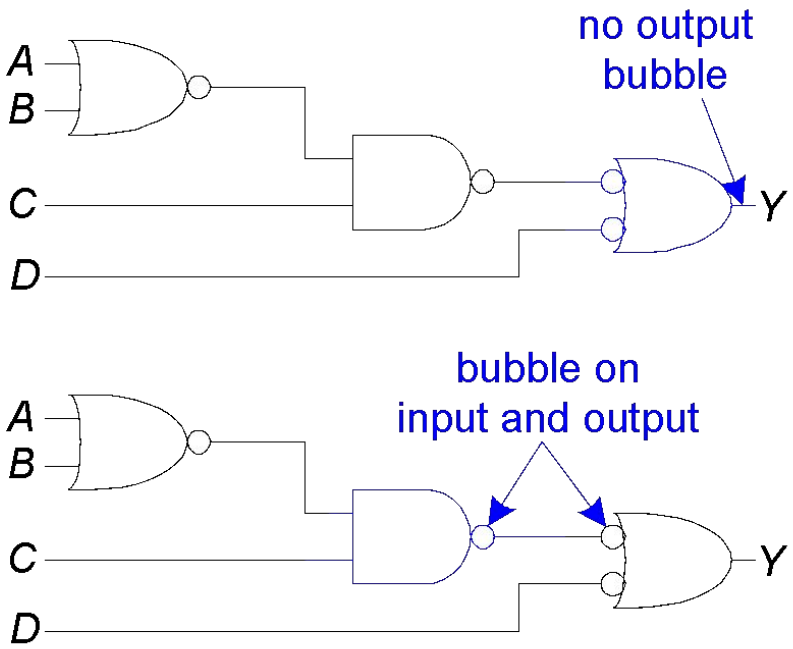
Bubble Pushing Example



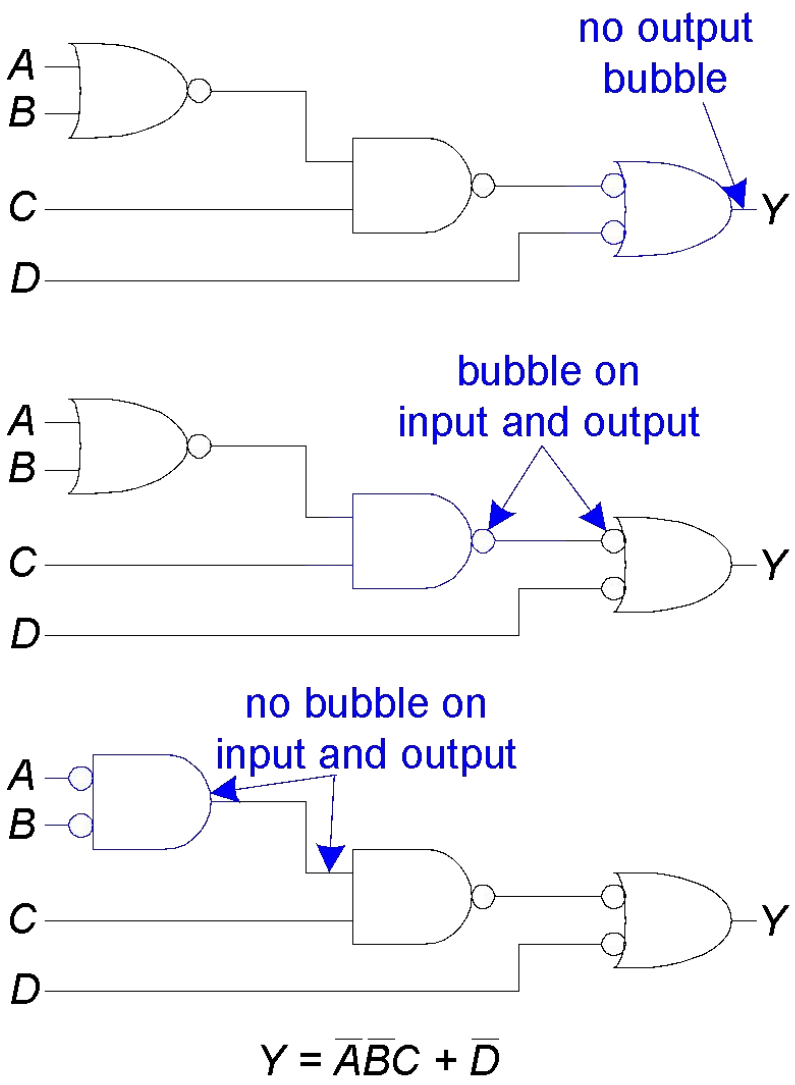
Bubble Pushing Example



Bubble Pushing Example



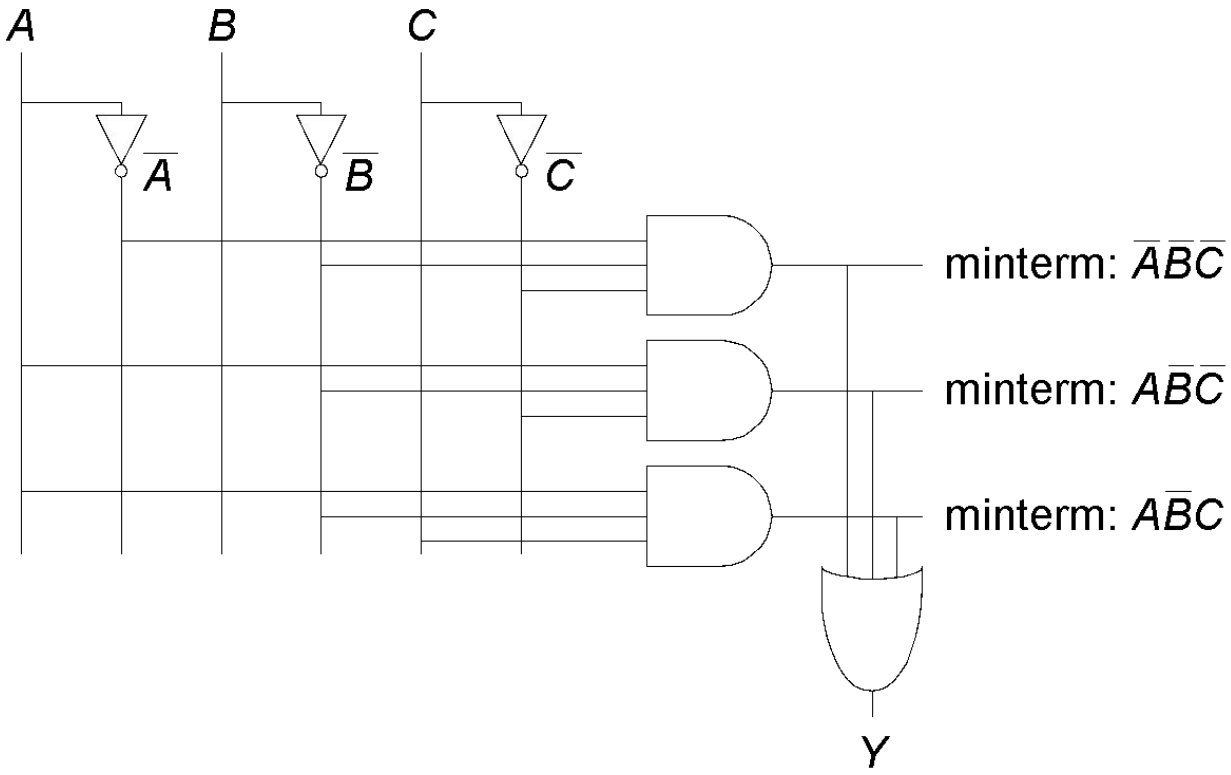
Bubble Pushing Example



$$Y = \overline{A}\overline{B}C + \overline{D}$$

From Logic to Gates

- Two-level logic: ANDs followed by ORs
- Example: $Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$

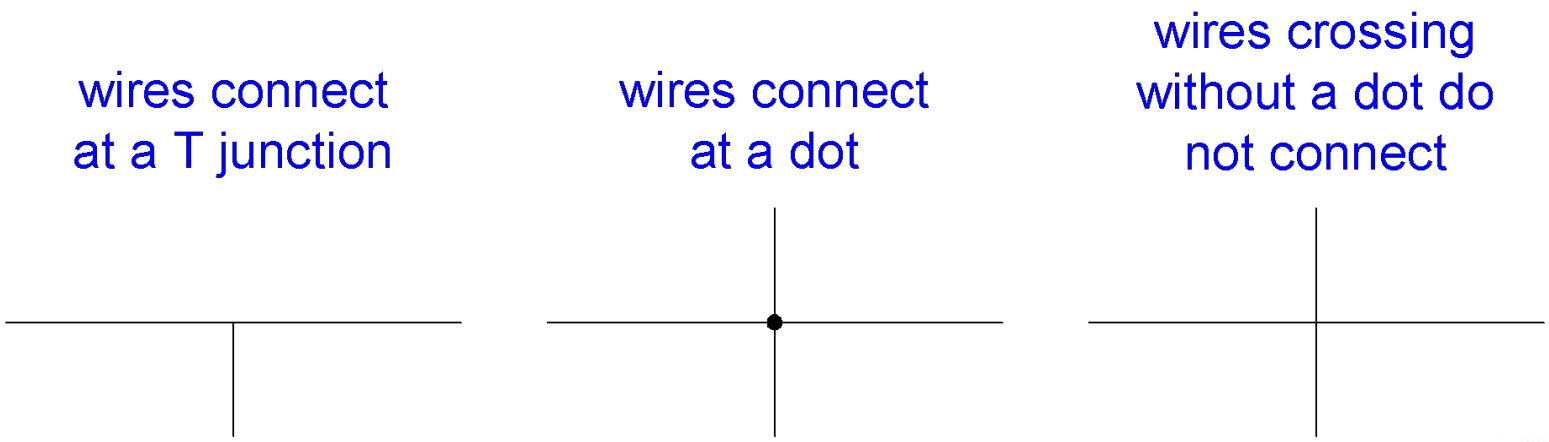


Circuit Schematics Rules

- Inputs on the left (or top)
- Outputs on right (or bottom)
- Gates flow from left to right
- Straight wires are best

Circuit Schematic Rules (cont.)

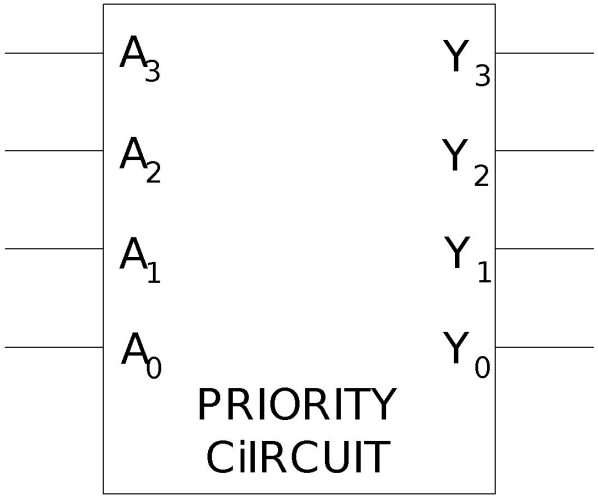
- Wires always connect at a T junction
- A dot where wires cross indicates a connection between the wires
- Wires crossing *without* a dot make no connection



Multiple-Output Circuits

- **Example: Priority Circuit**

  Output asserted corresponding to most significant TRUE input

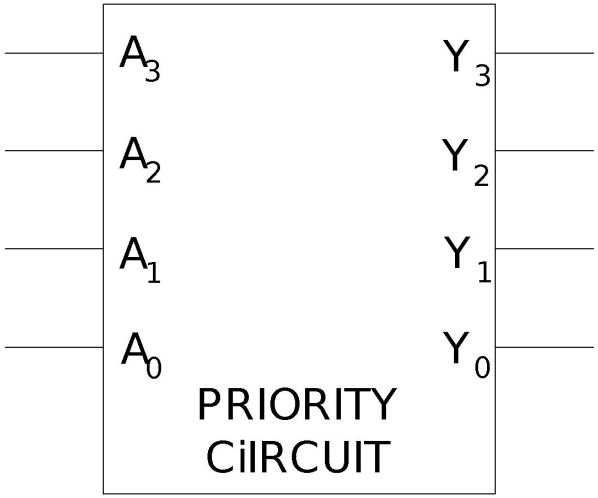


A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0				
0	0	0	1				
0	0	1	0				
0	0	1	1				
0	1	0	0				
0	1	0	1				
0	1	1	0				
0	1	1	1				
1	0	0	0				
1	0	0	1				
1	0	1	0				
1	0	1	1				
1	1	0	0				
1	1	0	1				
1	1	1	0				
1	1	1	1				

Multiple-Output Circuits

- **Example: Priority Circuit**

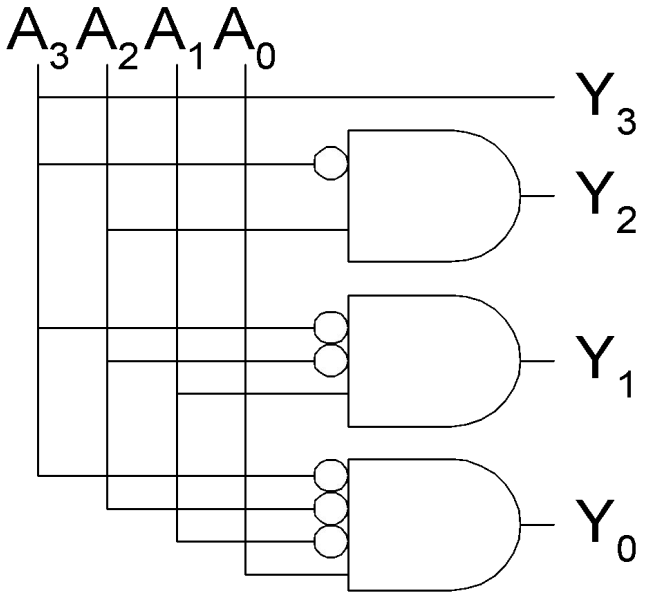
  Output asserted corresponding to most significant TRUE input



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	**0**	**0**	**0**	**0**
0	0	0	1	**0**	**0**	**0**	**1**
0	0	1	0	**0**	**0**	**1**	**0**
0	0	1	1	**0**	**0**	**1**	**0**
0	1	0	0	**0**	**1**	**0**	**0**
0	1	0	1	**0**	**1**	**0**	**0**
0	1	1	0	**0**	**1**	**0**	**0**
0	1	1	1	**0**	**1**	**0**	**0**
1	0	0	0	**1**	**0**	**0**	**0**
1	0	0	1	**1**	**0**	**0**	**0**
1	0	1	0	**1**	**0**	**0**	**0**
1	0	1	1	**1**	**0**	**0**	**0**
1	1	0	0	**1**	**0**	**0**	**0**
1	1	0	1	**1**	**0**	**0**	**0**
1	1	1	0	**1**	**0**	**0**	**0**
1	1	1	1	**1**	**0**	**0**	**0**

Priority Circuit Hardware

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	**0**	**0**	**0**	**0**
0	0	0	1	**0**	**0**	**0**	**1**
0	0	1	0	**0**	**0**	**1**	**0**
0	0	1	1	**0**	**0**	**1**	**0**
0	1	0	0	**0**	**1**	**0**	**0**
0	1	0	1	**0**	**1**	**0**	**0**
0	1	1	0	**0**	**1**	**0**	**0**
0	1	1	1	**0**	**1**	**0**	**0**
1	0	0	0	**1**	**0**	**0**	**0**
1	0	0	1	**1**	**0**	**0**	**0**
1	0	1	0	**1**	**0**	**0**	**0**
1	0	1	1	**1**	**0**	**0**	**0**
1	1	0	0	**1**	**0**	**0**	**0**
1	1	0	1	**1**	**0**	**0**	**0**
1	1	1	0	**1**	**0**	**0**	**0**
1	1	1	1	**1**	**0**	**0**	**0**

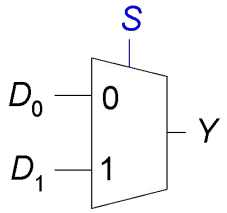


Combinational Building Blocks

- Multiplexers
- Decoders

Multiplexer (Mux)

- Selects between one of *N* inputs to connect to output
- $\log_2 N$-bit select input – control input
- **Example:** **2:1 Mux**

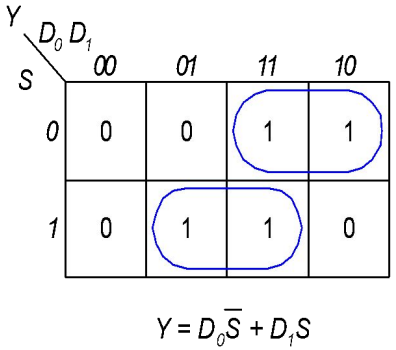


S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

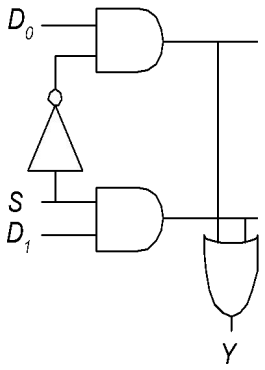
S	Y
0	D_0
1	D_1

Multiplexer Implementations

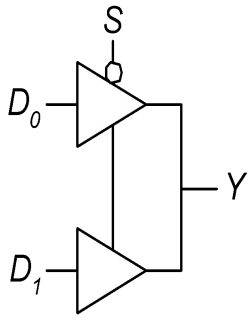
- **Logic gates**
  - Sum-of-products form



$Y = D_0\overline{S} + D_1S$



- **Tristates**
  - For an N-input mux, use N tristates
  - Turn on exactly one to select the appropriate input

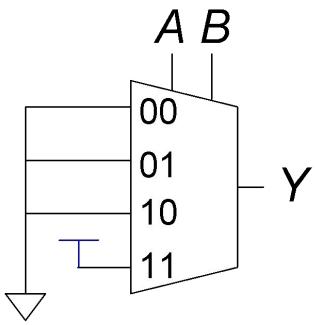


Logic using Multiplexers

- Using the mux as a lookup table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y = AB$



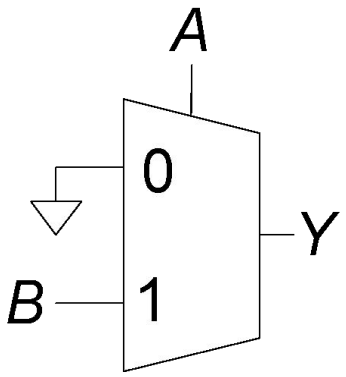
Logic using Multiplexers

- Reducing the size of the mux

$Y = AB$

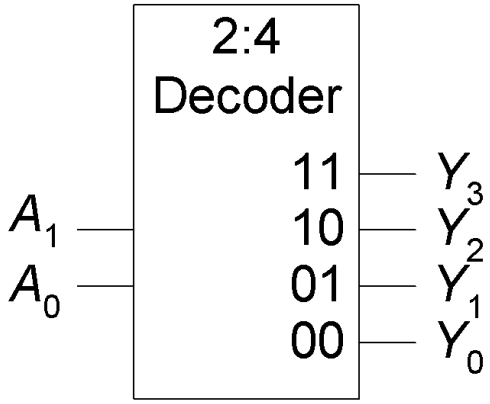
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

A	Y
0	0
1	B



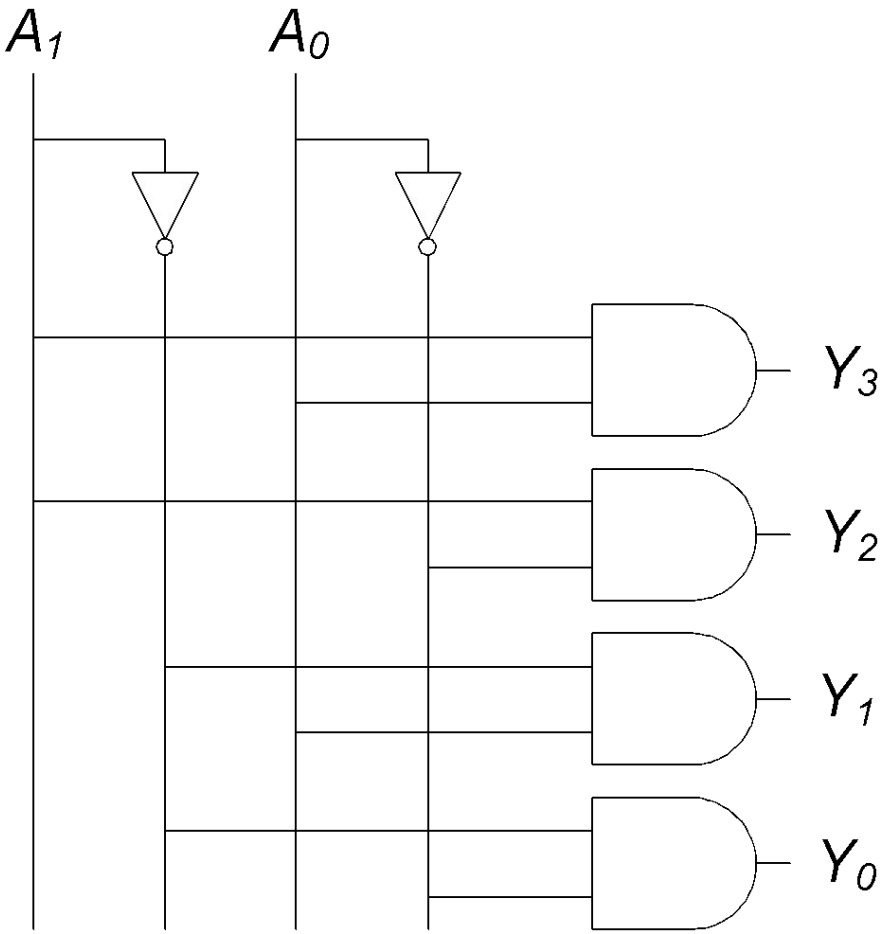
Decoders

- N inputs, 2^N outputs
- One-hot outputs: only one output HIGH at once



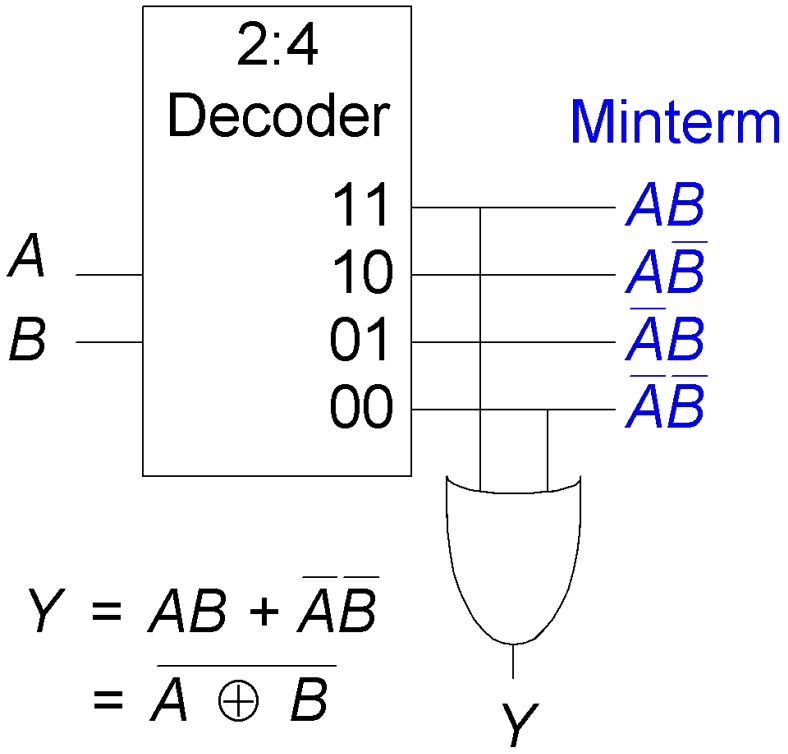
A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Decoder Implementation



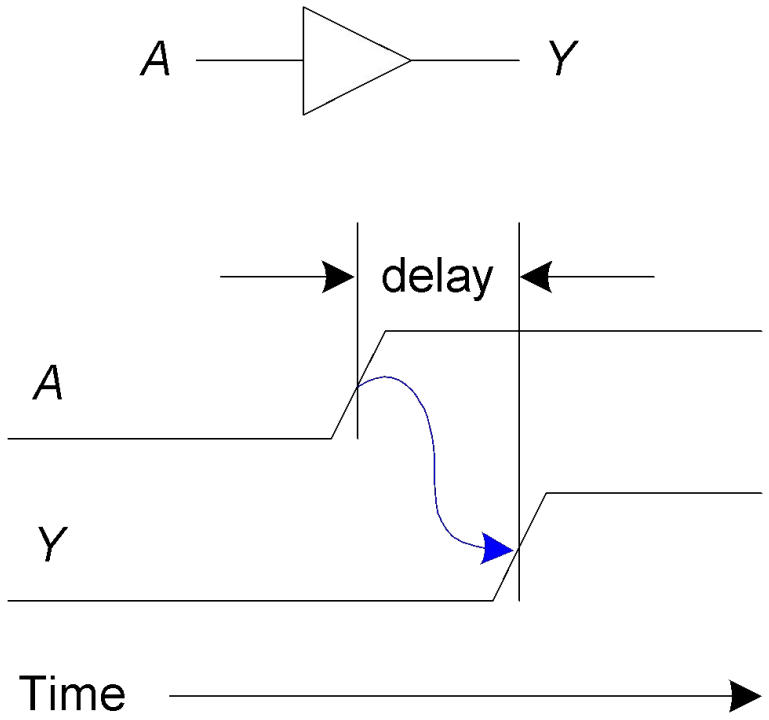
Logic Using Decoders

- OR minterms



Timing

- Delay between input change and output changing
- How to build fast circuits?



Introduction

- Outputs of sequential logic depend on current *and* prior input values – it has ***memory***.
- Some definitions:
  - **State:** all the information about a circuit necessary to explain its future behavior
  - **Latches and flip-flops:** state elements that store one bit of state
  - **Synchronous sequential circuits:** combinational logic followed by a bank of flip-flops

Sequential Circuits

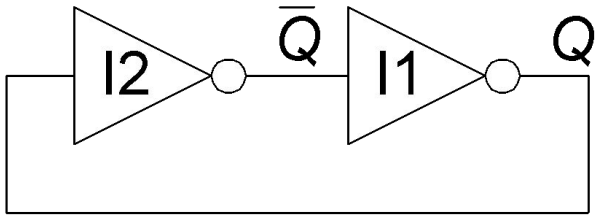
- Give sequence to events
- Have memory (short-term)
- Use feedback from output to input to store information

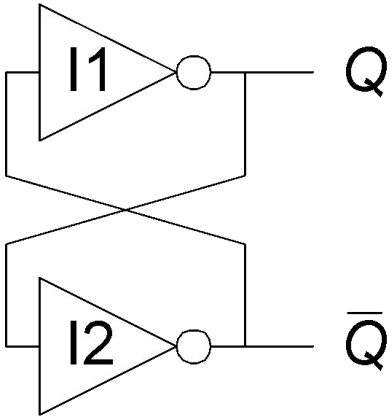
State Elements

- The state of a circuit influences its future behavior
- State elements store state
  - Bistable circuit
  - SR Latch
  - D Latch
  - D Flip-flop

Bistable Circuit

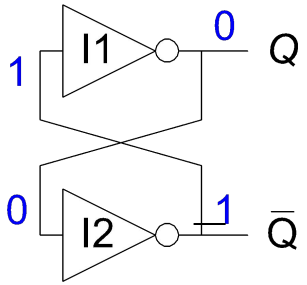
- Fundamental building block of other state elements
- Two outputs: $Q, \overline{Q}$
- No inputs



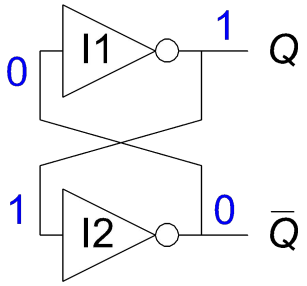


Bistable Circuit Analysis

- Consider the two possible cases:
  - $Q = 0$:
    then $\bar{Q} = 1$, $Q = 0$ (consistent)



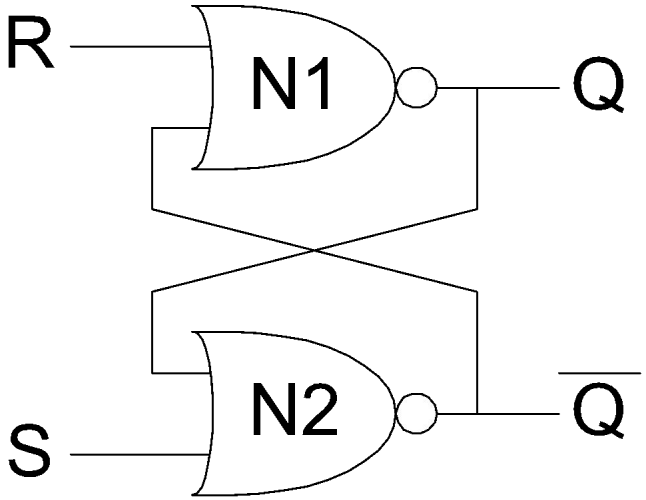
  - $Q = 1$:
    then $\bar{Q} = 0$, $Q = 1$ (consistent)



- Stores 1 bit of state in the state variable, Q (or $\bar{Q}$)
- But there are **no inputs to control the state**

SR (Set/Reset) Latch

- SR Latch

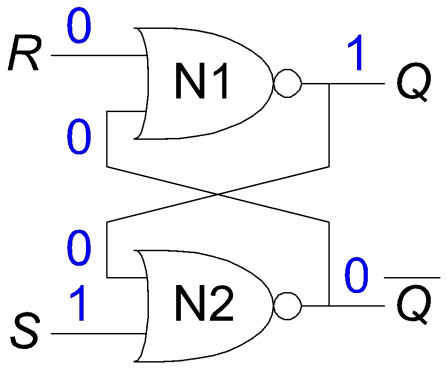


- Consider the four possible cases:
  - $S = 1, R = 0$
  - $S = 0, R = 1$
  - $S = 0, R = 0$
  - $S = 1, R = 1$

SR Latch Analysis

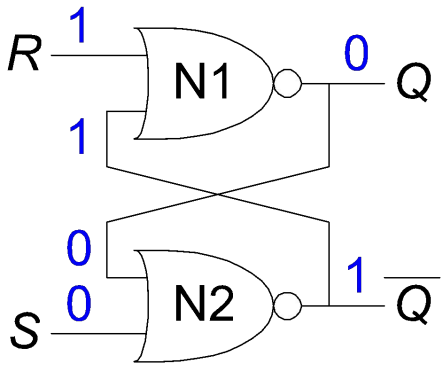
- $S = 1, R = 0$:

  then $Q = 1$ and $\bar{Q} = 0$



- $S = 0, R = 1$:

  then $Q = 1$ and $\bar{Q} = 0$

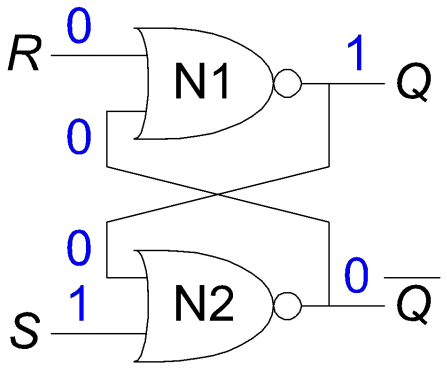


SR Latch Analysis

- $S = 1, R = 0$:

  then $Q = 1$ and $\bar{Q} = 0$

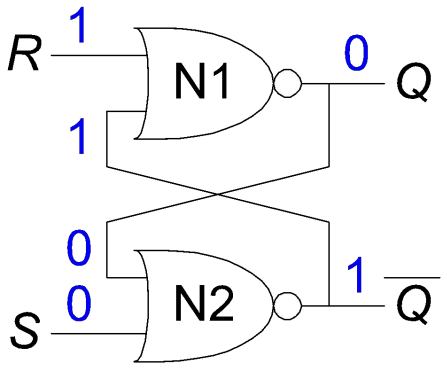
  ***Set* the output**



- $S = 0, R = 1$:

  then $Q = 1$ and $\bar{Q} = 0$

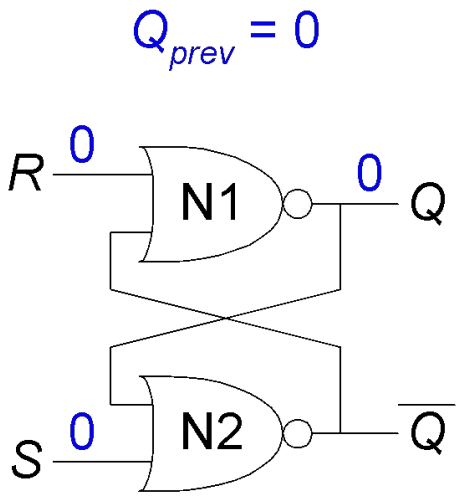
  ***Reset* the output**

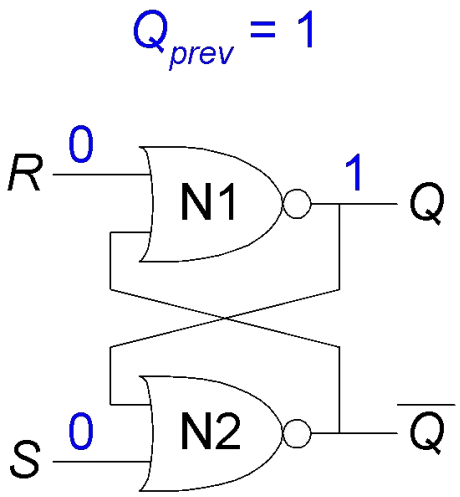


SR Latch Analysis

– $S = 0, R = 0$:

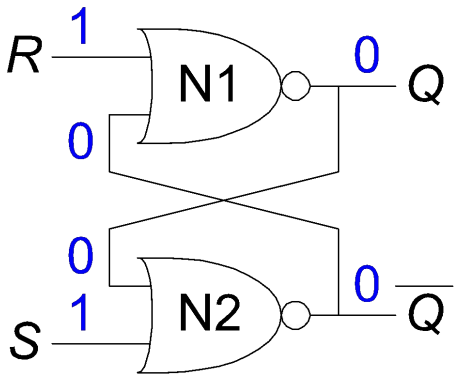
then $Q = Q_{prev}$





– $S = 1, R = 1$:

then $Q = 0, Q = 0$

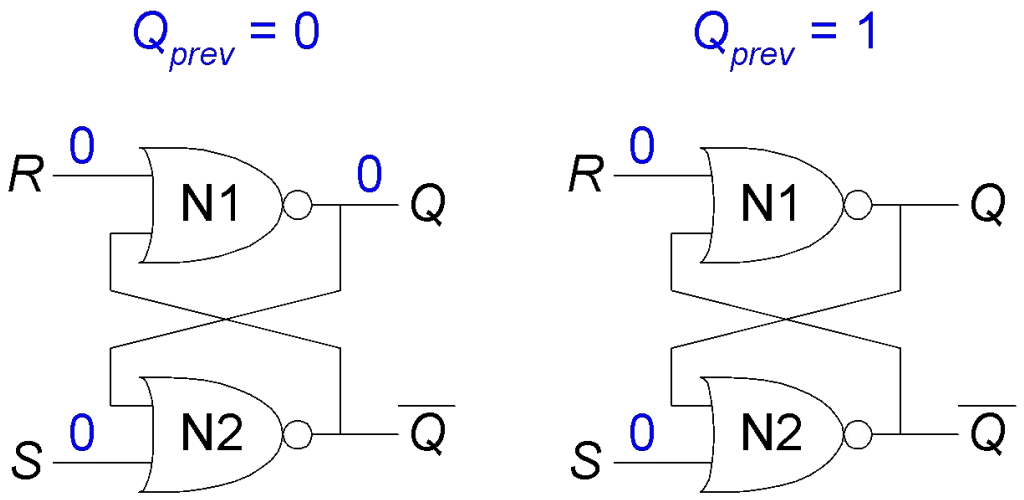


SR Latch Analysis

– $S = 0, R = 0$:

then $Q = Q_{prev}$

Memory!



– $S = 1, R = 1$:

then $Q = 0, Q = 0$

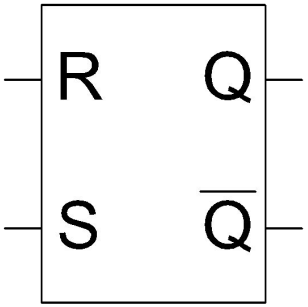
Invalid State

$Q \neq$ NOT Q

SR Latch Symbol

- SR stands for Set/Reset Latch
  - Stores one bit of state (Q)
- Control what value is being stored with S, R inputs
  - **Set:** Make the output 1 ($S = 1, R = 0, Q = \mathbf{1}$)
  - **Reset:** Make the output 0 ($S = 0, R = 1, Q = \mathbf{0}$)

SR Latch Symbol



D Latch

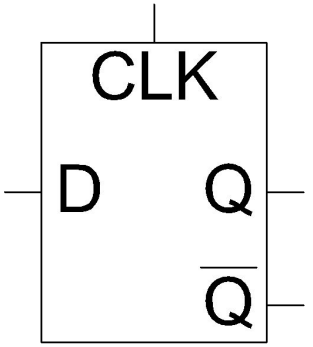
- Two inputs: *CLK*, *D*
  - ***CLK*:** controls *when* the output changes
  - ***D*** (the data input): controls *what* the output changes to
- Function
  - When ***CLK* = 1**,

    D passes through to *Q* (*transparent*)
  - When ***CLK* = 0**,

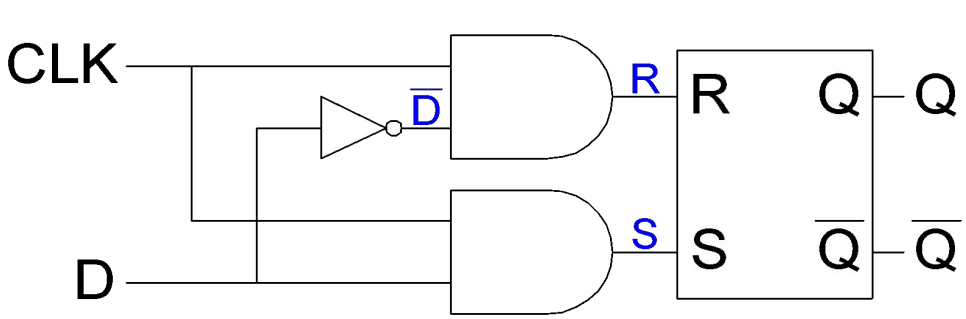
    Q holds its previous value (*opaque*)
- Avoids invalid case when

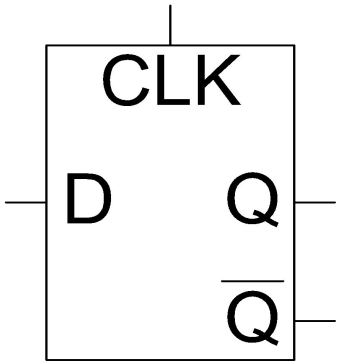
  $Q \neq \text{NOT } Q$

D Latch Symbol



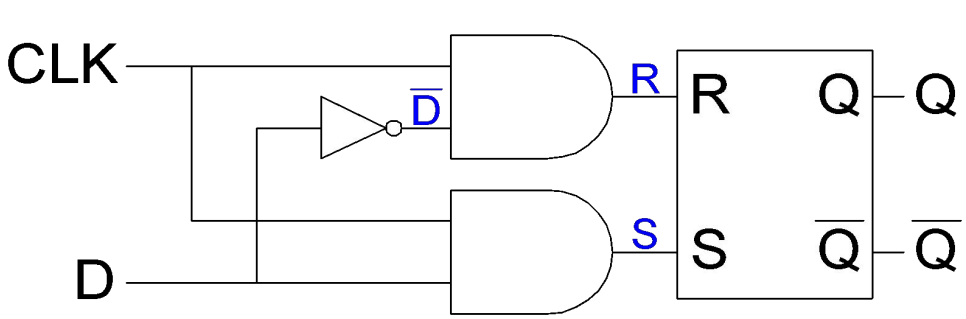
D Latch Internal Circuit

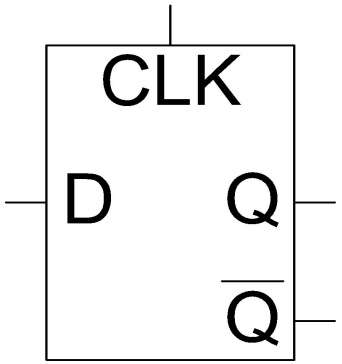




CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X					
1	0					
1	1					

D Latch Internal Circuit



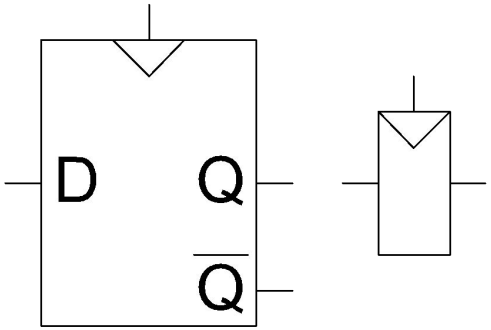


CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Q_{prev}	$\overline{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

D Flip-Flop

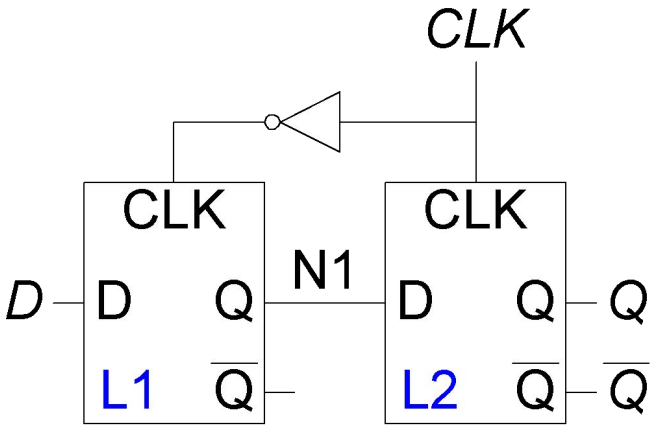
- **Inputs:** *CLK*, *D*
- **Function**
  - Samples *D* on rising edge of *CLK*
    - When *CLK* rises from 0 to 1, *D* passes through to *Q*
    - Otherwise, *Q* holds its previous value
  - *Q* changes only on rising edge of *CLK*
- Called *edge-triggered*
- Activated on the clock edge

D Flip-Flop Symbols

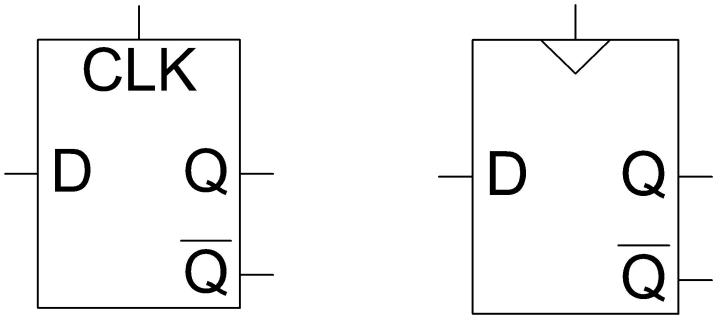


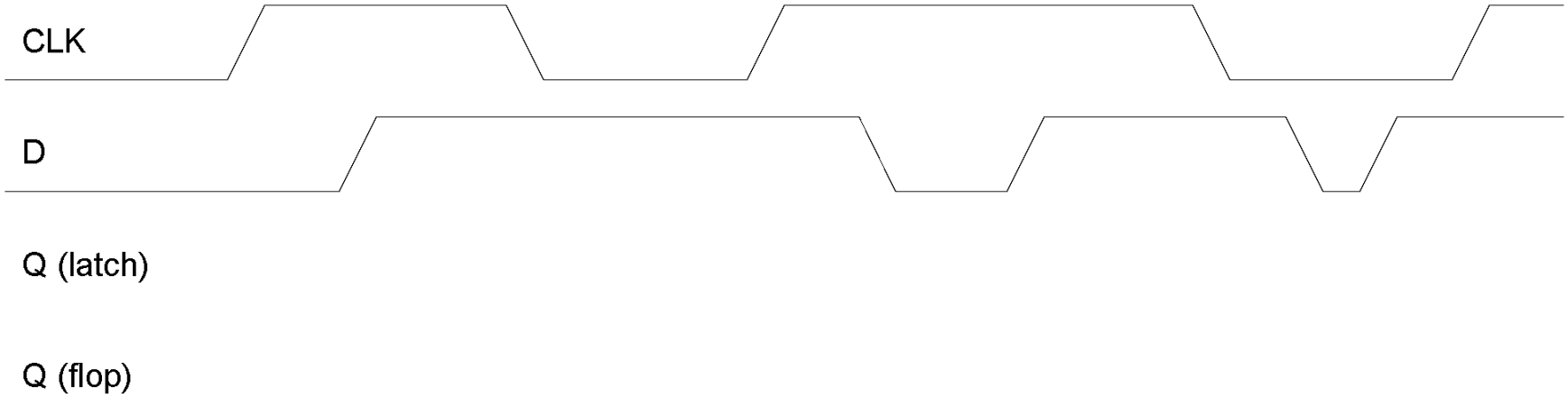
D Flip-Flop Internal Circuit

- Two back-to-back latches (L1 and L2) controlled by complementary clocks
- When ***CLK* = 0**
  - L1 is transparent
  - L2 is opaque
  - *D* passes through to N1
- When ***CLK* = 1**
  - L2 is transparent
  - L1 is opaque
  - N1 passes through to *Q*
- Thus, on the edge of the clock (when ***CLK* rises from 0 to 1**)
  - *D* passes through to *Q*

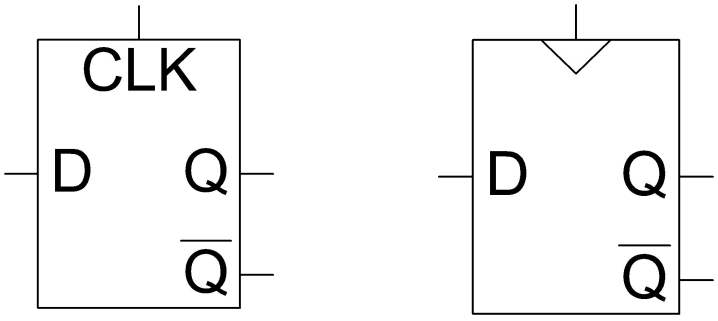


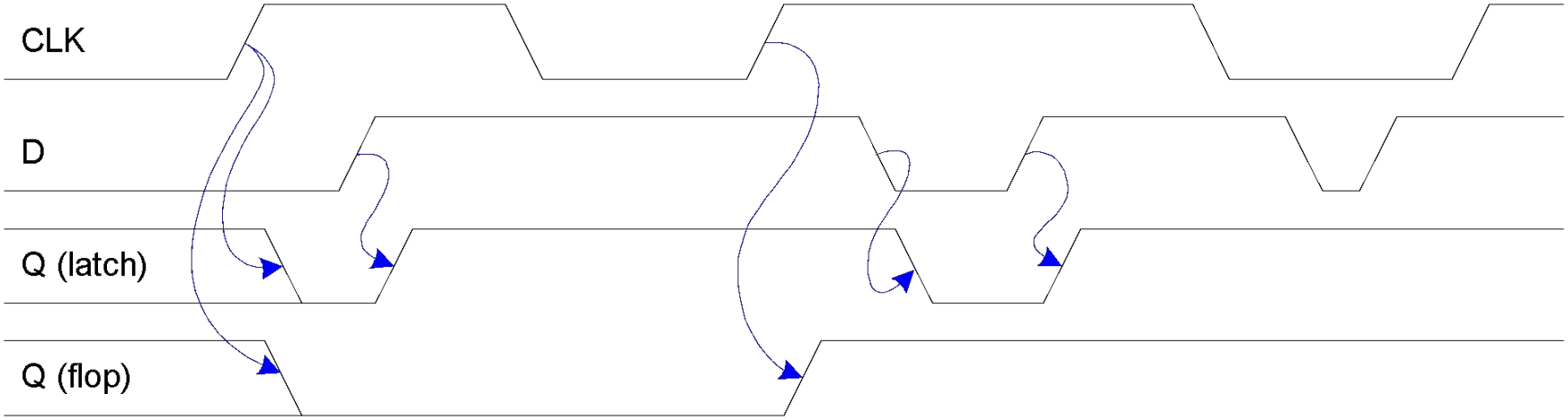
D Latch vs. D Flip-Flop



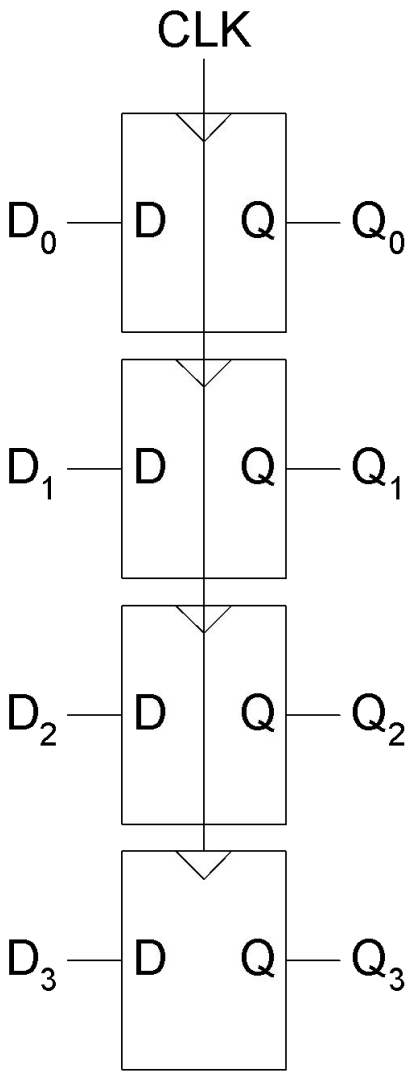


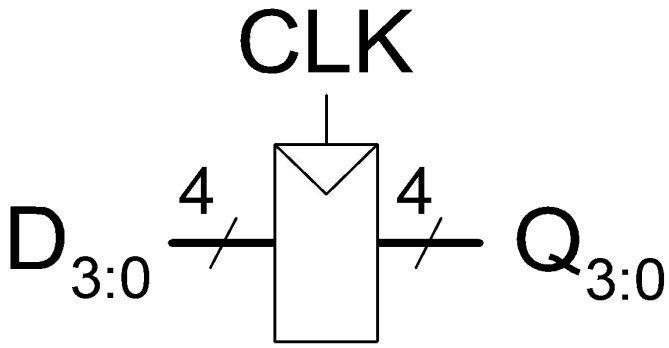
D Latch vs. D Flip-Flop





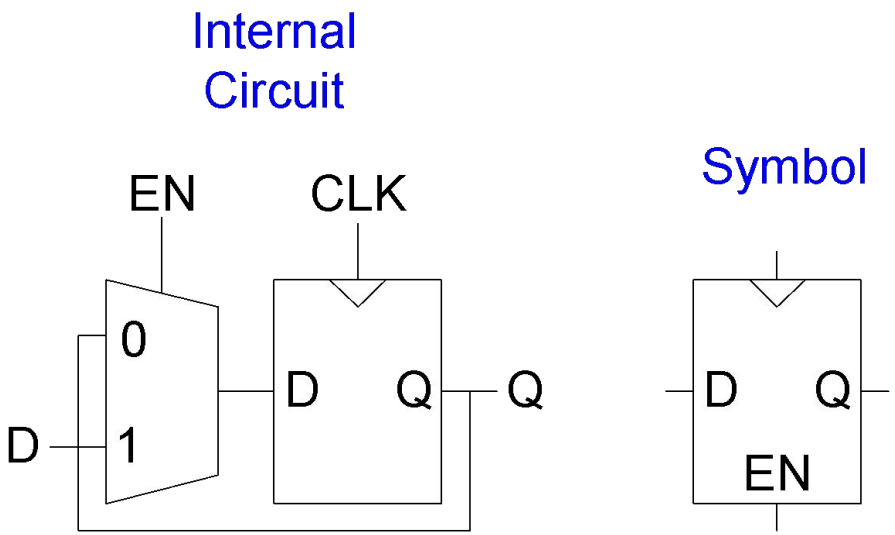
Registers





Enabled Flip-Flops

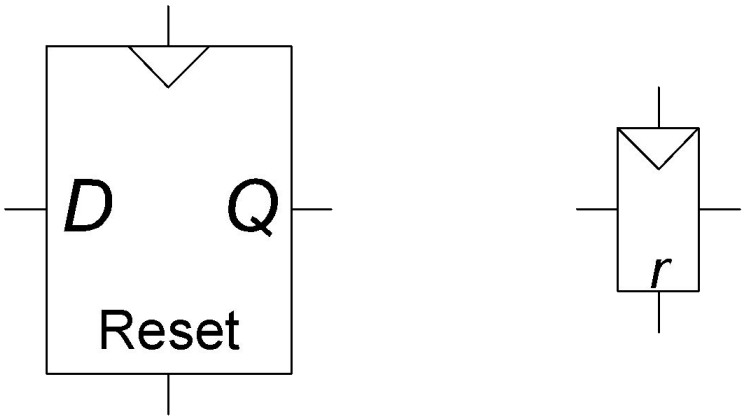
- **Inputs:** *CLK*, *D*, *EN*
  - The enable input (*EN*) controls when new data (*D*) is stored
- **Function**
  - ***EN* = 1:** *D* passes through to *Q* on the clock edge
  - ***EN* = 0:** the flip-flop retains its previous state



Resettable Flip-Flops

- **Inputs:** *CLK*, *D*, *Reset*
- **Function:**
  - ***Reset* = 1:** *Q* is forced to 0
  - ***Reset* = 0:** flip-flop behaves as ordinary D flip-flop

Symbols

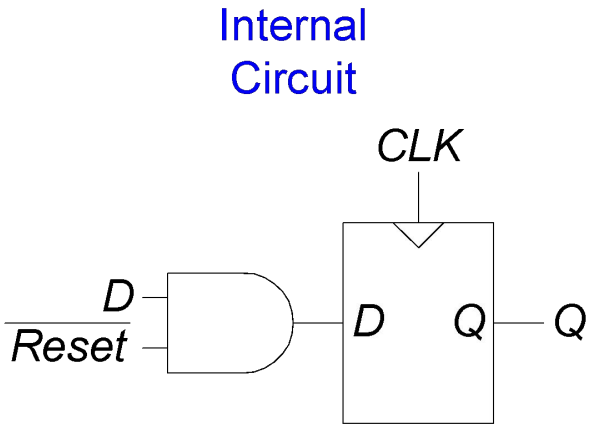


Resettable Flip-Flops

- Two types:
  - **Synchronous:** resets at the clock edge only
  - **Asynchronous:** resets immediately when *Reset* = 1
- Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop
- Synchronously resettable flip-flop?

Resettable Flip-Flops

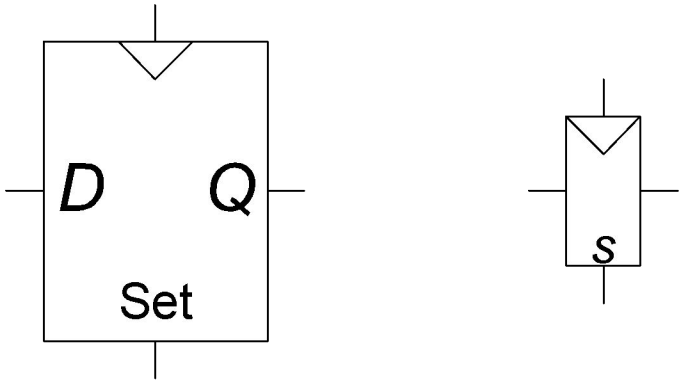
- Two types:
  - **Synchronous:** resets at the clock edge only
  - **Asynchronous:** resets immediately when *Reset* = 1
- Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop
- Synchronously resettable flip-flop?



Settable Flip-Flops

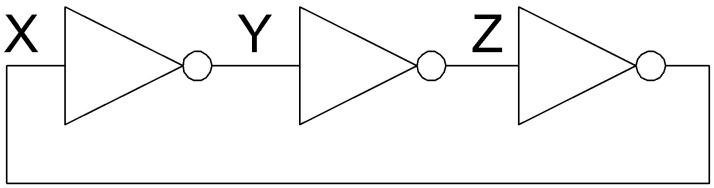
- **Inputs:** *CLK*, *D*, *Set*
- **Function:**
  - ***Set* = 1:** *Q* is set to 1
  - ***Set* = 0:** the flip-flop behaves as ordinary D flip-flop

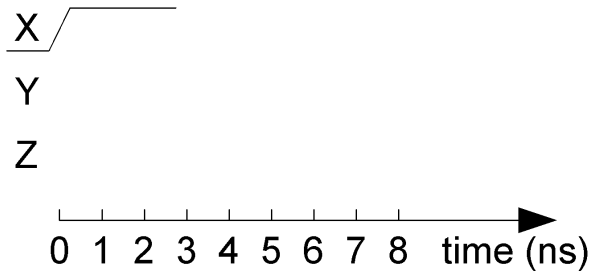
Symbols



Sequential Logic

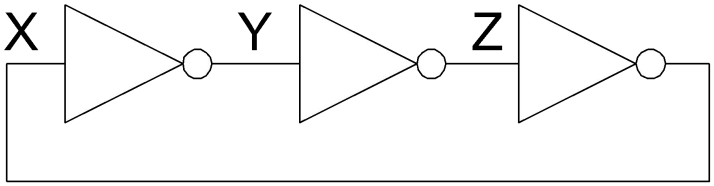
- Sequential circuits: all circuits that aren't combinational
- A problematic circuit:

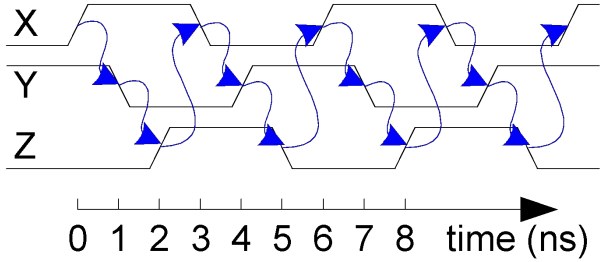




Sequential Logic

- Sequential circuits: all circuits that aren't combinational
- A problematic circuit:





- No inputs and 1-3 outputs
- Astable circuit, oscillates
- Period depends on inverter delay
- It has a ***cyclic path***: output fed back to input

Synchronous Sequential Logic Design

- Breaks cyclic paths by **inserting registers**
- Registers contain **state** of the system
- State changes at clock edge: system **synchronized** to the clock
- **Rules** of synchronous sequential circuit composition:
  - Every circuit element is either a register or a combinational circuit
  - At least one circuit element is a register
  - All registers receive the same clock signal
  - Every cyclic path contains at least one register
- Two common synchronous sequential circuits
  - Finite State Machines (FSMs)
  - Pipelines

Addition

- Decimal

$$\begin{array}{r} 3734 \\ +\ 5168 \\ \hline \end{array}$$

- Binary

$$\begin{array}{r} 1011 \\ +\ 0011 \\ \hline \end{array}$$

Addition

- Decimal

```
   11   ← carries
   3734
+  5168
------
   8902
```

- Binary

```
   11   ← carries
   1011
+  0011
------
   1110
```

Binary Addition Examples

- Add the following 4-bit binary numbers

$$\begin{array}{r} 1001 \\ +\ 0101 \\ \hline \end{array}$$

- Add the following 4-bit binary numbers

$$\begin{array}{r} 1011 \\ +\ 0110 \\ \hline \end{array}$$

Binary Addition Examples

- Add the following 4-bit binary numbers

```
   1
 1001
+0101
-----
 1110
```

- Add the following 4-bit binary numbers

```
 111
 1011
+0110
-----
10001
```

Overflow!

Overflow

- Digital systems operate on a **fixed number of bits**
- Overflow: when result is too big to fit in the available number of bits
- See previous example of 11 + 6

Signed Binary Numbers

- Sign/Magnitude Numbers
- Two's Complement Numbers

Two's Complement Numbers

- Don't have same problems as sign/magnitude numbers:
  - Addition works
  - Single representation for 0

Two's Complement Numbers

- Msb has value of -2^{N-1}

$$A = a_{n-1}\left(-2^{n-1}\right) + \sum_{i=0}^{n-2} a_i 2^i$$

- Most positive 4-bit number:
- Most negative 4-bit number:
- The most significant bit still indicates the sign (1 = negative, 0 = positive)
- Range of an *N*-bit two's comp number:

Two's Complement Numbers

- Msb has value of -2^{N-1}

$$A = a_{n-1}\left(-2^{n-1}\right) + \sum_{i=0}^{n-2} a_i 2^i$$

- Most positive 4-bit number: **0111**
- Most negative 4-bit number: **1000**
- The most significant bit still indicates the sign (1 = negative, 0 = positive)
- Range of an *N*-bit two's comp number:

$[-(2^{N-1}), 2^{N-1}-1]$

FROM ZERO TO ONE

"Taking the Two's Complement"

- Flip the sign of a two's complement number
- Method:
  1. Invert the bits
  2. Add 1
- Example: Flip the sign of $3_{10} = 0011_2$

"Taking the Two's Complement"

- Flip the sign of a two's complement number
- Method:
  1. Invert the bits
  2. Add 1
- Example: Flip the sign of $3_{10} = 0011_2$
  1. **1100**
  2. **+ 1**

  1101 = -3_{10}

Two's Complement Examples

- Take the two's complement of $6_{10} = 0110_2$

- What is the decimal value of 1001_2?

Two's Complement Examples

- Take the two's complement of $6_{10} = 0110_2$
  1. 1001
  2. + 1

  $1010_2 = -6_{10}$

- What is the decimal value of the two's complement number 1001_2?
  1. 0110
  2. + 1

  $0111_2 = 7_{10}$, so $1001_2 = -7_{10}$

Two's Complement Addition

- Add 6 + (-6) using two's complement numbers

$$\begin{array}{r} 0110 \\ +\ 1010 \\ \hline \end{array}$$

- Add -2 + 3 using two's complement numbers

$$\begin{array}{r} 1110 \\ +\ 0011 \\ \hline \end{array}$$

Two's Complement Addition

- Add 6 + (-6) using two's complement numbers

```
  111
   0110
+  1010
------
 10000
```

- Add -2 + 3 using two's complement numbers

```
  111
   1110
+  0011
------
 10001
```

Increasing Bit Width

- **Extend number from *N* to *M* bits (*M* > *N*) :**
  - Sign-extension
  - Zero-extension

Sign-Extension

- Sign bit copied to msb's
- Number value is same

- **Example 1:**
  - 4-bit representation of 3 = 0011
  - 8-bit sign-extended value: 00000011
- **Example 2:**
  - 4-bit representation of -5 = 1011
  - 8-bit sign-extended value: 11111011

Zero-Extension

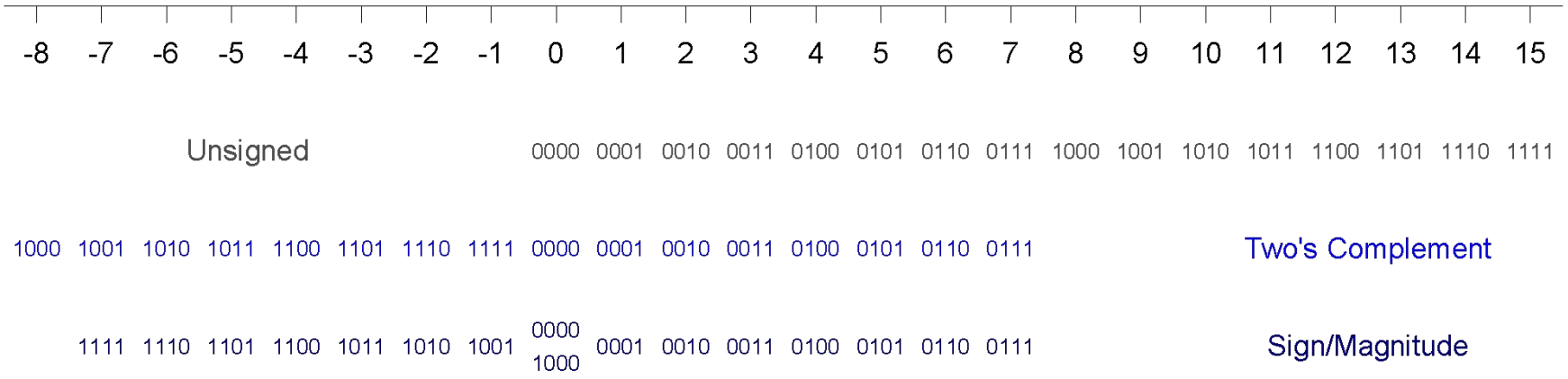
- Zeros copied to msb's
- Value changes for negative numbers

- **Example 1:**
  - 4-bit value = $0011_2 = 3_{10}$
  - 8-bit zero-extended value: $00000011 = 3_{10}$
- **Example 2:**
  - 4-bit value = $1011 = -5_{10}$
  - 8-bit zero-extended value: $00001011 = 11_{10}$

Number System Comparison

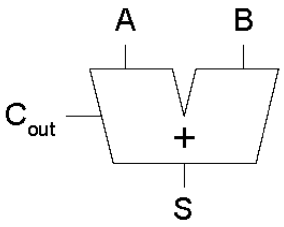
Number System	Range
Unsigned	$[0, 2^N-1]$
Sign/Magnitude	$[-(2^{N-1}-1), 2^{N-1}-1]$
Two's Complement	$[-2^{N-1}, 2^{N-1}-1]$

For example, 4-bit representation:



1-Bit Adders

Half Adder

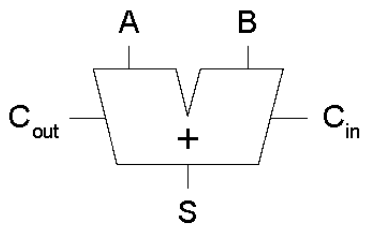


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

S =

C_{out} =

Full Adder



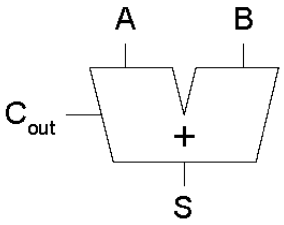
C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

S =

C_{out} =

1-Bit Adders

Half Adder

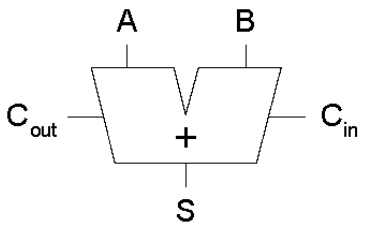


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S =

C_{out} =

Full Adder



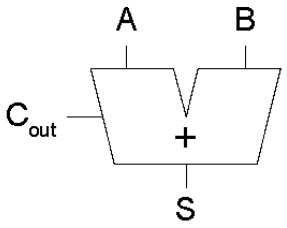
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S =

C_{out} =

1-Bit Adders

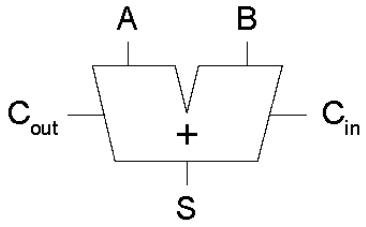
Half Adder



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$
$$C_{out} = AB$$

Full Adder



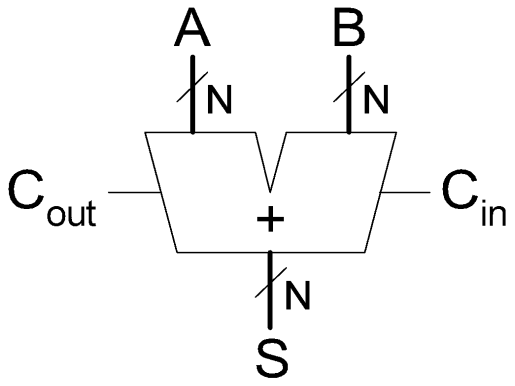
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Multibit Adders (CPAs)

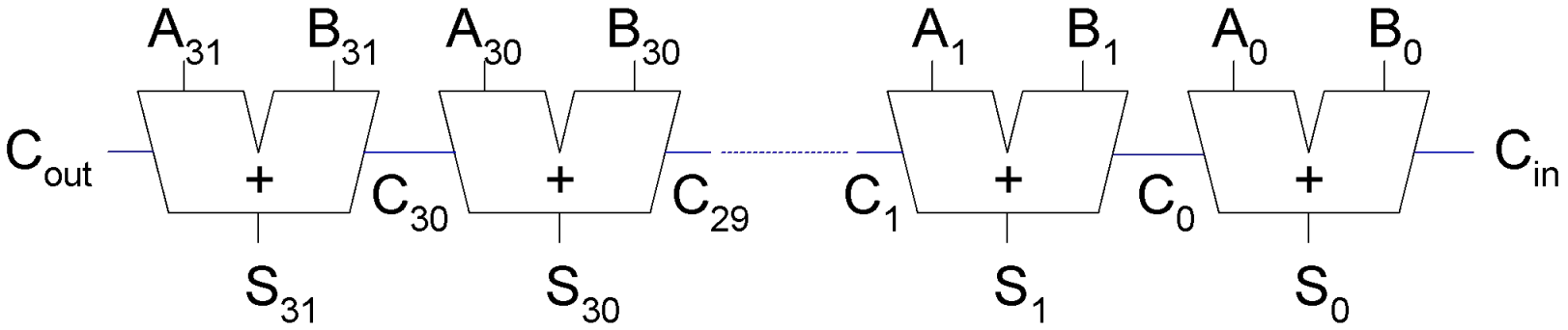
- Types of carry propagate adders (CPAs):
  - Ripple-carry (slow)
  - Carry-lookahead (fast)
  - Prefix (faster)
- Carry-lookahead and prefix adders faster for large adders but require more hardware

Symbol



Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**



Ripple-Carry Adder Delay

$$t_{\text{ripple}} = Nt_{FA}$$

where t_{FA} is the delay of a 1-bit full adder

Carry-Lookahead Adder

- Compute carry out (C_{out}) for k-bit blocks using *generate* and *propagate* signals
- **Some definitions:**
  - Column i produces a carry out by either *generating* a carry out or *propagating* a carry in to the carry out
  - Generate (G_i) and propagate (P_i) signals for each column:
    - Column i will generate a carry out if A_i AND B_i are both 1.

$$G_i = A_i B_i$$

    - Column i will propagate a carry in to the carry out if A_i OR B_i is 1.

$$P_i = A_i + B_i$$

    - The carry out of column i (C_i) is:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k-bit blocks
- **Step 3:** C_{in} propagates through each k-bit propagate/generate block

Carry-Lookahead Adder

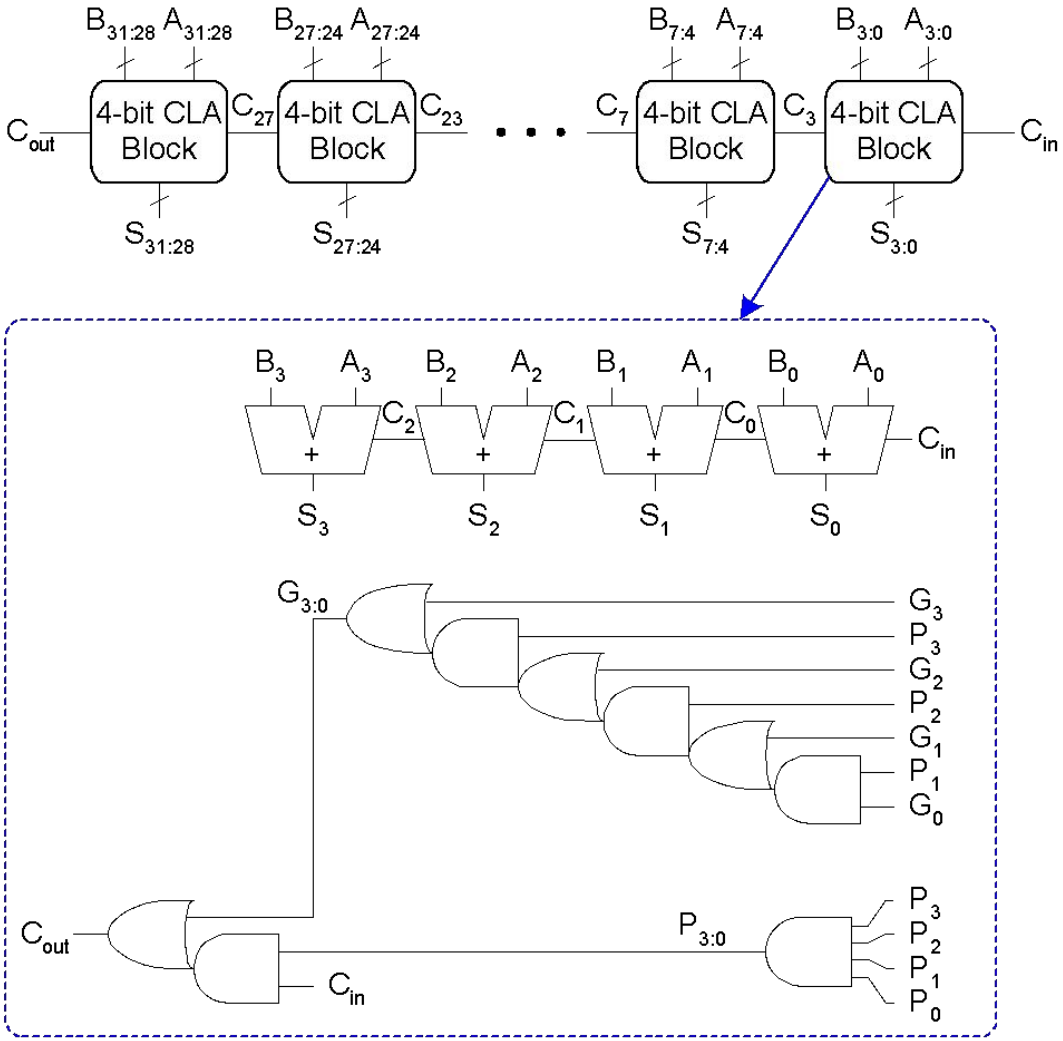
- **Example:** 4-bit blocks ($G_{3:0}$ and $P_{3:0}$) :

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$
$$P_{3:0} = P_3 P_2 P_1 P_0$$

- **Generally,**

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} G_j))$$
$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$
$$C_i = G_{i:j} + P_{i:j} C_{j-1}$$

32-bit CLA with 4-bit Blocks



Carry-Lookahead Adder Delay

For N-bit CLA with k-bit blocks:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

- t_{pg} : delay to generate all P_i, G_i
- t_{pg_block} : delay to generate all $P_{i:j}$, $G_{i:j}$
- t_{AND_OR} : delay from C_{in} to C_{out} of final AND/OR gate in k-bit CLA block

An N-bit carry-lookahead adder is generally much faster than a ripple-carry adder for $N > 16$

Prefix Adder

- Computes carry in (C_{i-1}) for each column, then computes sum:

$$S_i = (A_i \oplus B_i) \oplus C_i$$

- Computes G and P for 1-, 2-, 4-, 8-bit blocks, etc. until all G_i (carry in) known
- $\log_2 N$ stages

Prefix Adder

- Carry in either *generated* in a column or *propagated* from a previous column.
- Column -1 holds C_{in}, so

$$G_{-1} = C_{in}, P_{-1} = 0$$

- Carry in to column i = carry out of column i-1:

$$C_{i-1} = G_{i-1:-1}$$

$G_{i-1:-1}$: generate signal spanning columns i-1 to -1

- Sum equation:

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

- **Goal:** Quickly compute $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, … (called ***prefixes***)

Prefix Adder

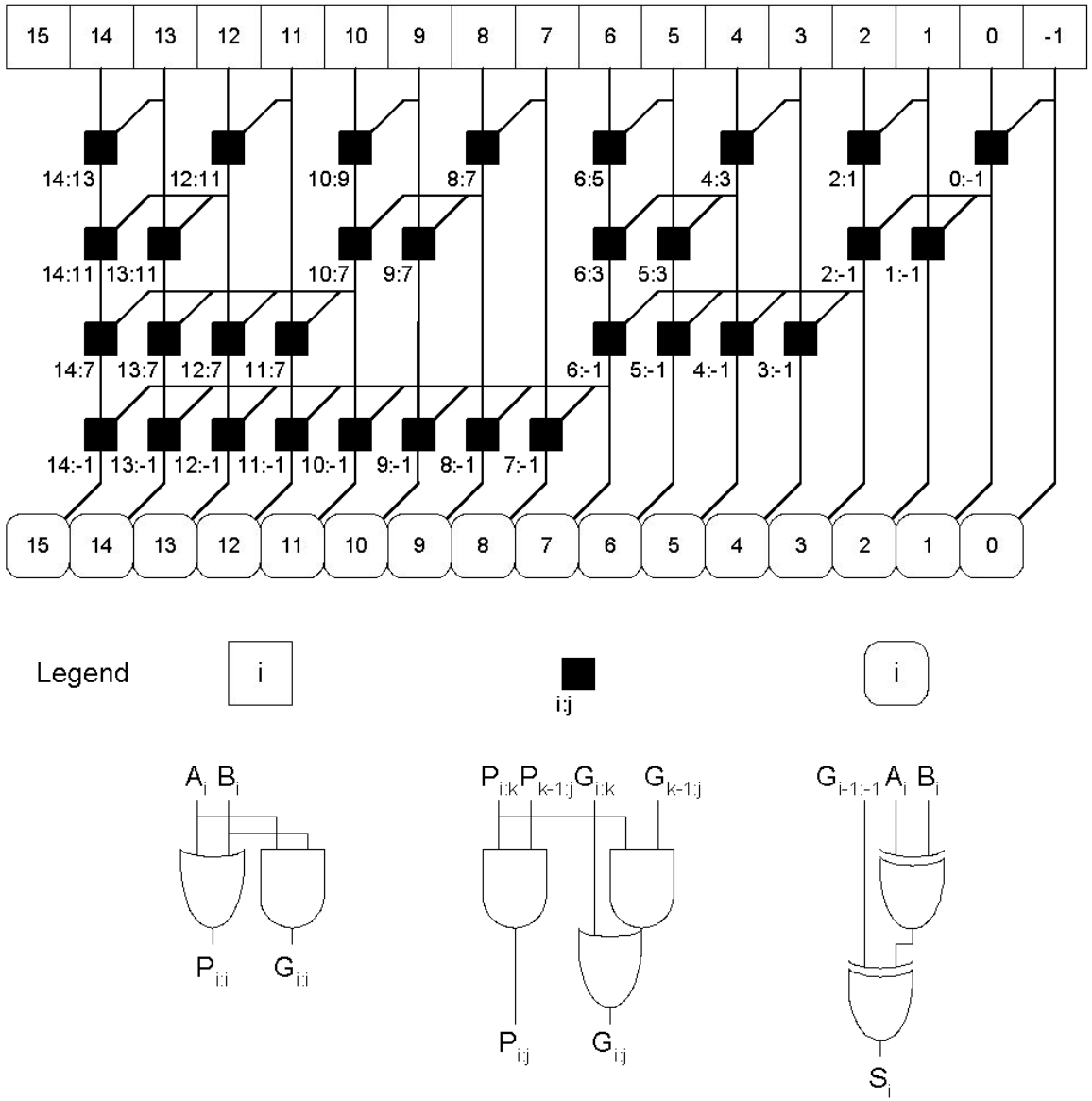
- Generate and propagate signals for a block spanning bits $i{:}j$:

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j}$$

$$P_{i:j} = P_{i:k} P_{k-1:j}$$

- In words:
  - **Generate:** block $i{:}j$ will generate a carry if:
    - upper part ($i{:}k$) generates a carry or
    - upper part propagates a carry generated in lower part (k-1$:j$)
  - **Propagate:** block $i{:}j$ will propagate a carry if *both* the upper and lower parts propagate the carry

Prefix Adder Schematic



Prefix Adder Delay

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR}$$

- t_{pg}: delay to produce P_i G_i (AND or OR gate)
- t_{pg_prefix}: delay of black prefix cell (AND-OR gate)

Adder Delay Comparisons

Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps

Adder Delay Comparisons

Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps

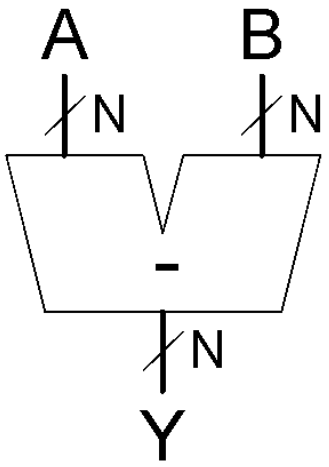
$$t_{\text{ripple}} = Nt_{FA} = 32(300\text{ ps}) = \mathbf{9.6\text{ ns}}$$

$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA}$$
$$= [100 + 600 + (7)200 + 4(300)]\text{ ps} = \mathbf{3.3\text{ ns}}$$

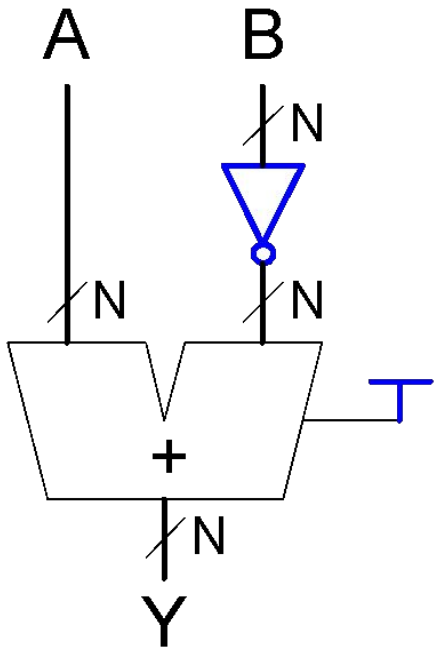
$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}}$$
$$= [100 + \log_2 32(200) + 100]\text{ ps} = \mathbf{1.2\text{ ns}}$$

Subtracter

Symbol



Implementation



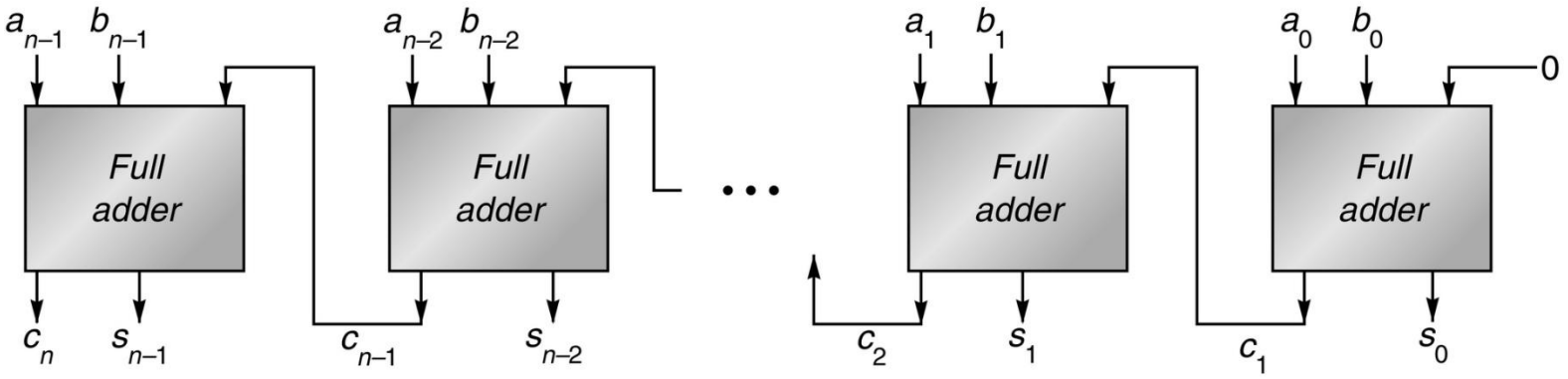


Figure J.1 Ripple-carry adder, consisting of *n* full adders. The carry-out of one full adder is connected to the carry-in of the adder for the next most-significant bit. The carries ripple from the least-significant bit (on the right) to the most-significant bit (on the left).

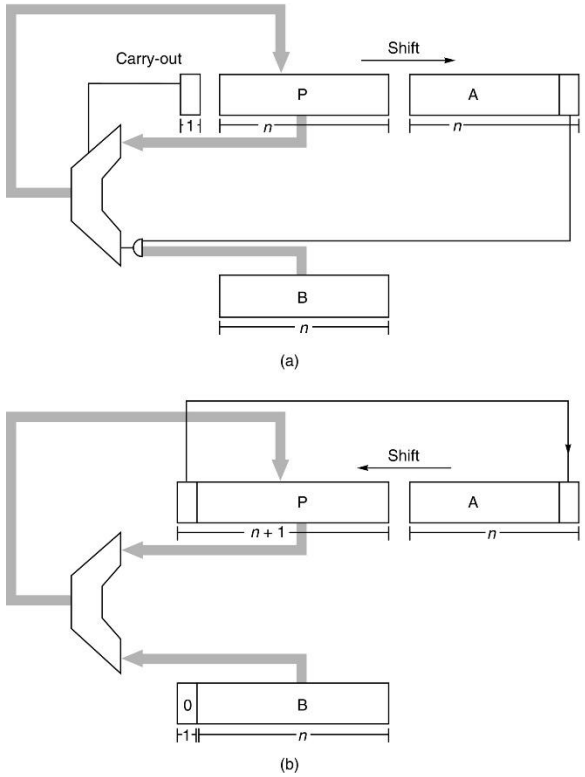


Figure J.2 Block diagram of (a) multiplier and (b) divider for *n*-bit unsigned integers. Each multiplication step consists of adding the contents of P to either B or 0 (depending on the low-order bit of A), replacing P with the sum, and then shifting both P and A one bit right. Each division step involves first shifting P and A one bit left, subtracting B from P, and, if the difference is nonnegative, putting it into P. If the difference is nonnegative, the low-order bit of A is set to 1.

P	A	
00000	1110	Divide $14 = 1110_2$ by $3 = 11_2$. B always contains 0011_2.
00001	110	step 1(i): shift.
−00011		step 1(ii): subtract.
−00010	110**0**	step 1(iii): result is negative, set quotient bit to 0.
00001	110**0**	step 1(iv): restore.
00011	10**0**	step 2(i): shift.
−00011		step 2(ii): subtract.
00000	10**01**	step 2(iii): result is nonnegative, set quotient bit to 1.
00001	0**01**	step 3(i): shift.
−00011		step 3(ii): subtract.
−00010	0**010**	step 3(iii): result is negative, set quotient bit to 0.
00001	0**010**	step 3(iv): restore.
00010	**010**	step 4(i): shift.
−00011		step 4(ii): subtract.
−00001	**0100**	step 4(iii): result is negative, set quotient bit to 0.
00010	**0100**	step 4(iv): restore. The quotient is 0100_2 and the remainder is 00010_2.

(a)

00000	1110	Divide $14 = 1110_2$ by $3 = 11_2$. B always contains 0011_2.
00001	110	step 1(i-b): shift.
+11101		step 1(ii-b): subtract b (add two's complement).
11110	110**0**	step 1(iii): P is negative, so set quotient bit to 0.
11101	10**0**	step 2(i-a): shift.
+00011		step 2(ii-a): add b.
00000	10**01**	step 2(iii): P is nonnegative, so set quotient bit to 1.
00001	0**01**	step 3(i-b): shift.
+11101		step 3(ii-b): subtract b.
11110	0**010**	step 3(iii): P is negative, so set quotient bit to 0.
11100	**010**	step 4(i-a): shift.
+00011		step 4(ii-a): add b.
11111	**0100**	step 4(iii): P is negative, so set quotient bit to 0.
+00011		Remainder is negative, so do final restore step.
00010		The quotient is 0100_2 and the remainder is 00010_2.

(b)

Figure J.3 Numerical example of (a) restoring division and (b) nonrestoring division.

P	A	
0000	1010	Put $-6 = 1010_2$ into A, $-5 = 1011_2$ into B.
0000	1010	step 1(i): $a_0 = a_{-1} = 0$, so from rule I add 0.
0000	0101	step 1(ii): shift.
+0101		step 2(i): $a_1 = 1$, $a_0 = 0$.Rule III says subtract b (or add $-b = -1011_2 = 0101_2$).
0101	0101	
0010	1010	step 2(ii): shift.
+ 1011		step 3(i): $a_2 = 0$, $a_1 = 1$. Rule II says add b (1011).
1101	1010	
1110	1101	step 3(ii): shift. (Arithmetic shift—load 1 into leftmost bit.)
+ 0101		step 4(i): $a_3 = 1$, $a_2 = 0$. Rule III says subtract b.
0011	1101	
0001	1110	step 4(ii): shift. Final result is $00011110_2 = 30$.

Figure J.4 Numerical example of Booth recoding. Multiplication of *a* = − 6 by *b* = − 5 to get 30.

Machine	Trap on signed overflow?	Trap on unsigned overflow?	Set bit on signed overflow?	Set bit on unsigned overflow?
VAX	If enable is on	No	Yes. `Add` sets V bit.	Yes. `Add` sets C bit.
IBM 370	If enable is on	No	Yes. `Add` sets cond code.	Yes. Logical `add` sets cond code.
Intel 8086	No	No	Yes. `Add` sets V bit.	Yes. `Add` sets C bit.
MIPS R3000	Two `add` instructions; one always traps, the other never does.	No	No. Software must deduce it from sign of operands and result.	
SPARC	No	No	`Addcc` sets V bit. `Add` does not.	`Addcc` sets C bit. `Add` does not.

Figure J.5 Summary of how various machines handle integer overflow. Both the 8086 and SPARC have an instruction that traps if the V bit is set, so the cost of trapping on overflow is one extra instruction.

Language	Division	Remainder
FORTRAN	$-5/3 = -1$	MOD$(-5, 3) = -2$
Pascal	-5 DIV $3 = -1$	-5 MOD $3 = 1$
Ada	$-5/3 = -1$	-5 MOD $3 = 1$ -5 REM $3 = -2$
C	$-5/3$ undefined	-5% 3 undefined
Modula-3	-5 DIV $3 = -2$	-5 MOD $3 = 1$

Figure J.6 Examples of integer division and integer remainder in various programming languages.

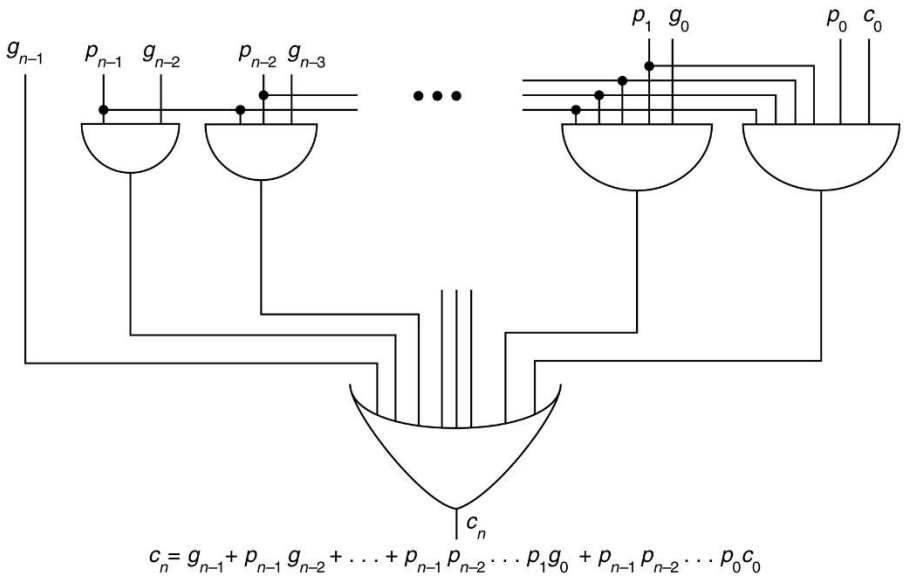


Figure J.14 Pure carry-lookahead circuit for computing the carry-out c_n of an *n*-bit adder.

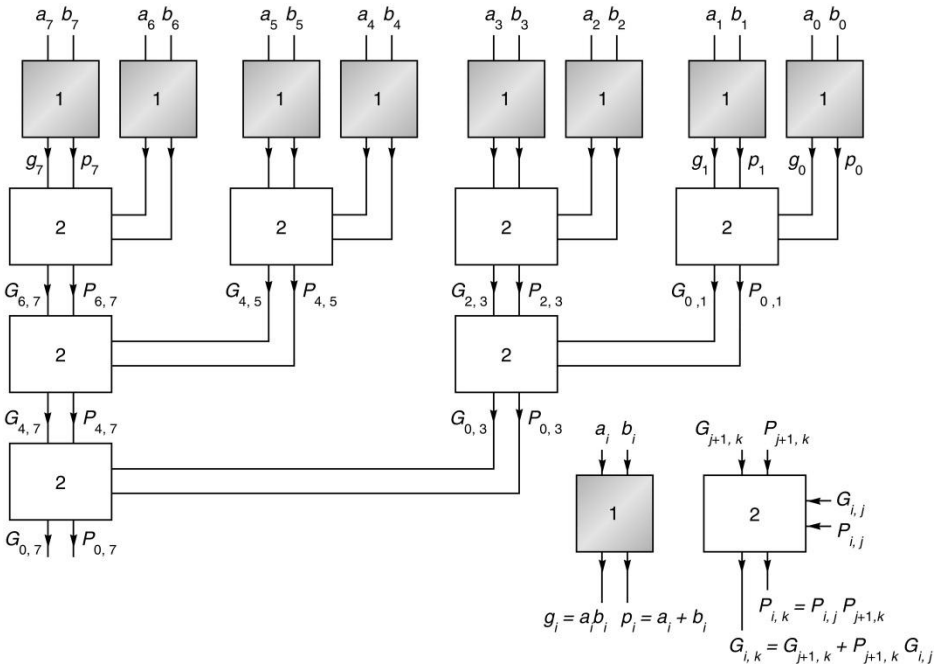


Figure J.15 First part of carry-lookahead tree. As signals flow from the top to the bottom, various values of *P* and *G* are computed.

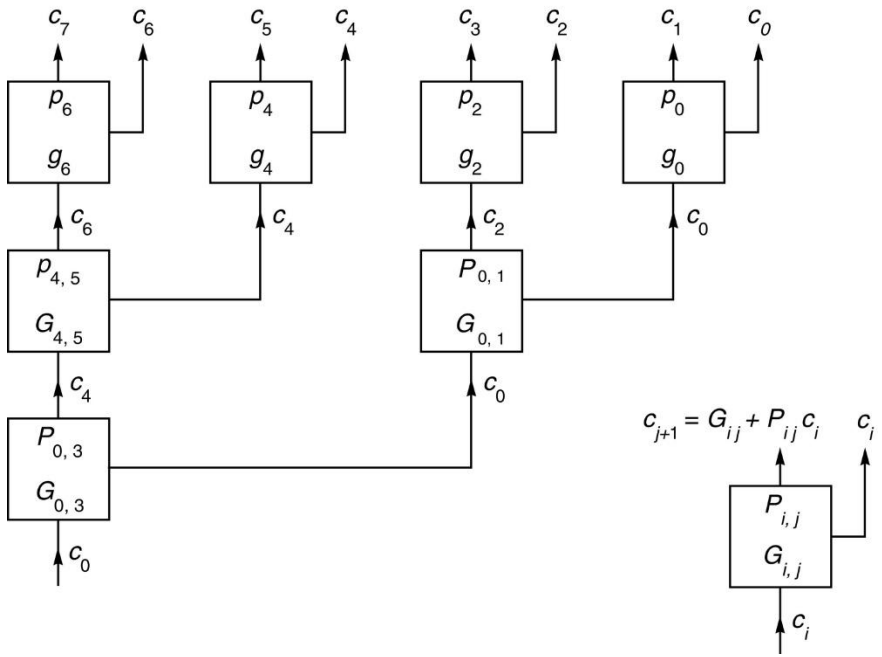


Figure J.16 Second part of carry-lookahead tree. Signals flow from the bottom to the top, combining with *P* and *G* to form the carries.

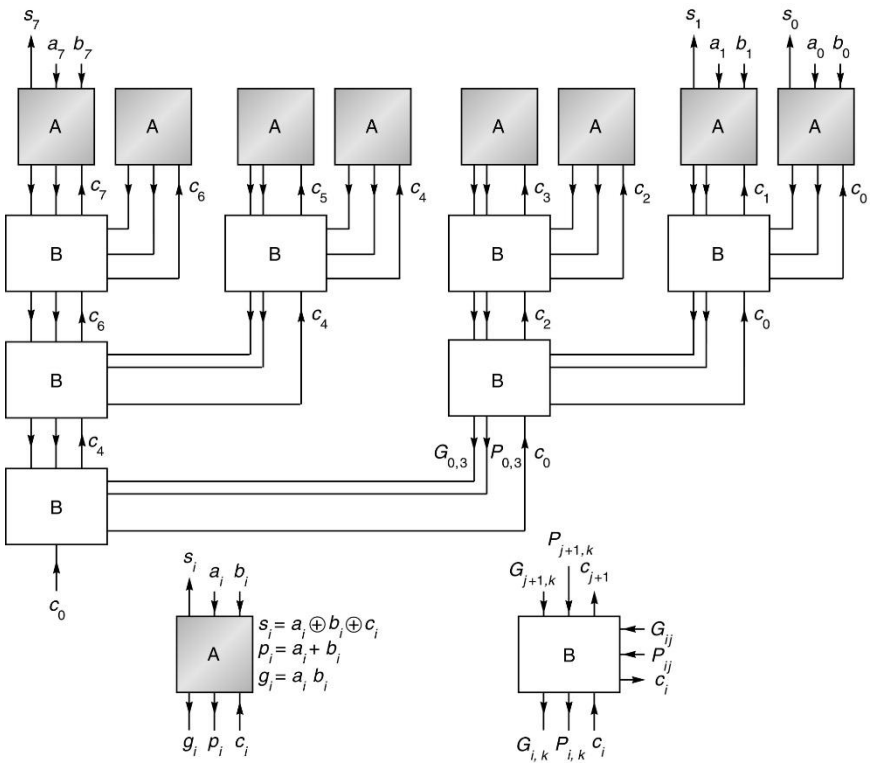


Figure J.17 Complete carry-lookahead tree adder. This is the combination of Figures J.15 and J.16. The numbers to be added enter at the top, flow to the bottom to combine with c_0, and then flow back up to compute the sum bits.

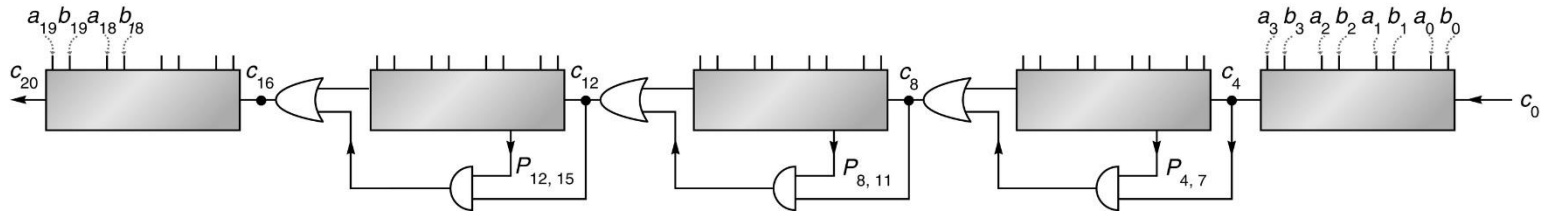


Figure J.18 Carry-skip adder. This is a 20-bit carry-skip adder (n = 20) with each block 4 bits wide (k = 4).

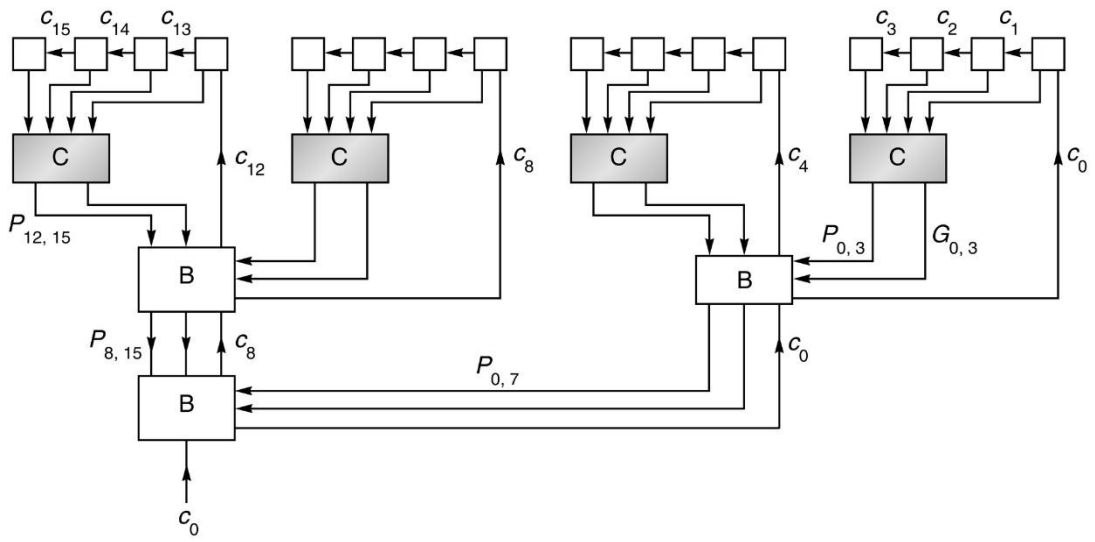


Figure J.19 Combination of CLA and ripple-carry adder. In the top row, carries ripple within each group of four boxes.

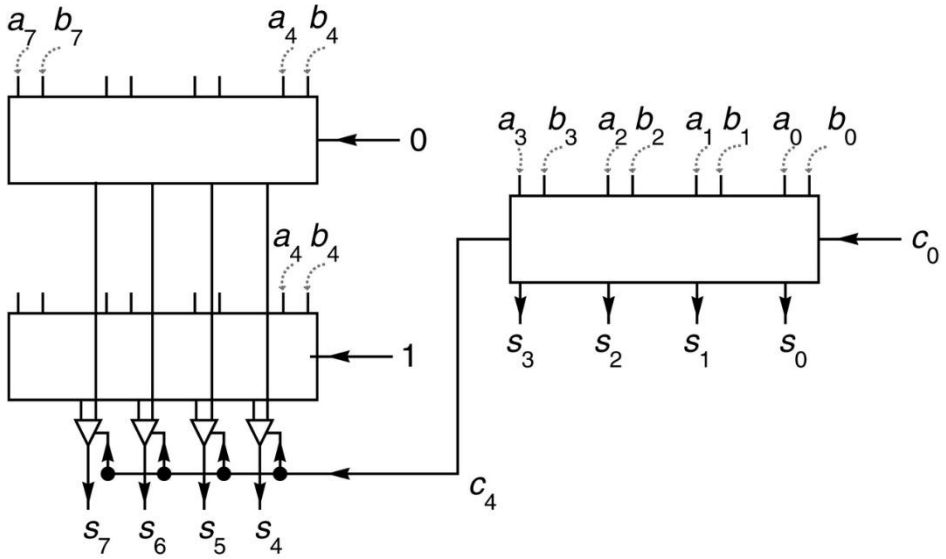


Figure J.20 Simple carry-select adder. At the same time that the sum of the low-order 4 bits is being computed, the high-order bits are being computed twice in parallel: once assuming that $c_4 = 0$ and once assuming $c_4 = 1$.

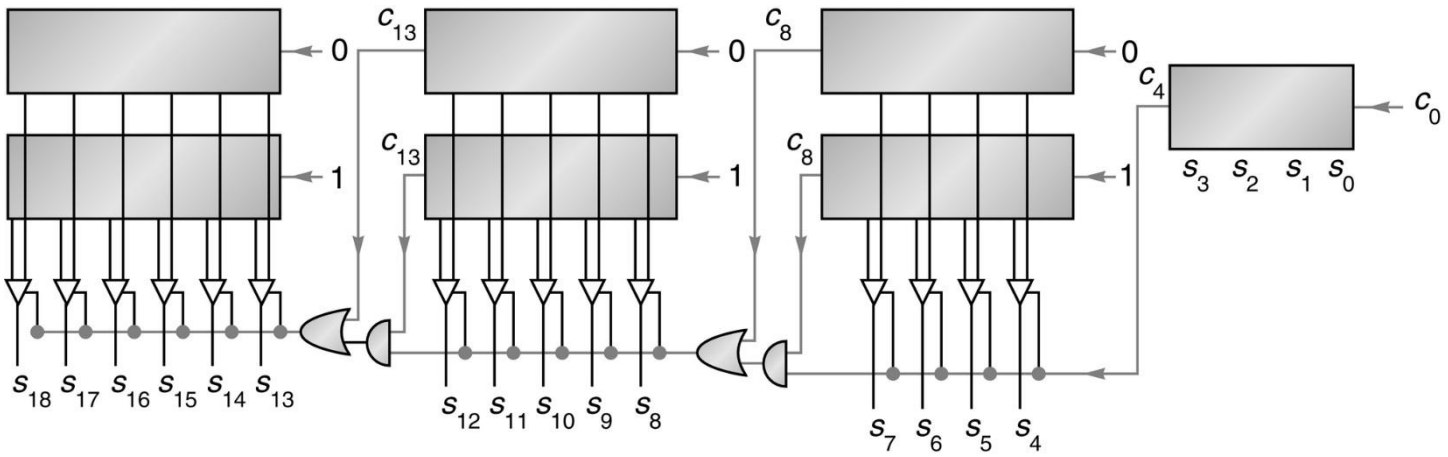


Figure J.21 Carry-select adder. As soon as the carry-out of the rightmost block is known, it is used to select the other sum bits.

Adder	Time	Space
Ripple	$0(n)$	$0(n)$
CLA	$0(\log n)$	$0(n \log n)$
Carry-skip	$0(\sqrt{n})$	$0(n)$
Carry-select	$0(\sqrt{n})$	$0(n)$

Figure J.22 Asymptotic time and space requirements for four different types of adders.

P	A	
00000	1000	Divide 8 = 1000 by 3 = 0011. B contains 0011.
00010	0000	Step 1: B had two leading 0 s, so shift left by 2. B now contains 1100.
		Step 2.1: Top three bits are equal. This is case (a), so
00100	0000	set $q_0 = 0$ and shift.
		Step 2.2: Top three bits not equal and P ≥ 0 is case (c), so
01000	0001	set $q_1 = 1$ and shift.
+ 10100		Subtract B.
11100	0001	Step 2.3: Top bits equal is case (a), so
11000	0010	set $q_2 = 0$ and shift.
		Step 2.4: Top three bits unequal is case (b), so
10000	0101	set $q_3 = -1$ and shift.
+ 01100		Add B.
11100		Step 3. remainder is negative so restore it and subtract 1 from q.
+ 01100		
01000		Must undo the shift in step 1, so right-shift by 2 to get true remainder. Remainder = 10, quotient = $0101 - 1 = 0010$.

Figure J.23 SRT division of $1000_2/0011_2$. The quotient bits are shown in bold, using the notation 1 for − 1.

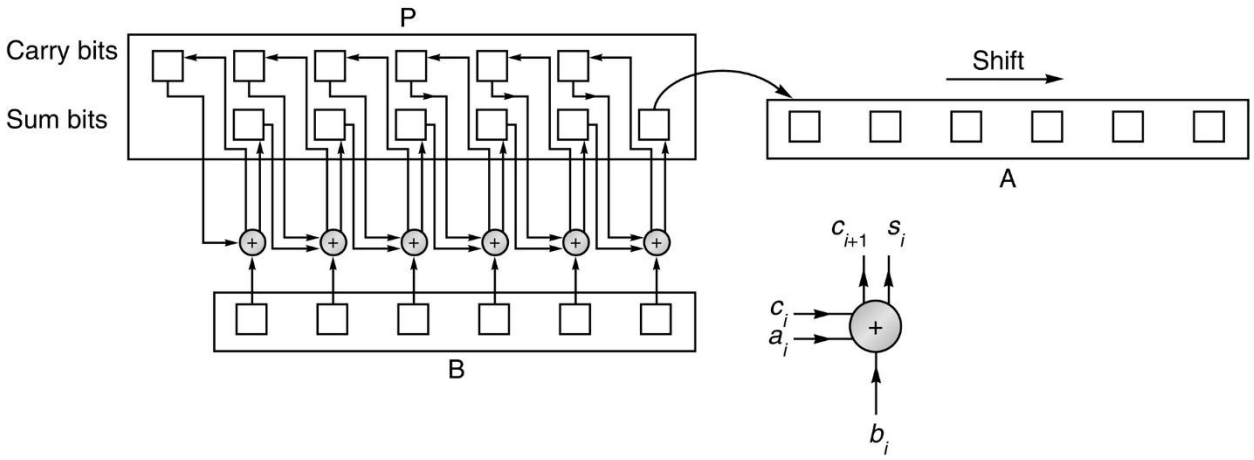


Figure J.24 Carry-save multiplier. Each circle represents a (3,2) adder working independently. At each step, the only bit of P that needs to be shifted is the low-order sum bit.

Low-order bits of A		Last bit shifted out	
$2i+1$	$2i$	$2i-1$	Multiple
0	0	0	0
0	0	1	$+b$
0	1	0	$+b$
0	1	1	$+2b$
1	0	0	$-2b$
1	0	1	$-b$
1	1	0	$-b$
1	1	1	0

Figure J.25 Multiples of *b* to use for radix-4 Booth recoding. For example, if the two low-order bits of the A register are both 1, and the last bit to be shifted out of the A register is 0, then the correct multiple is − *b*, obtained from the second-to-last row of the table.

P	A	L	
00000	1001		Multiply $-7 = 1001$ times $-5 = 1011$. B contains 1011.
+ 11011			Low-order bits of A are 0, 1; L = 0, so add B.
11011	1001		
11110	1110	0	Shift right by two bits, shifting in 1 s on the left.
+ 01010			Low-order bits of A are 1, 0; L = 0, so add −2b.
01000	1110	0	
00010	0011	1	Shift right by two bits.
			Product is $35 = 0100011$.

Figure J.26 Multiplication of − 7 times − 5 using radix-4 Booth recoding. The column labeled L contains the last bit shifted out the right end of A.

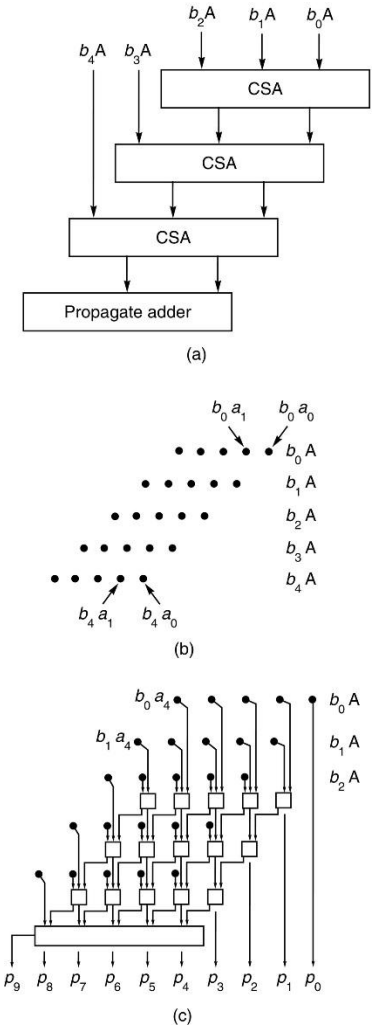


Figure J.27 An array multiplier. The 5-bit number in A is multiplied by $b_4b_3b_2b_1b_0$. Part (a) shows the block diagram, (b) shows the inputs to the array, and (c) expands the array to show all the adders.

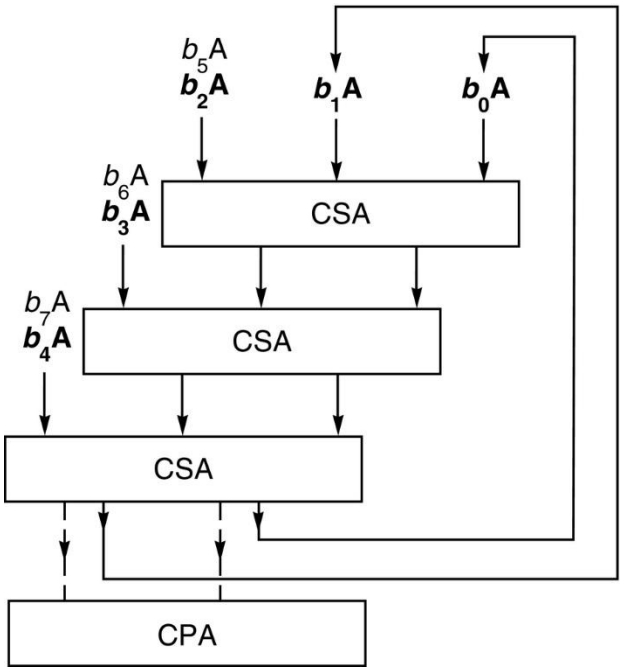


Figure J.28 Multipass array multiplier. Multiplies two 8-bit numbers with about half the hardware that would be used in a one-pass design like that of Figure J.27. At the end of the second pass, the bits flow into the CPA. The inputs used in the first pass are marked in bold.

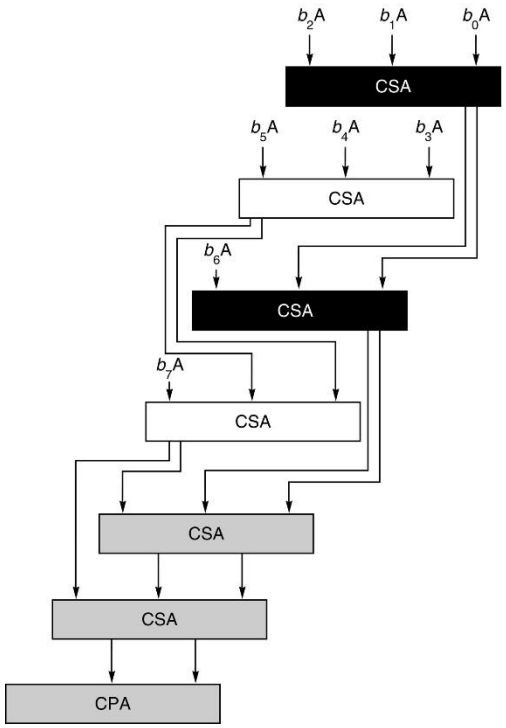


Figure J.29 Even/odd array. The first two adders work in parallel. Their results are fed into the third and fourth adders, which also work in parallel, and so on.

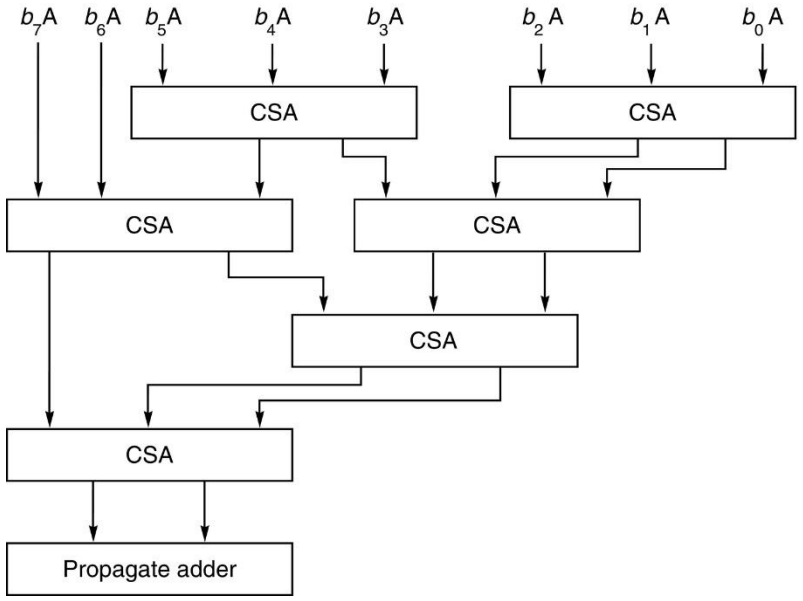


Figure J.30 Wallace tree multiplier. An example of a multiply tree that computes a product in 0(log *n*) steps.

$\begin{array}{r} 1 \\ +\ 1 \\ \hline 1\ 0 \end{array}$	$\begin{array}{r} 1 \\ +\ \bar{1} \\ \hline 0\ 0 \end{array}$	$\begin{array}{r} \bar{1} \\ +\ \bar{1} \\ \hline \bar{1}\ 0 \end{array}$	$\begin{array}{r} 0 \\ +\ 0 \\ \hline 0\ 0 \end{array}$	$\begin{array}{r} \bar{1}\ x \\ +\ 0\ y \\ \hline 1\ \bar{1} \\ 0\ 1 \end{array}$	if $x \geq 0$ and $y \geq 0$ otherwise	$\begin{array}{r} \bar{1}\ x \\ +\ 0\ y \\ \hline 1\ \bar{1} \\ \bar{1}\ 1 \end{array}$	if $x \geq 0$ and $y \geq 0$ otherwise

Figure J.31 Signed-digit addition table. The leftmost sum shows that when computing 1 + 1, the sum bit is 0 and the carry bit is 1.

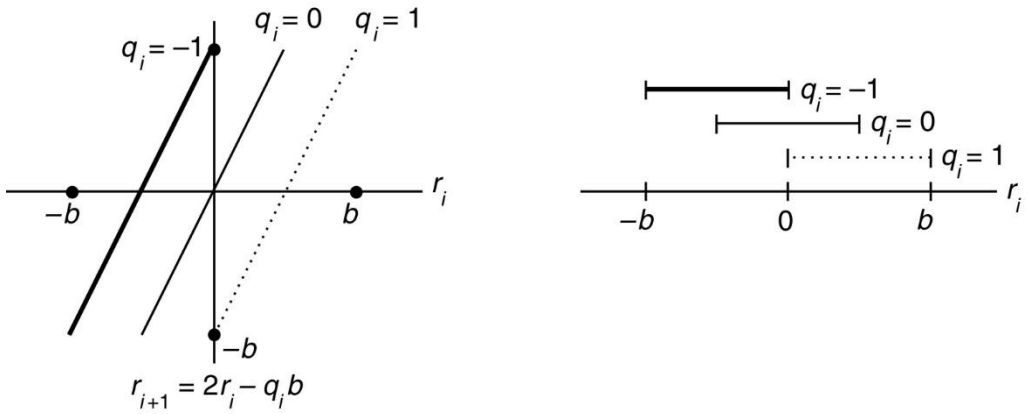


Figure J.32 Quotient selection for radix-2 division. The *x*-axis represents the *i*th remainder, which is the quantity in the (P,A) register pair. The *y*-axis shows the value of the remainder after one additional divide step. Each bar on the right-hand graph gives the range of r_i values for which it is permissible to select the associated value of q_i.

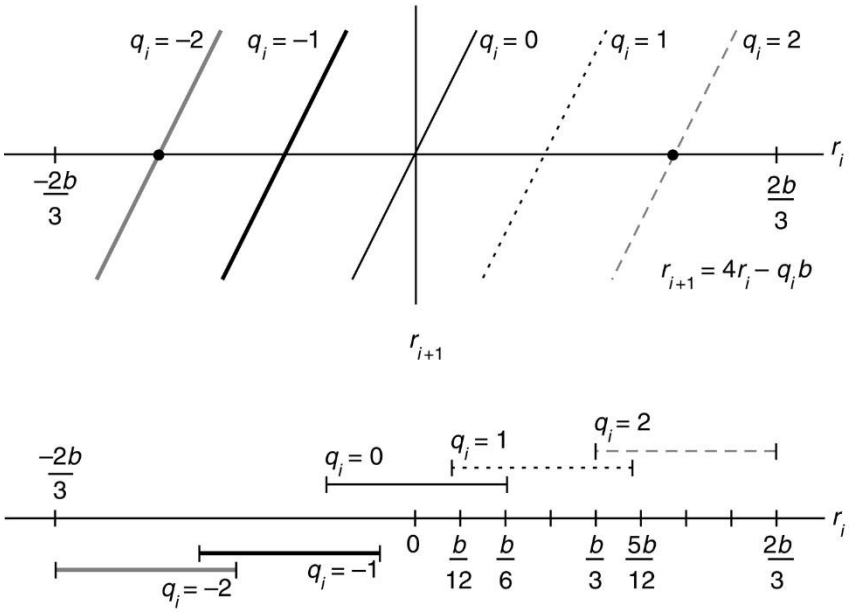


Figure J.33 Quotient selection for radix-4 division with quotient digits − 2, − 1, 0, 1, 2.

b	Range of P		q	b	Range of P		q
8	−12	−7	−2	12	−18	−10	−2
8	−6	−3	−1	12	−10	−4	−1
8	−2	1	0	12	−4	3	0
8	2	5	1	12	3	9	1
8	6	11	2	12	9	17	2
9	−14	−8	−2	13	−19	−11	−2
9	−7	−3	−1	13	−10	−4	−1
9	−3	2	0	13	−4	3	0
9	2	6	1	13	3	9	1
9	7	13	2	13	10	18	2
10	−15	−9	−2	14	−20	−11	−2
10	−8	−3	−1	14	−11	−4	−1
10	−3	2	0	14	−4	3	0
10	2	7	1	14	3	10	1
10	8	14	2	14	10	19	2
11	−16	−9	−2	15	−22	−12	−2
11	−9	−3	−1	15	−12	−4	−1
11	−3	2	0	15	−5	4	0
11	2	8	1	15	3	11	1
11	8	15	2	15	11	21	2

Figure J.34 Quotient digits for radix-4 SRT division with a propagate adder. The top row says that if the high-order 4 bits of *b* are 1000_2 = 8, and if the top 6 bits of P are between 110100_2 = − 12 and 111001_2 = − 7, then − 2 is a valid quotient digit.

P	A		
000000000	10010101	Divide 149 by 5. B contains 00000101.	
000010010	10100000	Step 1:	B had 5 leading 0s, so shift left by 5. B now contains 10100000, so use $b=10$ section of table.
001001010	1000000	Step 2.1:	Top 6 bits of P are 2, so shift left by 2. From table, can pick q to be 0 or 1. Choose $q_0=0$.
100101010	000002	Step 2.2:	Top 6 bits of P are 9, so shift left 2. $q_1=2$.
+ 011000000			Subtract $2b$.
111101010	000002	Step 2.3:	Top bits $=-3$, so shift left 2. Can pick 0 or -1 for q, pick $q_2=0$.
110101000	00020		
		Step 2.4:	Top bits $=-11$, so shift left 2. $q_3=-2$.
010100000	0202		
+ 101000000			Add $2b$.
111100000		Step 3:	Remainder is negative, so restore by adding b and subtract 1 from q.
+ 010100000			
010000000		Answer:	$q=020\overline{2}-1=29$
			To get remainder, undo shift in step 1 so remainder $=010000000>>5=4$.

Figure J.35 Example of radix-4 SRT division. Division of 149 by 5.