Digital Logic Design

- Basics
- Combinational Circuits
- Sequential Circuits

Pu-Jen Cheng

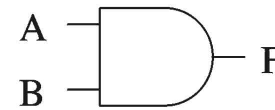
Adapted from the slides prepared by S. Dandamudi for the book, Fundamentals of Computer Organization and Design.

Introduction to Digital Logic Basics

- Hardware consists of a few simple building blocks
  - These are called *logic gates*
    - AND, OR, NOT, …
    - NAND, NOR, XOR, …
- Logic gates are built using transistors
    - NOT gate can be implemented by a single transistor
    - AND gate requires 3 transistors
- Transistors are the fundamental devices
    - Pentium consists of 3 million transistors
    - Compaq Alpha consists of 9 million transistors
    - Now we can build chips with more than 100 million transistors

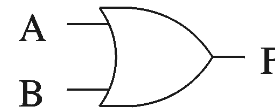
Basic Concepts

- Simple gates
  - AND
  - OR
  - NOT
- Functionality can be expressed by a truth table
  - A truth table lists output for each possible input combination
- Precedence
  - NOT > AND > OR
  - $F = A\,\overline{B} + \overline{A}\,B$
    $= (A\,(\overline{B})) + ((\overline{A})\,B)$



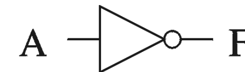
AND gate

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1



OR gate

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1



NOT gate

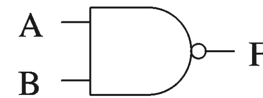
A	F
0	1
1	0

Logic symbol

Truth table

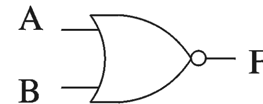
Basic Concepts (cont.)

- Additional useful gates
  - NAND
  - NOR
  - XOR
- NAND = AND + NOT
- NOR = OR + NOT
- XOR implements exclusive-OR function
- NAND and NOR gates require only 2 transistors
  - AND and OR need 3 transistors!



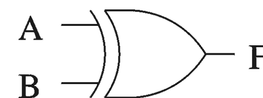
NAND gate

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0



NOR gate

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0



XOR gate

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Logic symbol

Truth table

Basic Concepts (cont.)

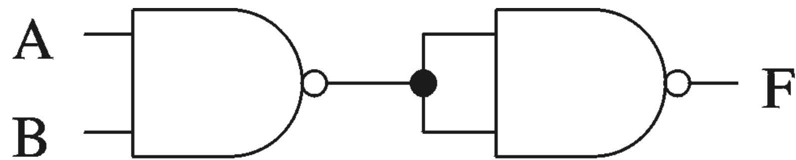
- Number of functions
  - With N logical variables, we can define
    2^{2^N} functions
  - Some of them are useful
    - AND, NAND, NOR, XOR, …
  - Some are not useful:
    - Output is always 1
    - Output is always 0
  - “Number of functions” definition is useful in proving completeness property

Basic Concepts (cont.)

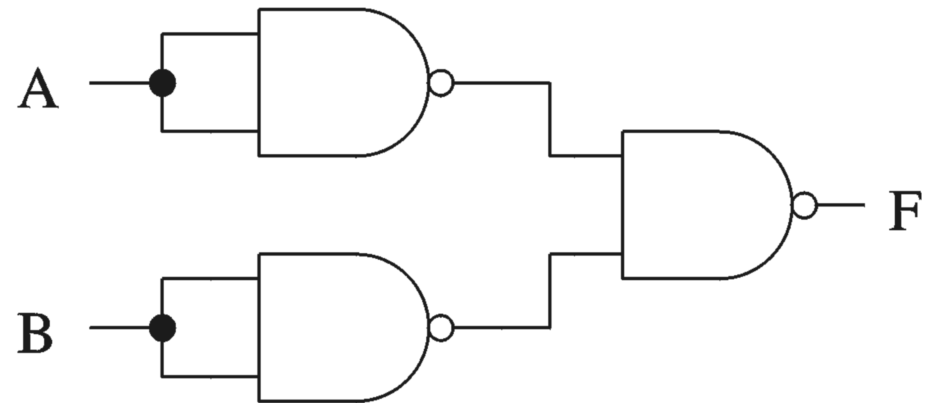
- Complete sets
  - A set of gates is complete
    - If we can implement any logical function using only the type of gates in the set
      - You can uses as many gates as you want
  - Some example complete sets
    - {AND, OR, NOT} ← Not a minimal complete set
    - {AND, NOT}
    - {OR, NOT}
    - {NAND}
    - {NOR}
  - Minimal complete set
      - A complete set with no redundant elements.

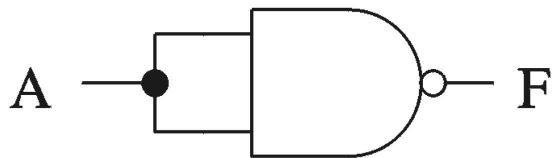
Basic Concepts (cont.)

- Proving NAND gate is universal
- NAND gate is called *universal gate*



AND gate



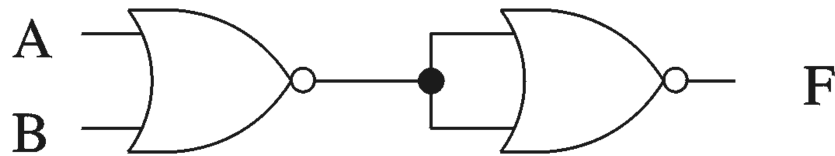


NOT gate

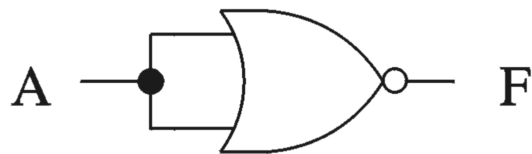
OR gate

Basic Concepts (cont.)

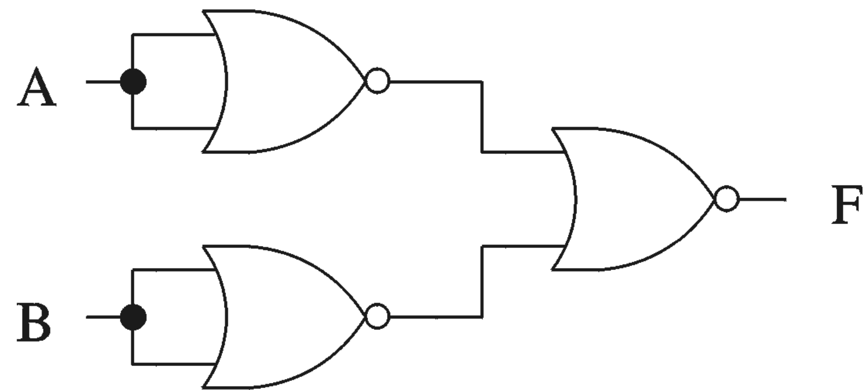
- Proving NOR gate is universal
- NOR gate is called *universal gate*



OR gate

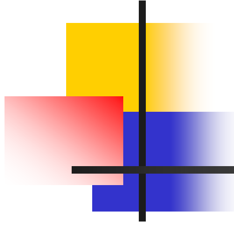


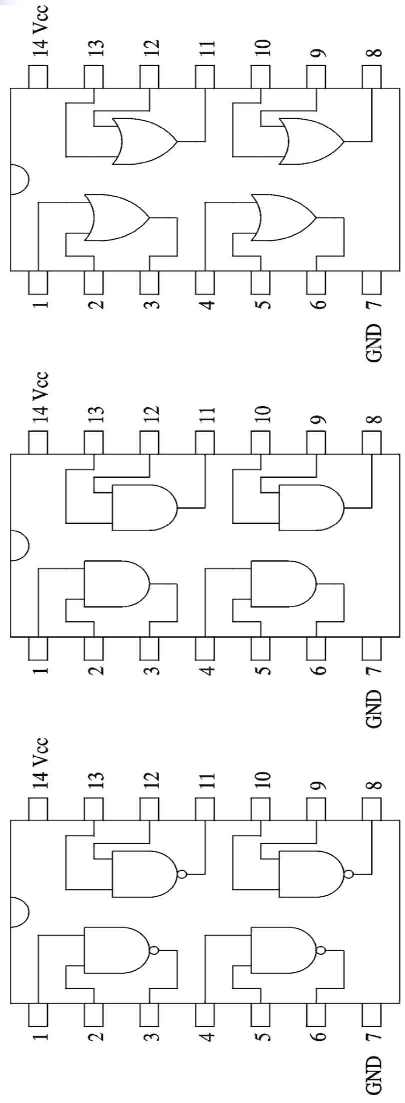
NOT gate

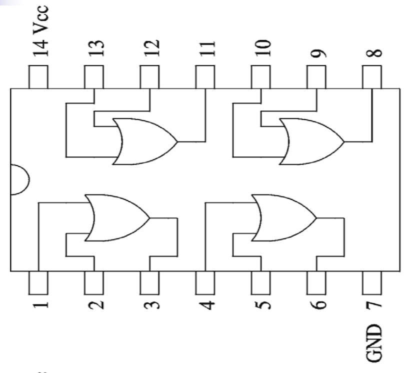


AND gate

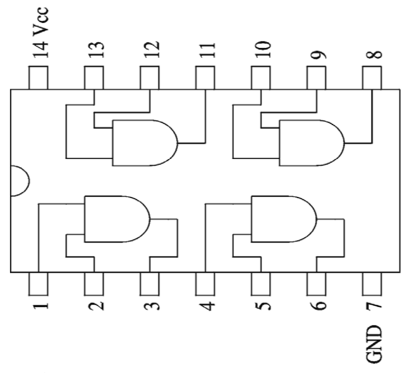
Logic Chips





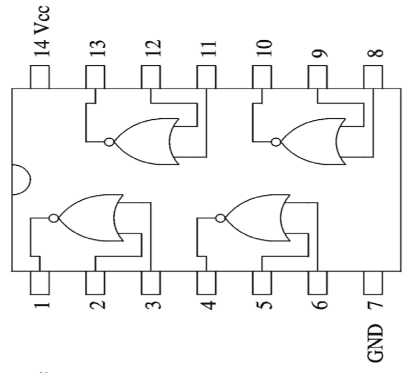


7432

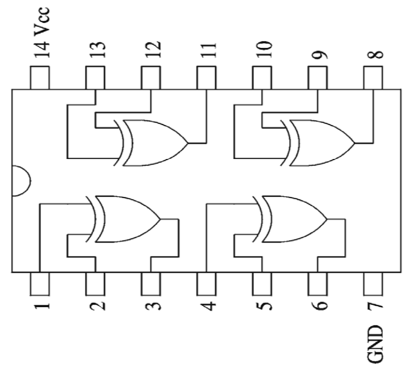


7408

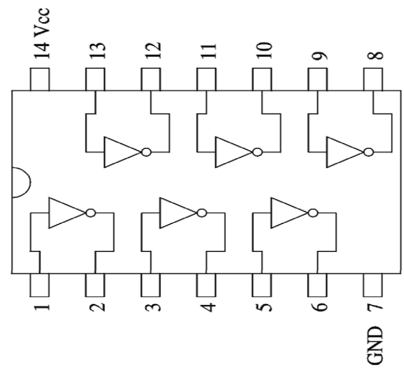
7400



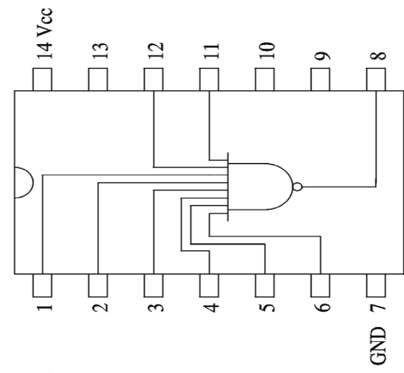
7402



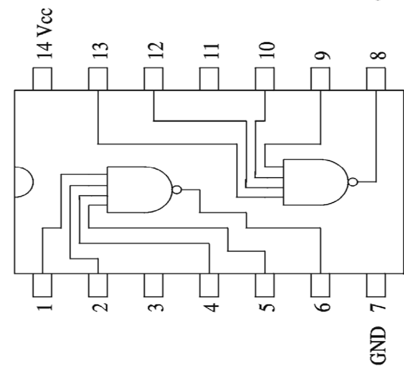
7486



7404



7430



7420

Logic Chips (cont.)

- Integration levels
  - SSI (small scale integration)
    - Introduced in late 1960s
    - 1-10 gates (previous examples)
  - MSI (medium scale integration)
    - Introduced in late 1960s
    - 10-100 gates
  - LSI (large scale integration)
    - Introduced in early 1970s
    - 100-10,000 gates
  - VLSI (very large scale integration)
    - Introduced in late 1970s
    - More than 10,000 gates

Logic Functions

- Logical functions can be expressed in several ways:
  - Truth table
  - Logical expressions
  - Graphical form
- Example:
  - Majority function
    - Output is 1 whenever majority of inputs is 1
    - We use 3-input majority function

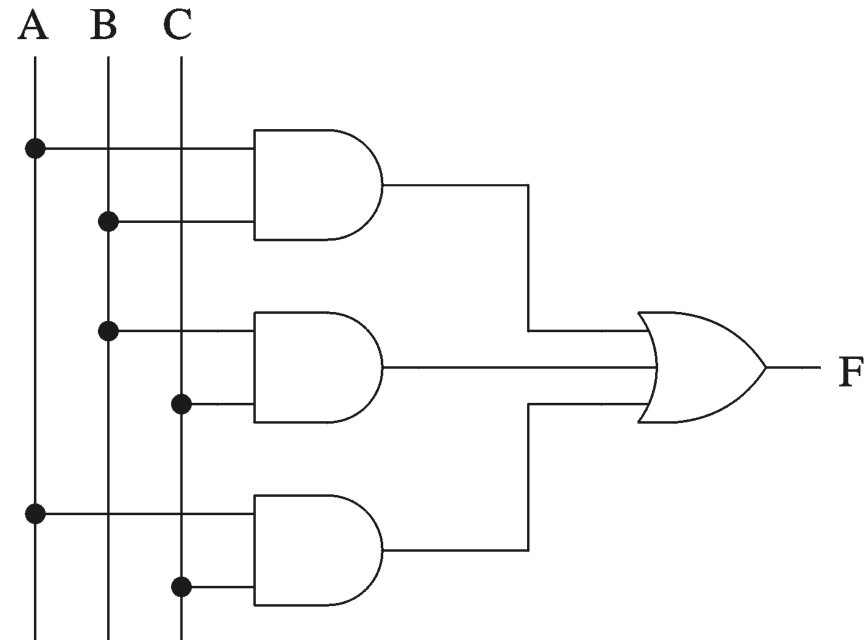
Logic Functions (cont.)

3-input majority function

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

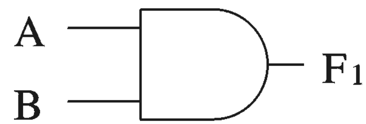
- Logical expression form

$$F = A\,B + B\,C + A\,C$$

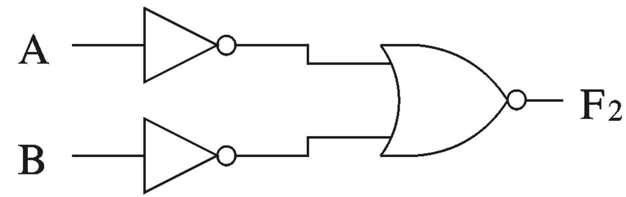


Logical Equivalence

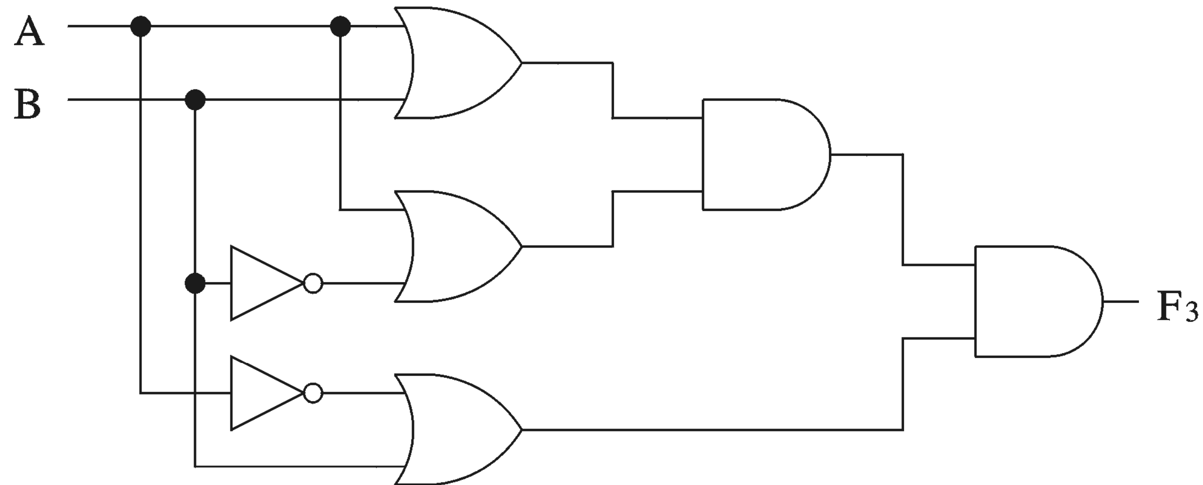
- All three circuits implement F = A B function



(a)



(b)

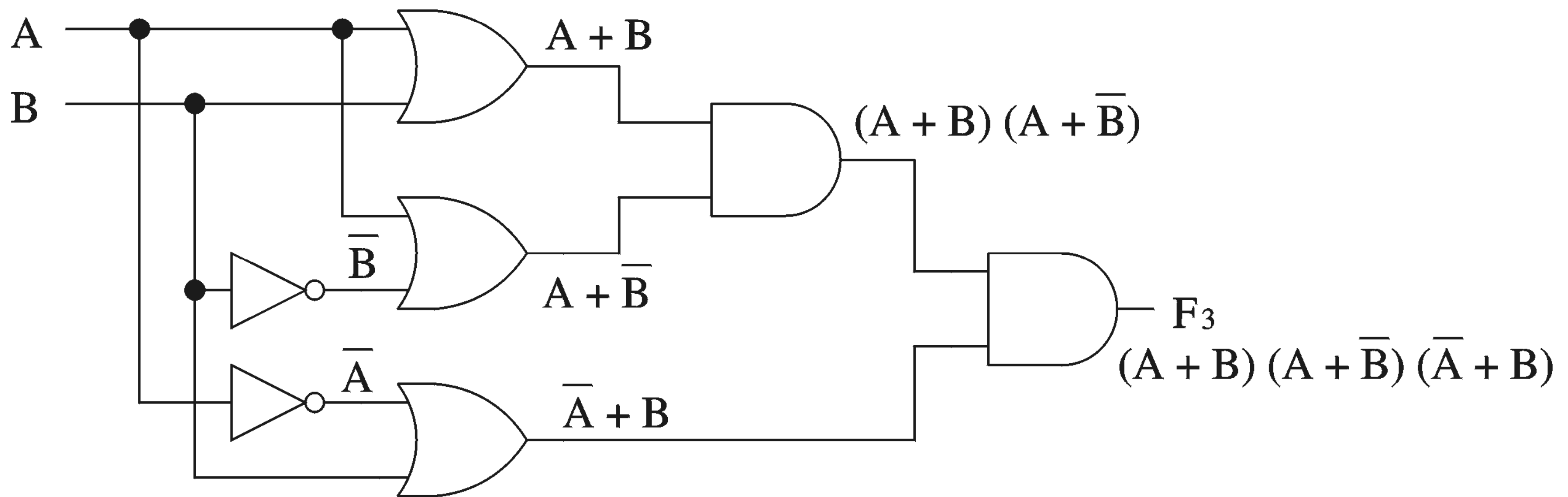


(c)

- Proving logical equivalence of two circuits
  - Derive the logical expression for the output of each circuit
  - Show that these two expressions are equivalent
    - Two ways:
      - You can use the truth table method
        - For every combination of inputs, if both expressions yield the same output, they are equivalent
        - Good for logical expressions with small number of variables
      - You can also use algebraic manipulation
        - Need Boolean identities

Logical Equivalence (cont.)

- Derivation of logical expression from a circuit
  - Trace from the input to output
    - Write down intermediate logical expressions along the path



Logical Equivalence (cont.)

- Proving logical equivalence: Truth table method

A	B	F1 = A B	F3 = $(A + B)(A + \overline{B})(\overline{A} + B)$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Boolean Algebra

Boolean identities

Name	AND version	OR version
Identity	$x \cdot 1 = x$	$x + 0 = x$
Complement	$x \cdot \overline{x} = 0$	$x + \overline{x} = 1$
Commutative	$x \cdot y = y \cdot x$	$x + y = y + x$
Distribution	$x \cdot (y+z) = xy+xz$	$x + (y \cdot z) = (x+y)(x+z)$
Idempotent	$x \cdot x = x$	$x + x = x$
Null	$x \cdot 0 = 0$	$x + 1 = 1$

Boolean Algebra (cont.)

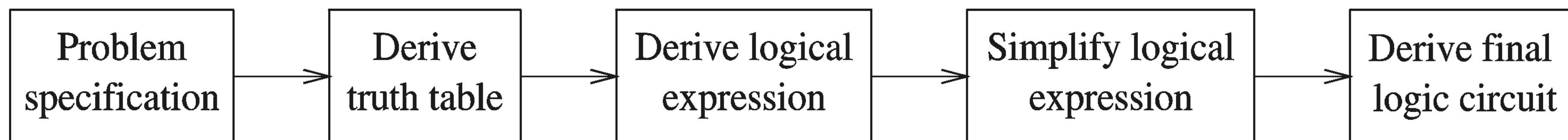
Name	AND version	OR version
Involution	$\overline{\overline{x}} = x$	---
Absorption	$x \cdot (x+y) = x$	$x + (x \cdot y) = x$
Associative	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	$x + (y + z) = (x + y) + z$
de Morgan	$\overline{x \cdot y} = \overline{x} + \overline{y}$	$\overline{x + y} = \overline{x} \cdot \overline{y}$

- Proving logical equivalence: Boolean algebra method
  - To prove that two logical functions F1 and F2 are equivalent
    - Start with one function and apply Boolean laws to derive the other function
    - Needs intuition as to which laws should be applied and when
      - Practice helps
    - Sometimes it may be convenient to reduce both functions to the same expression
  - Example: F1= A B and F3 are equivalent

$$\mathbf{A\,B = (A + B)\,(A + \overline{B})\,(\overline{A} + B)}$$

Logic Circuit Design Process

- A simple logic design process involves
  - Problem specification
  - Truth table derivation
  - Derivation of logical expression
  - Simplification of logical expression
  - Implementation



Deriving Logical Expressions

- Derivation of logical expressions from truth tables
  - sum-of-products (SOP) form
  - product-of-sums (POS) form
- SOP form
  - Write an AND term for each input combination that produces a 1 output
    - Write the variable if its value is 1; complement otherwise
  - OR the AND terms to get the final expression
- POS form
  - Dual of the SOP form

Deriving Logical Expressions (cont.)

- 3-input majority function

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

- SOP logical expression
- Four product terms
  - Because there are 4 rows with a 1 output

$$F = \overline{A}\,B\,C + A\,\overline{B}\,C + A\,B\,\overline{C} + A\,B\,C$$

Deriving Logical Expressions (cont.)

- 3-input majority function

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

- POS logical expression
- Four sum terms
  - Because there are 4 rows with a 0 output

$$F = (A + B + C)\,(A + B + \overline{C})\,(A + \overline{B} + C)\,(\overline{A} + B + C)$$

Logical Expression Simplification

- Two basic methods
  - Algebraic manipulation
    - Use Boolean laws to simplify the expression
      - Difficult to use
      - Don’t know if you have the simplified form
  - Karnaugh map (K-map) method
    - Graphical method
    - Easy to use
      - Can be used to simplify logical expressions with a few variables

Algebraic Manipulation

- Majority function example

Added extra

$$\bar{A}BC + A\bar{B}C + AB\bar{C} + ABC =$$

$$\bar{A}BC + A\bar{B}C + AB\bar{C} + ABC + ABC + ABC$$

- We can now simplify this expression as

$$BC + AC + AB$$

- A difficult method to use for complex expressions

Karnaugh Map Method

Note the order

A \ B	0	1
0		
1		

(a) Two-variable K-map

A \ BC	00	01	11	10
0				
1				

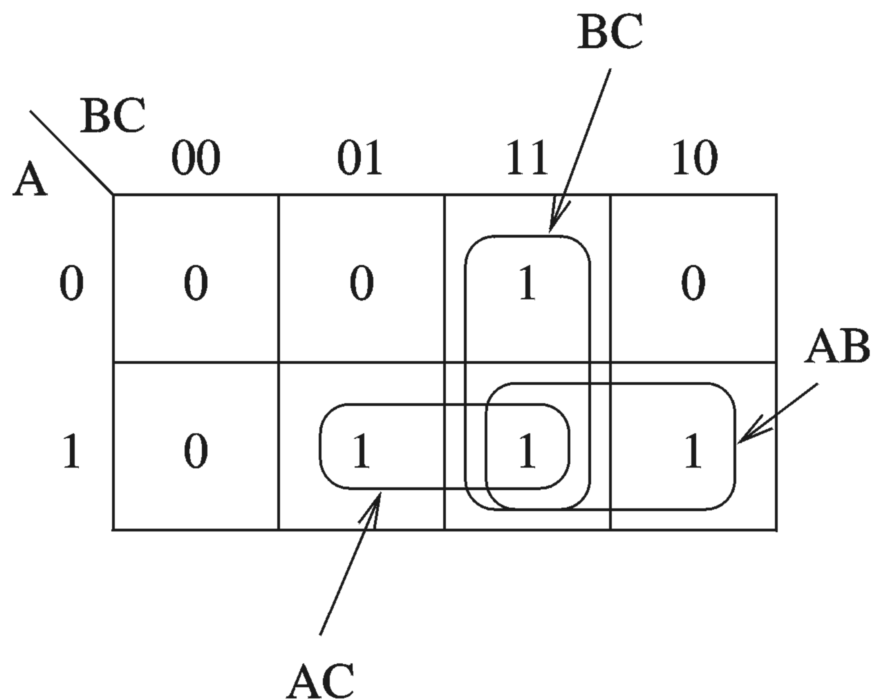
(b) Three-variable K-map

AB \ CD	00	01	11	10
00				
01				
11				
10				

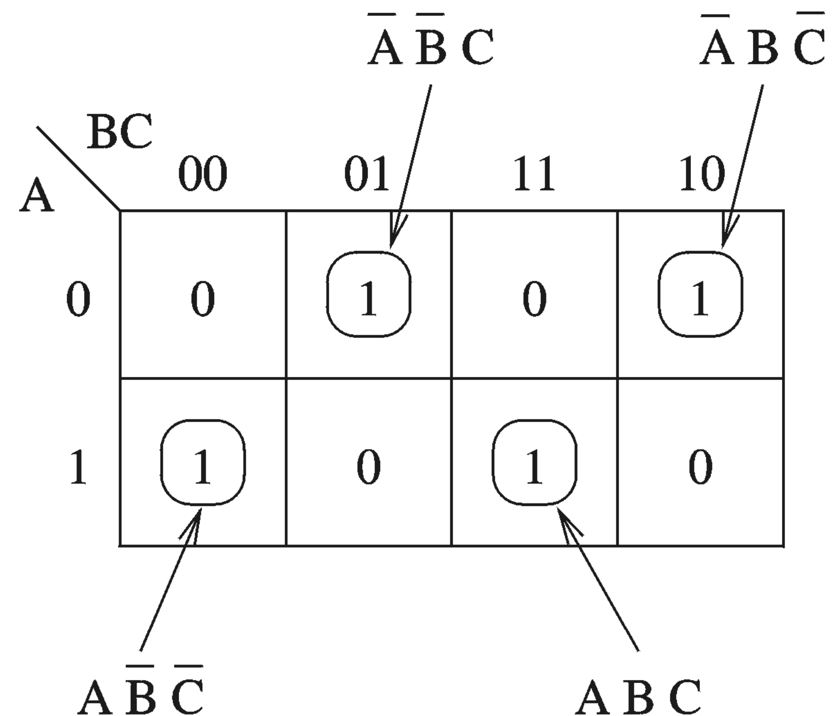
(c) Four-variable K-map

Karnaugh Map Method (cont.)

Simplification examples



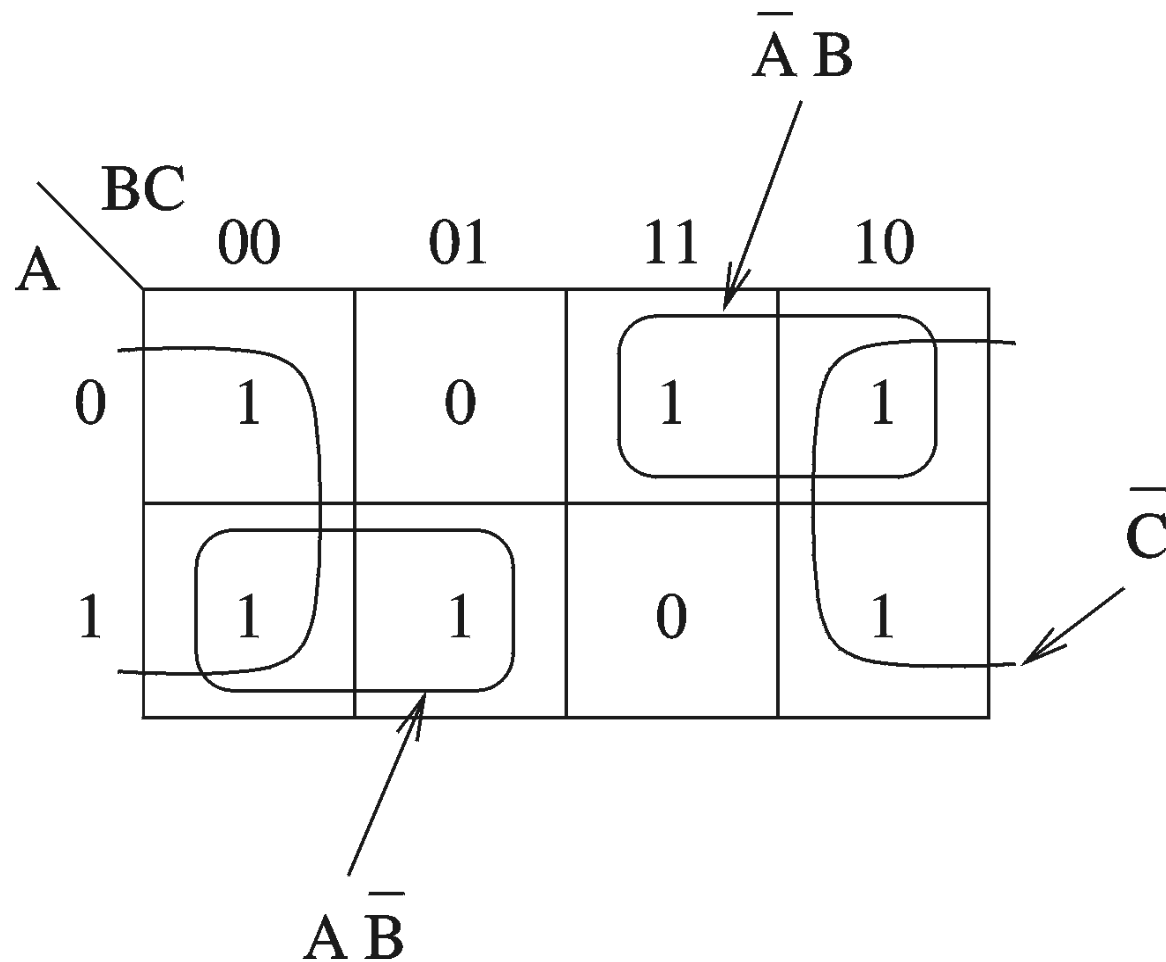
(a) Majority function



(b) Even-parity function

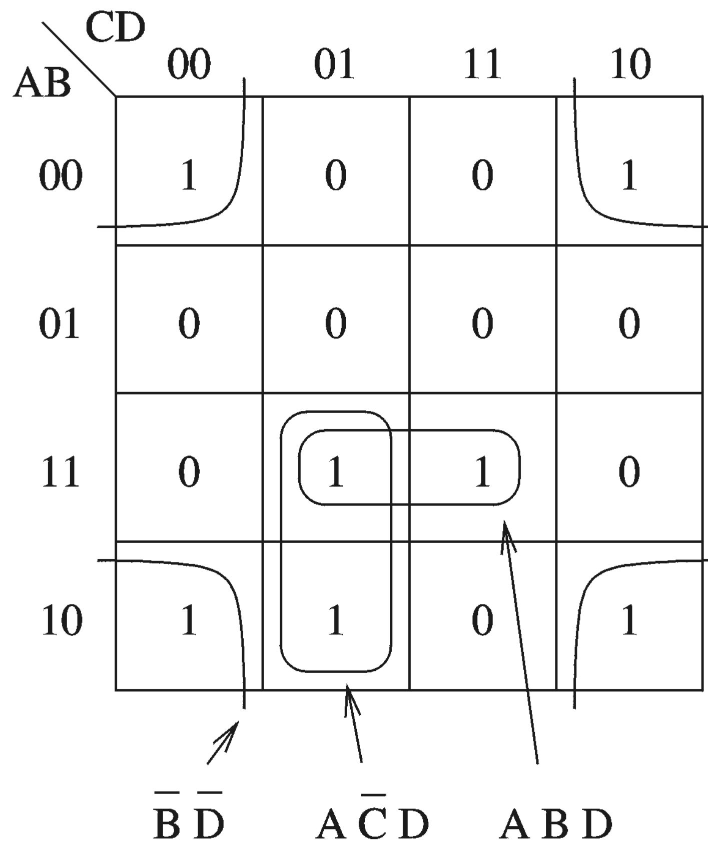
Karnaugh Map Method (cont.)

First and last columns/rows are adjacent

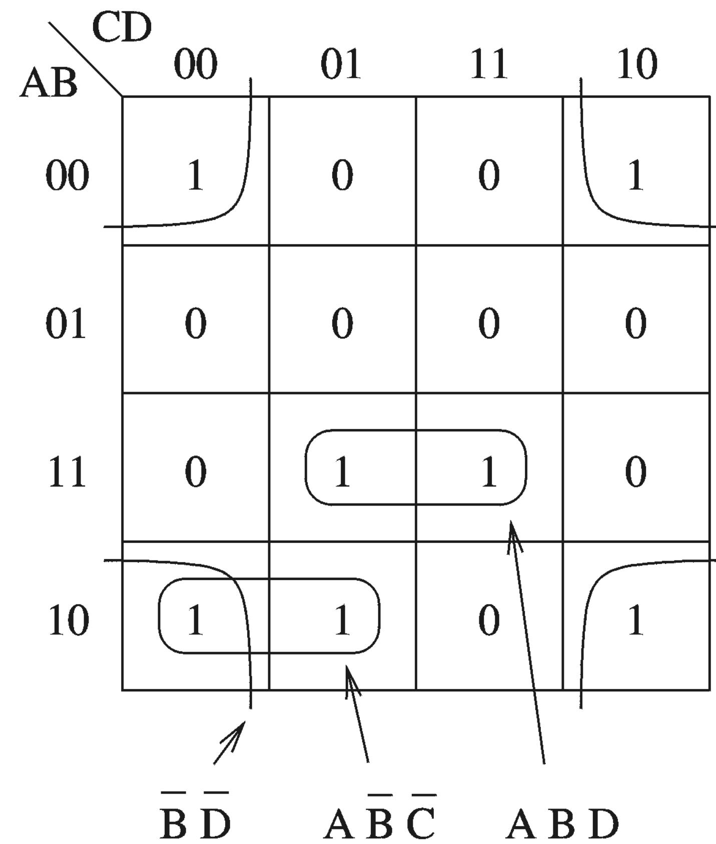


Karnaugh Map Method (cont.)

Minimal expression depends on groupings



(a)



(b)

Karnaugh Map Method (cont.)

No redundant groupings

AB \ CD	00	01	11	10
00	0	0	1	0
01	1	1	1	0
11	0	1	1	1
10	0	1	0	0

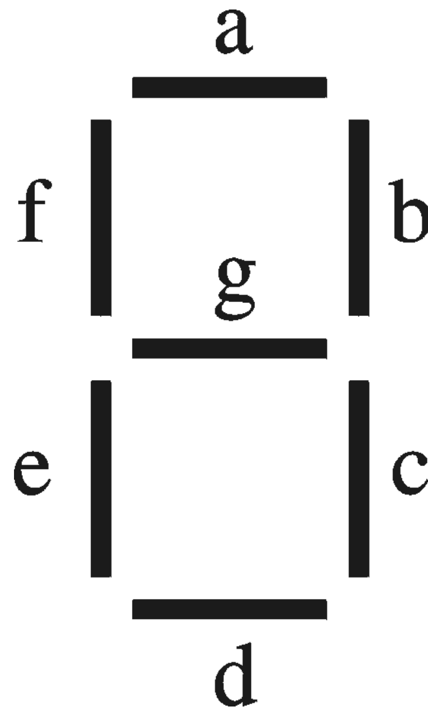
(a) Nonminimal simplification

AB \ CD	00	01	11	10
00	0	0	1	0
01	1	1	1	0
11	0	1	1	1
10	0	1	0	0

(b) Minimal simplification

Karnaugh Map Method (cont.)

- Example
  - Seven-segment display
  - Need to select the right LEDs to display a digit



Truth table for segment d

No	A	B	C	D	Seg.	No	A	B	C	D	Seg.
0	0	0	0	0	1	8	1	0	0	0	1
1	0	0	0	1	0	9	1	0	0	1	1
2	0	0	1	0	1	10	1	0	1	0	?
3	0	0	1	1	1	11	1	0	1	1	?
4	0	1	0	0	0	12	1	1	0	0	?
5	0	1	0	1	1	13	1	1	0	1	?
6	0	1	1	0	1	14	1	1	1	0	?
7	0	1	1	1	0	15	1	1	1	1	?

Karnaugh Map Method (cont.)

Don’t cares simplify the expression a lot

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	0	0	0	0
10	1	1	0	0

(a) Simplification with no don’t cares

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	d	d	d	d
10	1	1	d	d

(b) Simplification with don’t cares

Implementation Using NAND Gates

- Using NAND gates
  - Get an equivalent expression

$$AB + CD = \overline{\overline{AB + CD}}$$

  - Using de Morgan's law

$$AB + CD = \overline{\overline{AB} \cdot \overline{CD}}$$

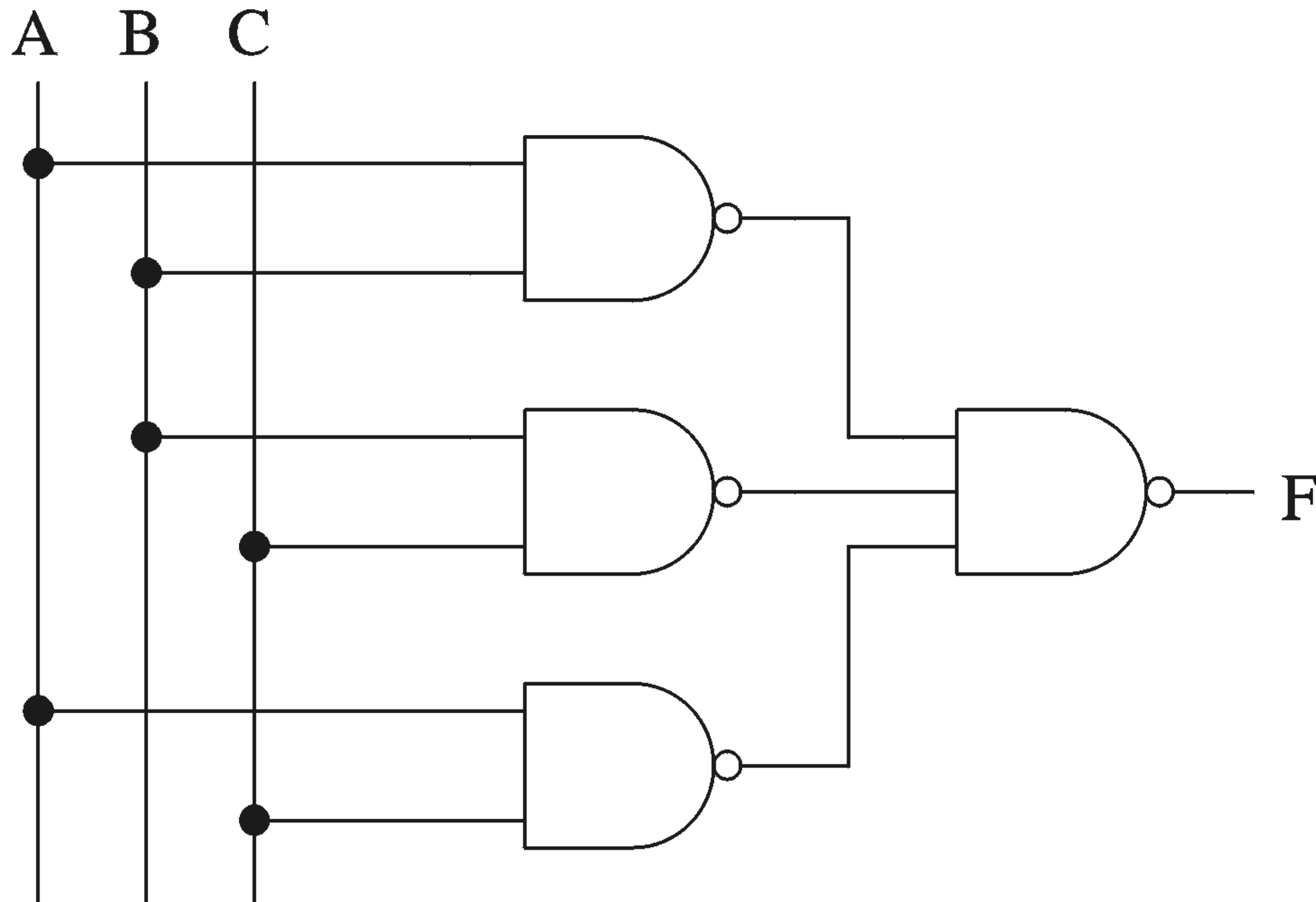
  - Can be generalized
    - Majority function

$$AB + BC + AC = \overline{\overline{AB} \cdot \overline{BC} \cdot \overline{AC}}$$

Idea: NAND Gates: Sum-of-Products, NOR Gates: Product-of-Sums

Implementation Using NAND Gates (cont.)

- Majority function



Introduction to Combinational Circuits

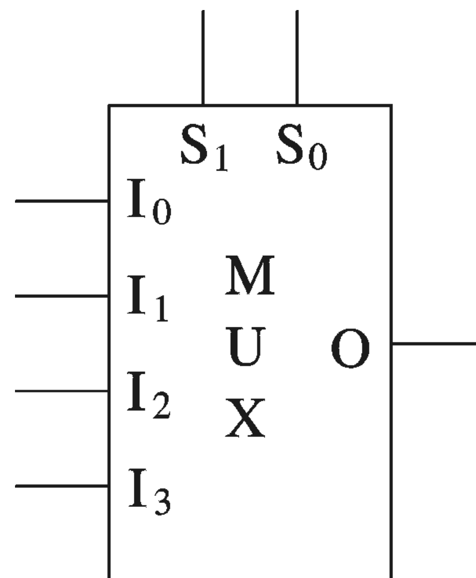
- Combinational circuits
  - Output depends only on the current inputs
- Combinational circuits provide a higher level of abstraction
  - Help in reducing design complexity
  - Reduce chip count
- We look at some useful combinational circuits

Multiplexers

- Multiplexer
  - 2^n data inputs
  - n selection inputs
  - a single output

- Selection input determines the input that should be connected to the output

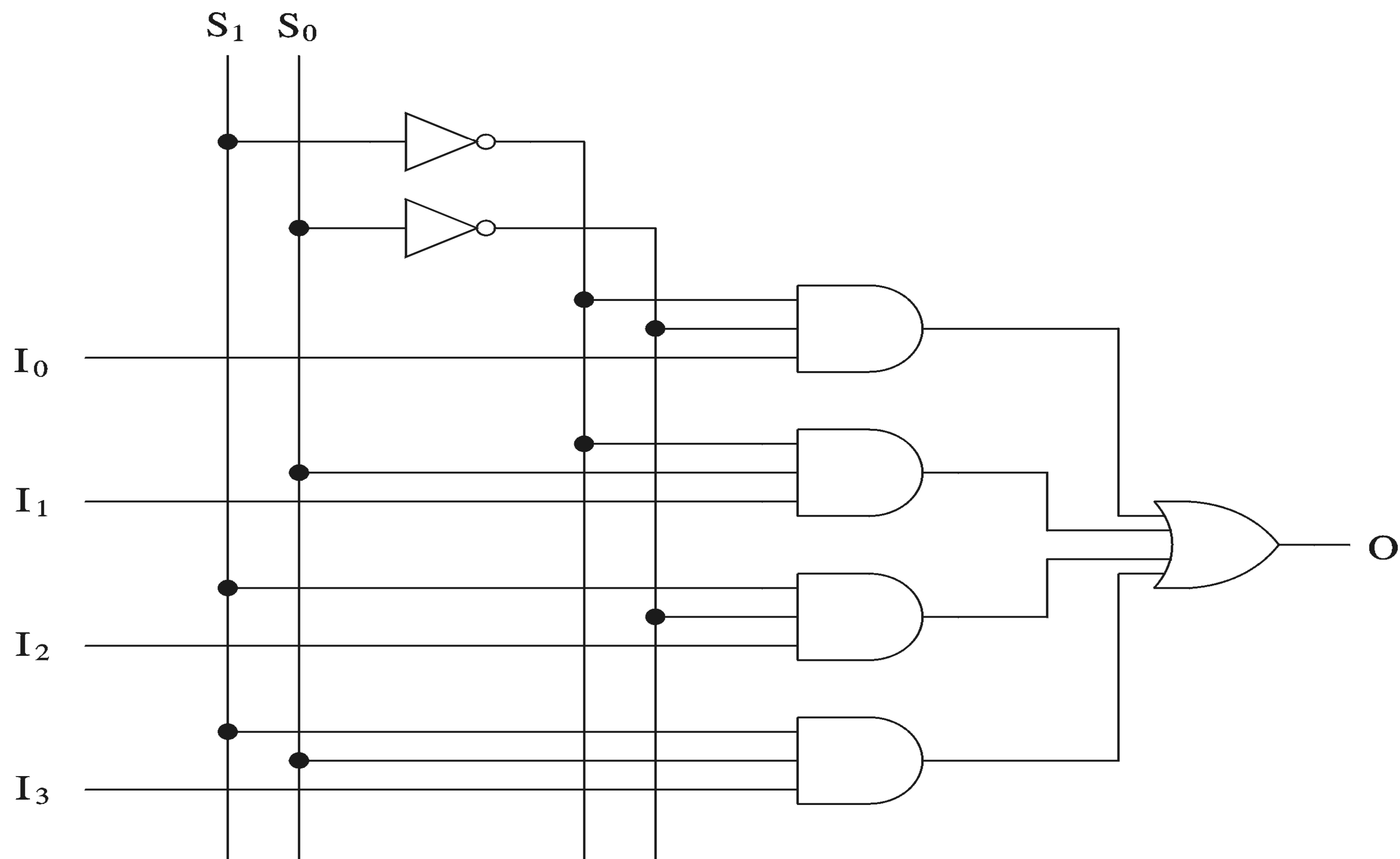
4-data input MUX



S_1	S_0	O
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

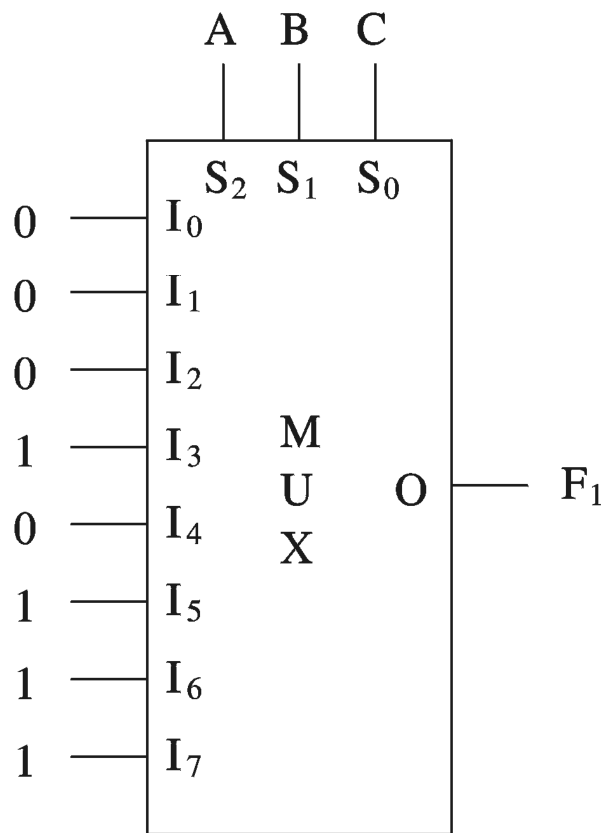
Multiplexers (cont.)

4-data input MUX implementation

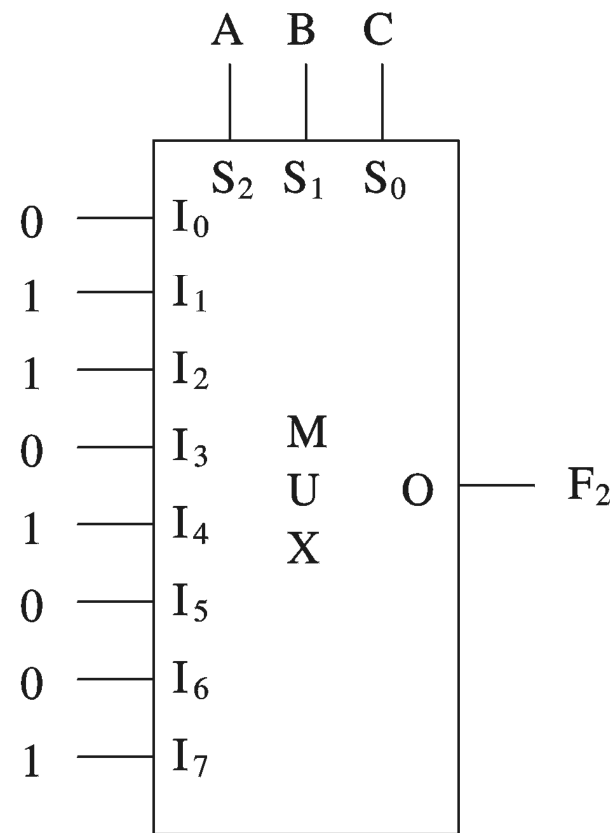


Multiplexers (cont.)

MUX implementations



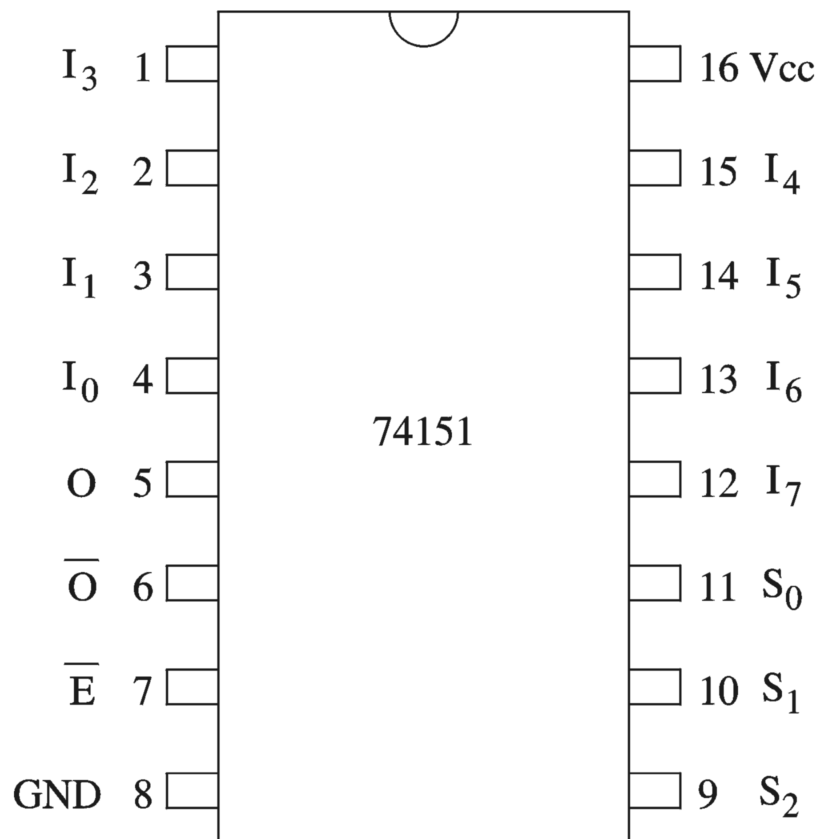
Majority function



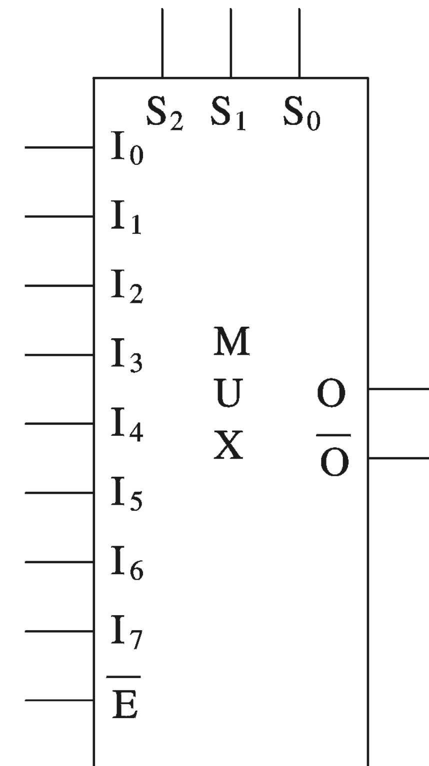
Even-parity function

Multiplexers (cont.)

Example chip: 8-to-1 MUX



(a) Connection diagram



(b) Logic symbol

Multiplexers (cont.)

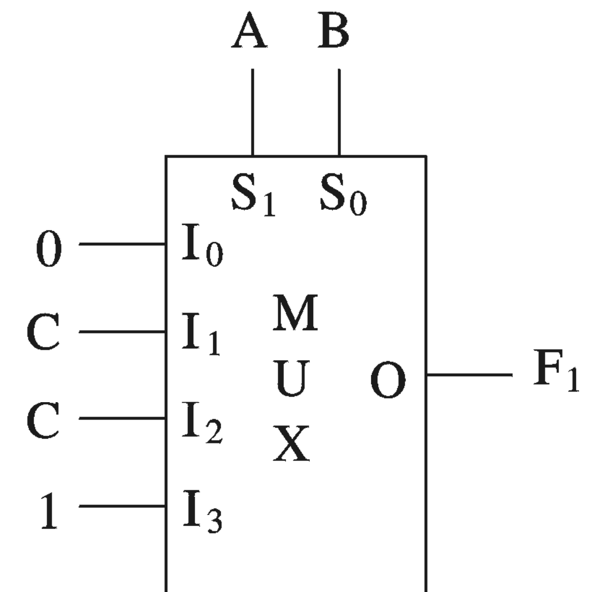
Efficient implementation: Majority function

Original truth table

A	B	C	F_1
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

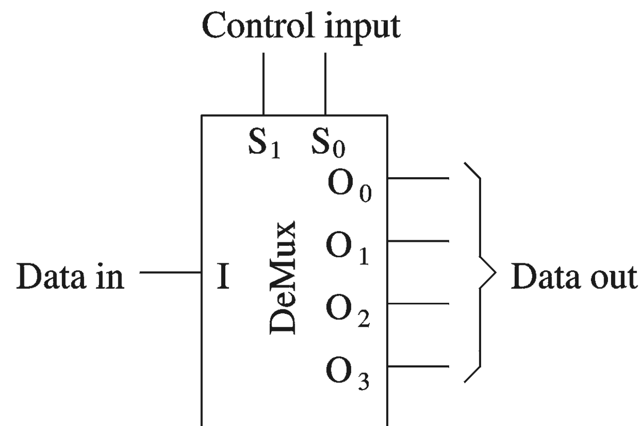
New truth table

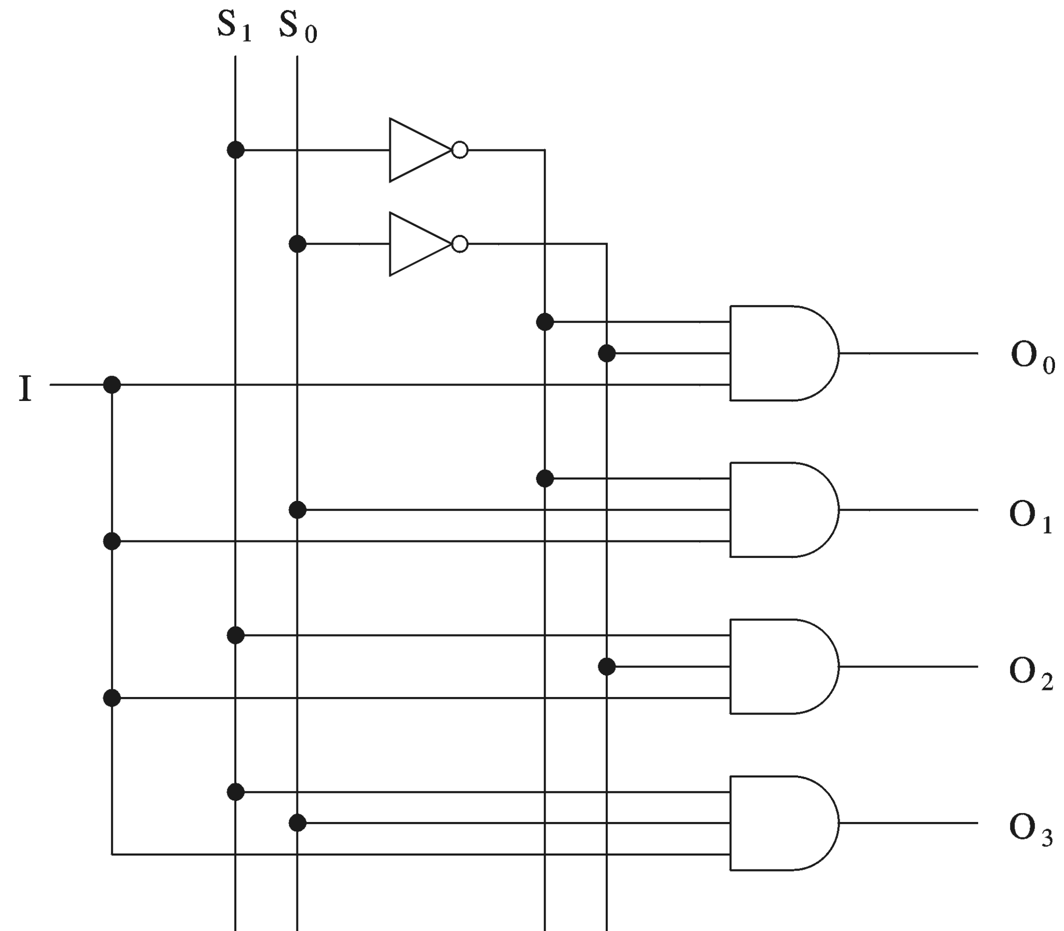
A	B	F_1
0	0	0
0	1	C
1	0	C
1	1	1



Demultiplexers (DeMUX)

- Demultiplexer
  - a single input
  - n selection inputs
  - 2^n outputs

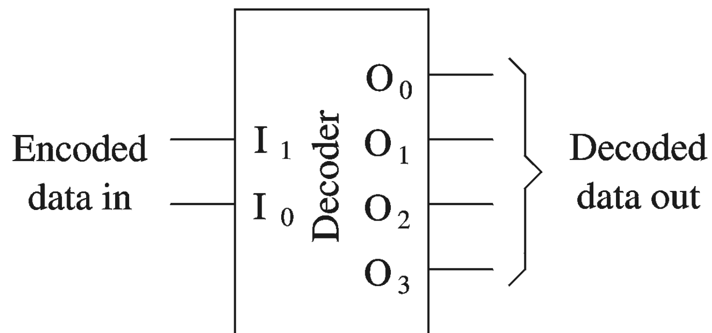


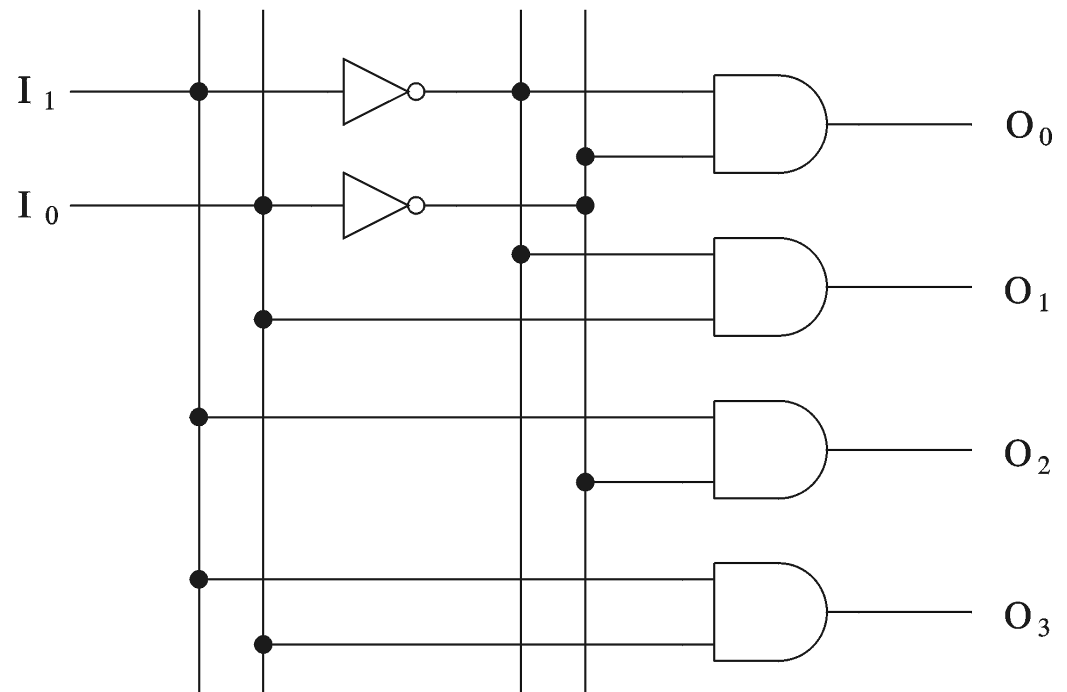


Decoders

- Decoder selects one-out-of-N inputs

I_1	I_0	O_3	O_2	O_1	O_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

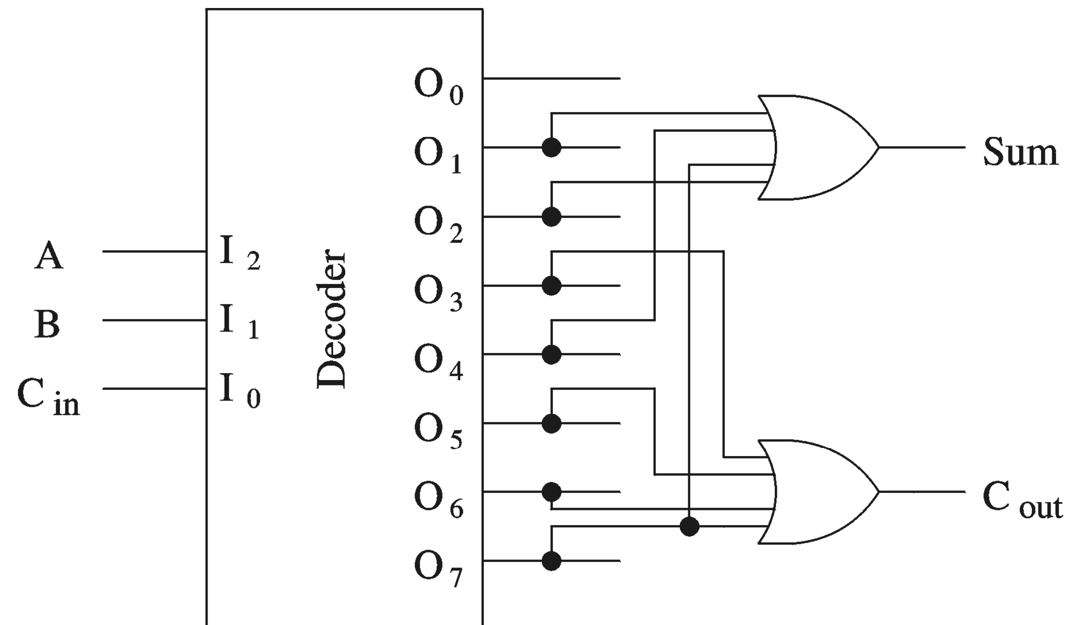




Decoders (cont.)

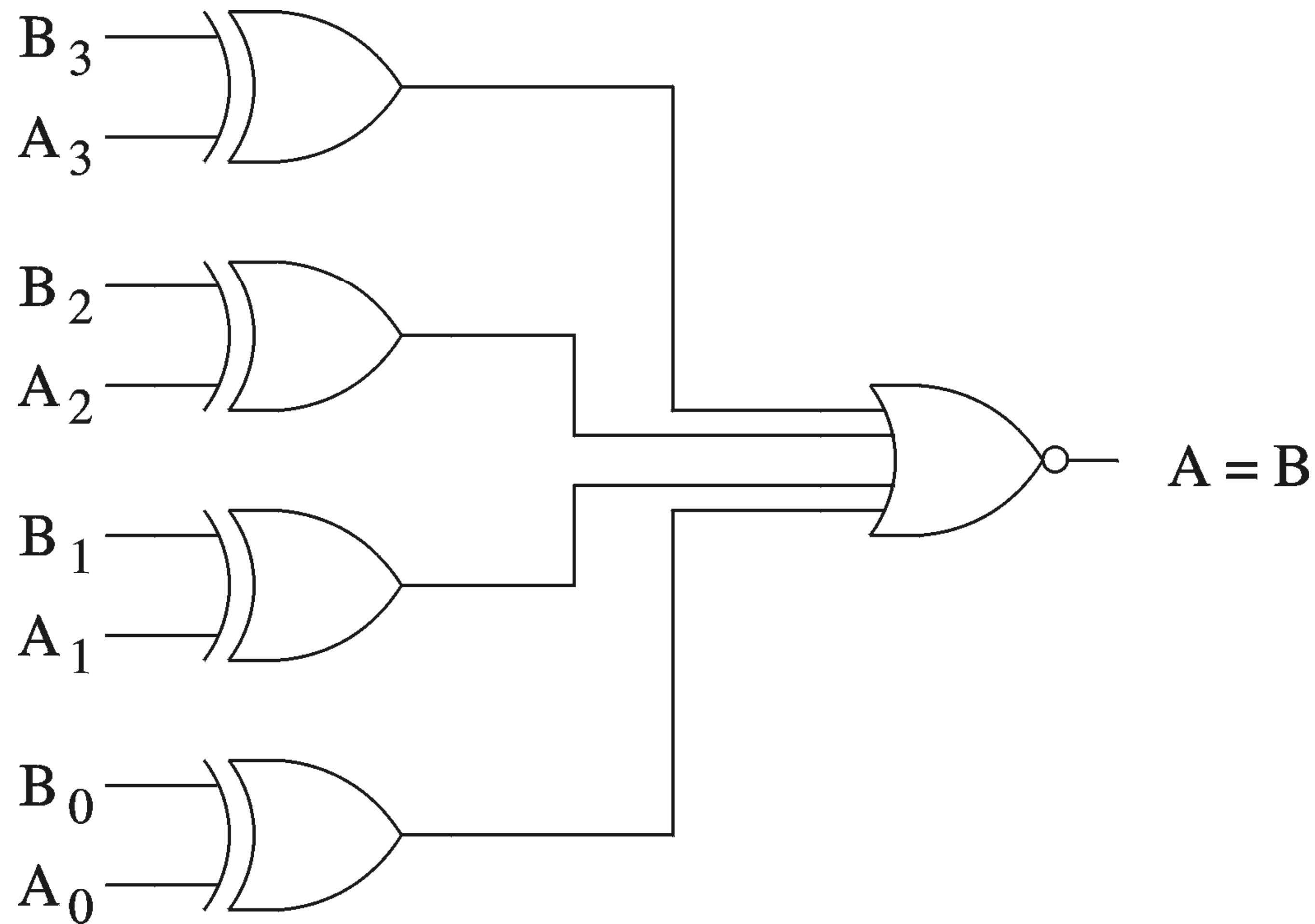
Logic function implementation

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



Comparator

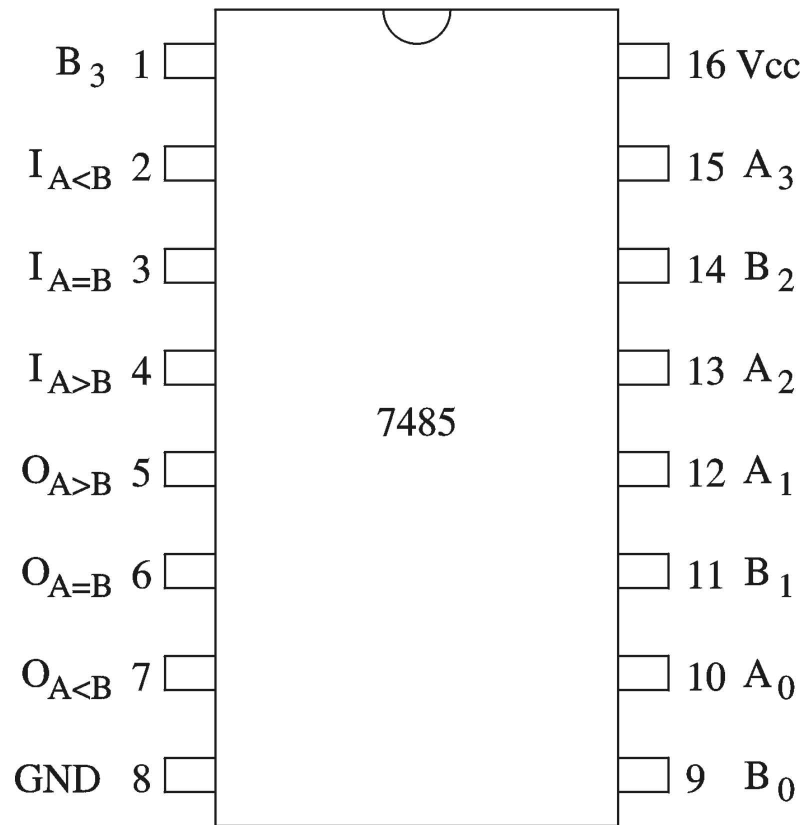
- Used to implement comparison operators (= , > , < , ≥ , ≤)



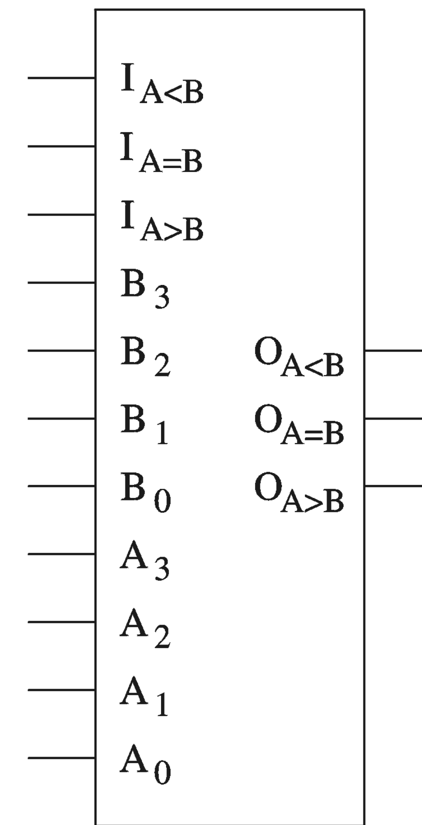
Comparator (cont.)

A=B: $O_x = I_x$ (x=A<B, A=B, & A>B)

4-bit magnitude comparator chip



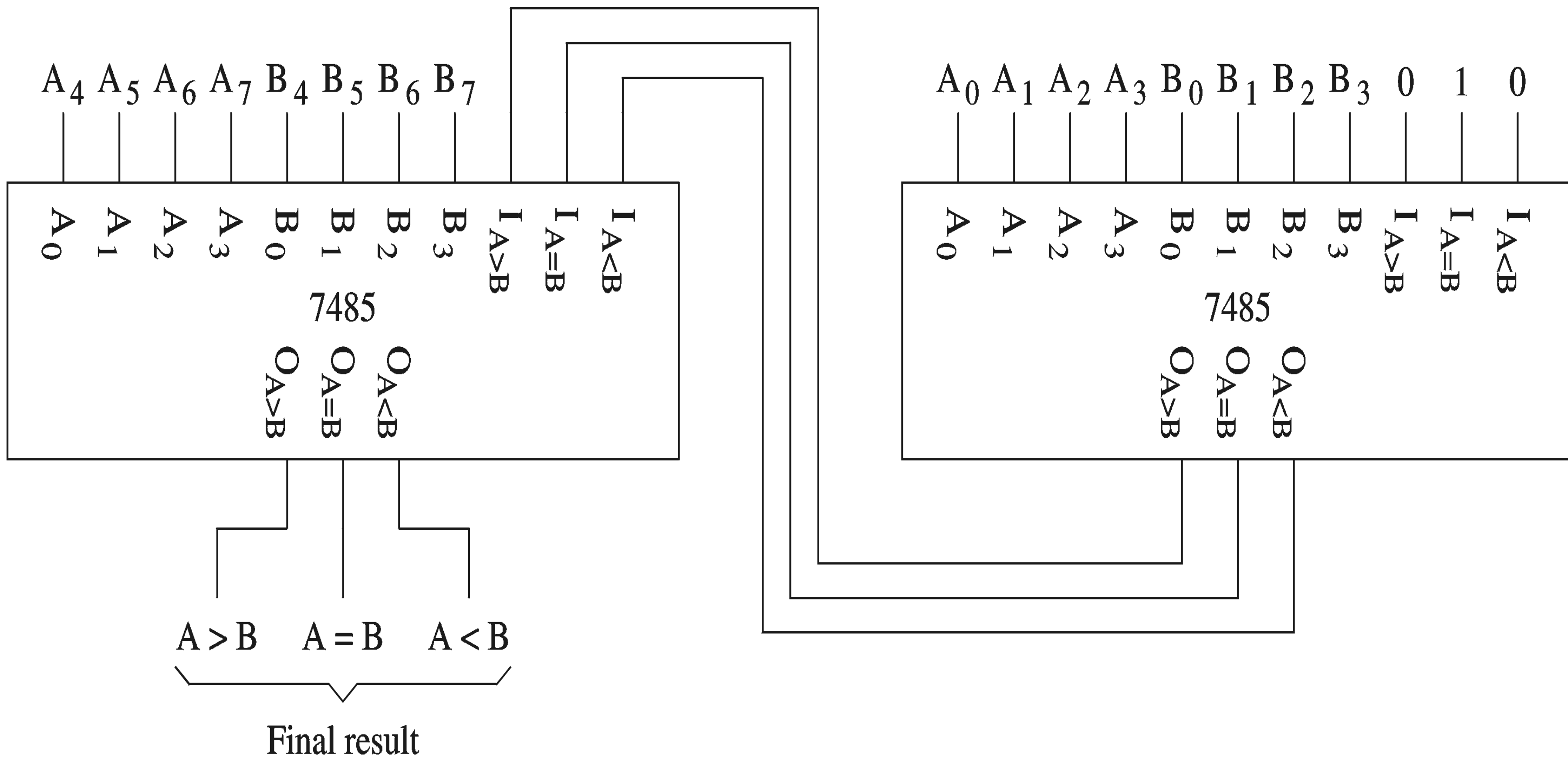
(a) Connection diagram



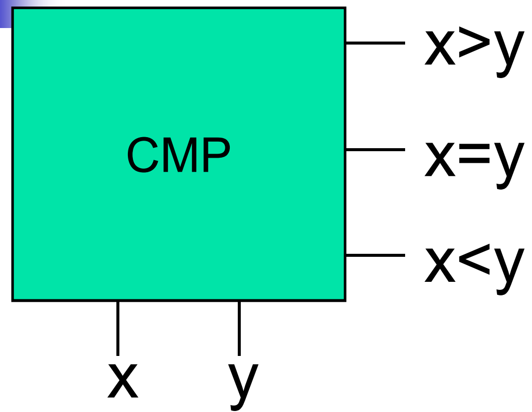
(b) Logic symbol

Comparator (cont.)

Serial construction of an 8-bit comparator

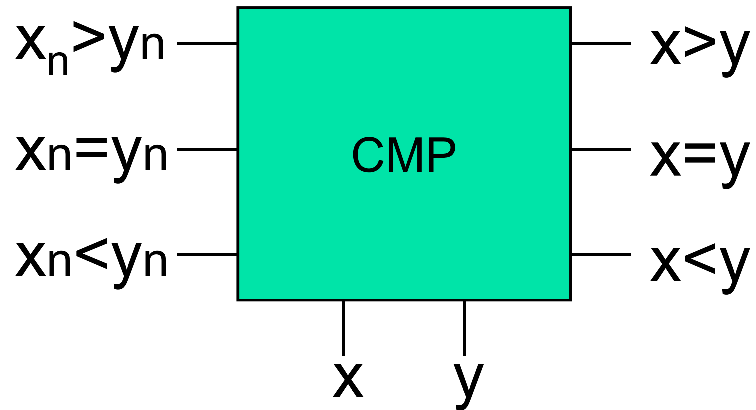


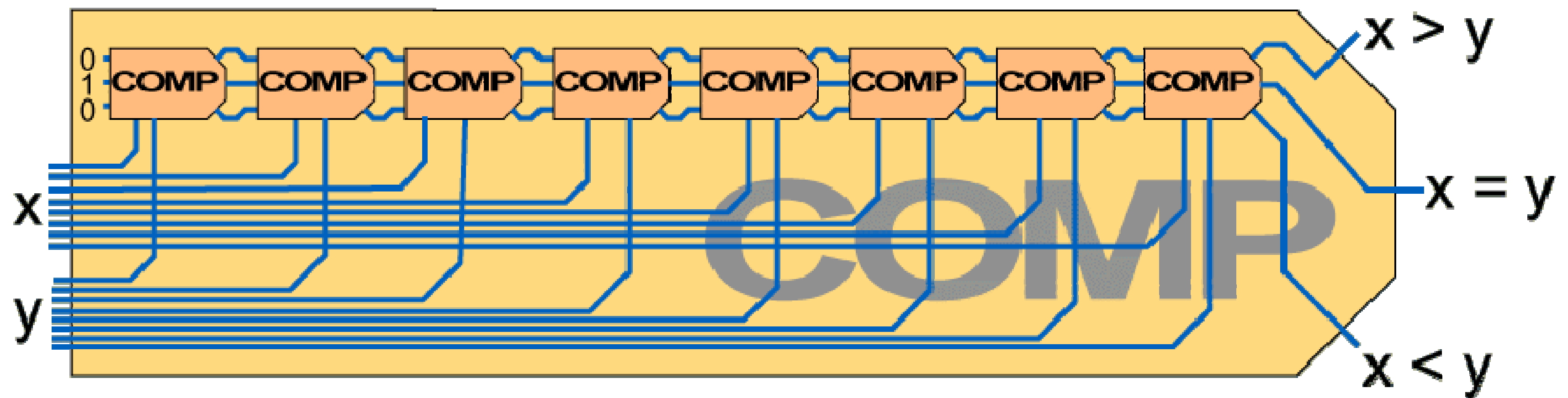
1-bit Comparator



x	y	x>y	x=y	x<y

8-bit comparator



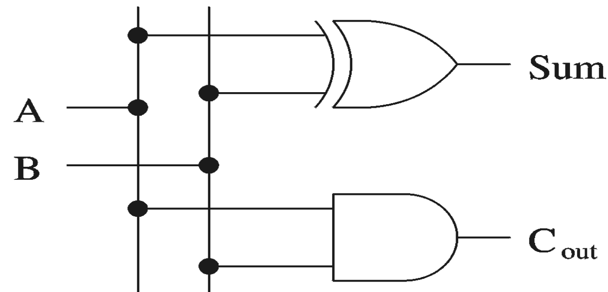


Adders

- Half-adder
  - Adds two bits
    - Produces a *sum* and *carry*
  - Problem: Cannot use it to build larger inputs
- Full-adder
  - Adds three 1-bit values
    - Like half-adder, produces a *sum* and *carry*
  - Allows building N-bit adders
    - Simple technique
      - Connect C_{out} of one adder to C_{in} of the next
    - These are called *ripple-carry adders*

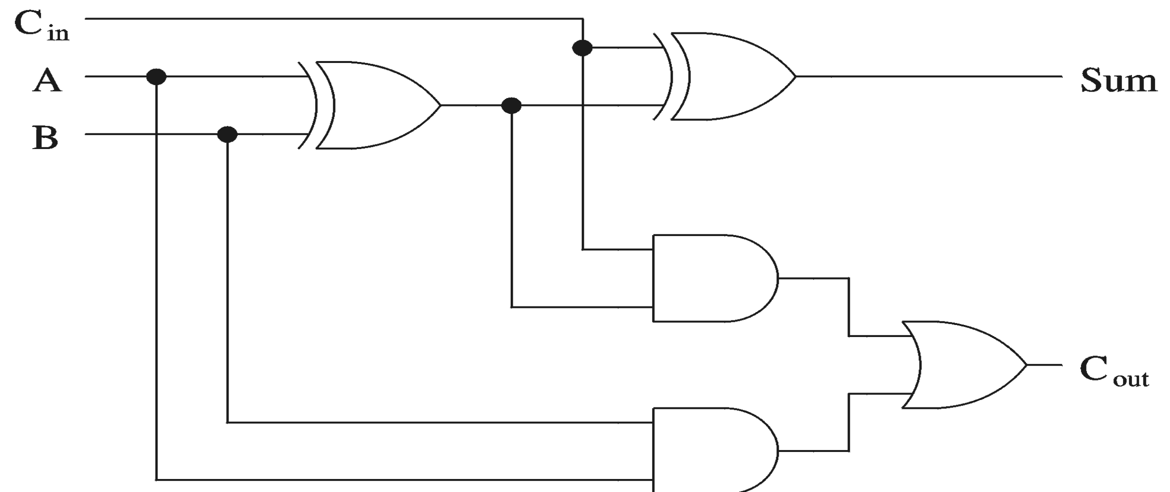
Adders (cont.)

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



(a) Half-adder truth table and implementation

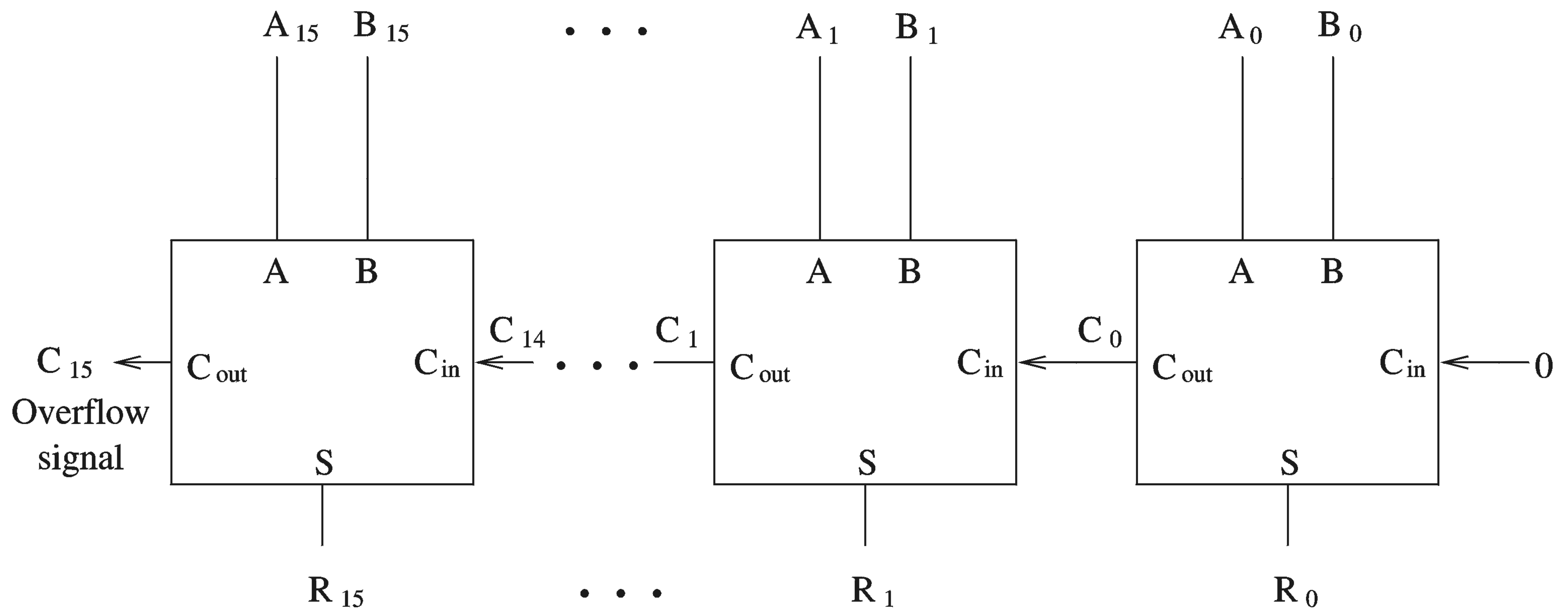
A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



(b) Full-adder truth table and implementation

Adders (cont.)

A 16-bit ripple-carry adder



Adders (cont.)

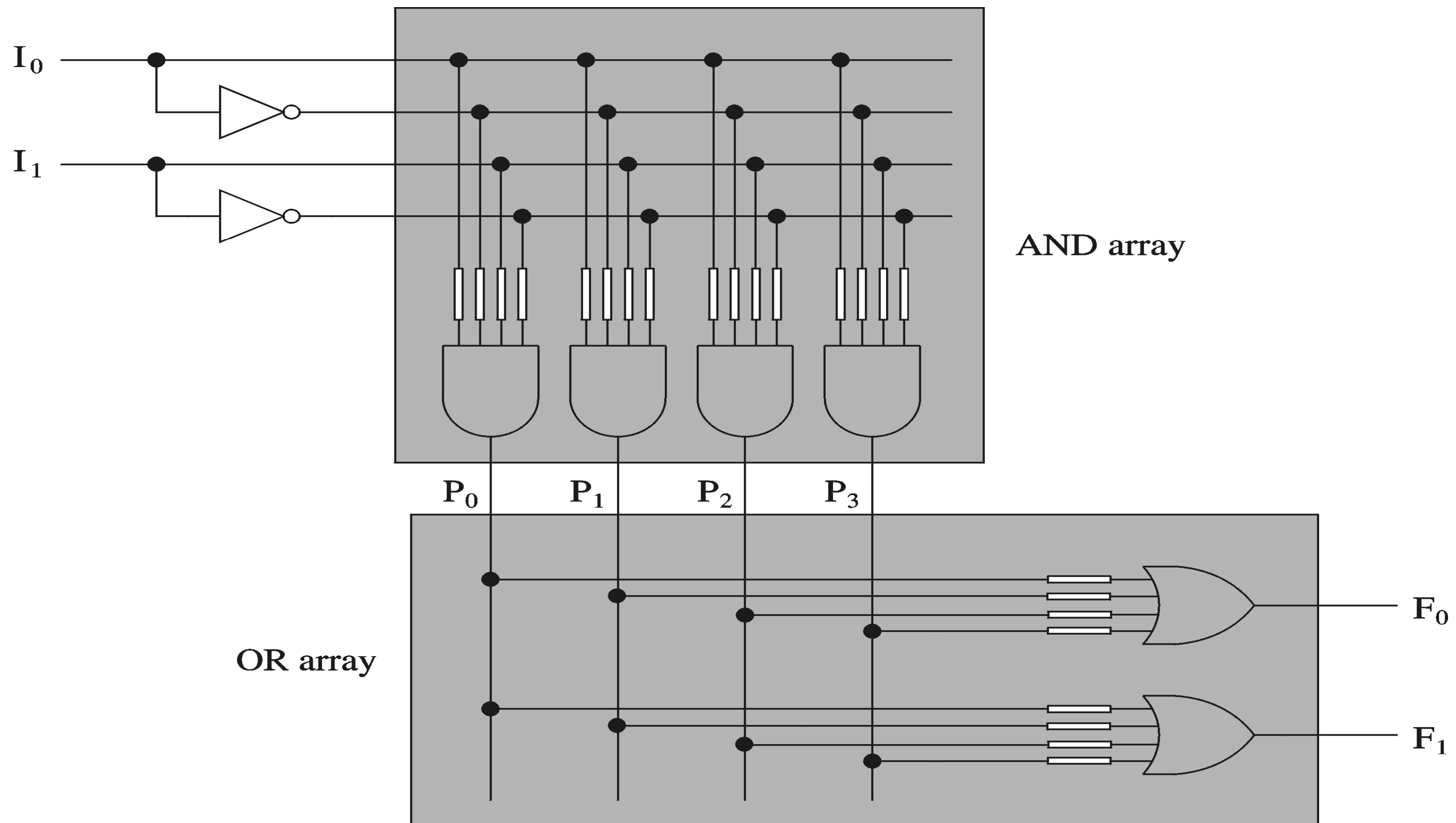
- Ripple-carry adders can be slow
  - Delay proportional to number of bits
- Carry lookahead adders
  - Eliminate the delay of ripple-carry adders
  - Carry-ins are generated independently
    - $C_0 = A_0 \; B_0$
    - $C_1 = A_0 \; B_0 \; A_1 + A_0 \; B_0 \; B_1 + A_1 \; B_1$
    - . . .
  - Requires complex circuits
  - Usually, a combination carry lookahead and ripple-carry techniques are used

Programmable Logic Arrays

- PLAs
  - Implement sum-of-product expressions
    - No need to simplify the logical expressions
  - Take *N* inputs and produce *M* outputs
    - Each input represents a logical variable
    - Each output represents a logical function output
  - Internally uses
    - An AND array
      - Each AND gate receives 2*N* inputs
        - *N* inputs and their complements
    - An OR array

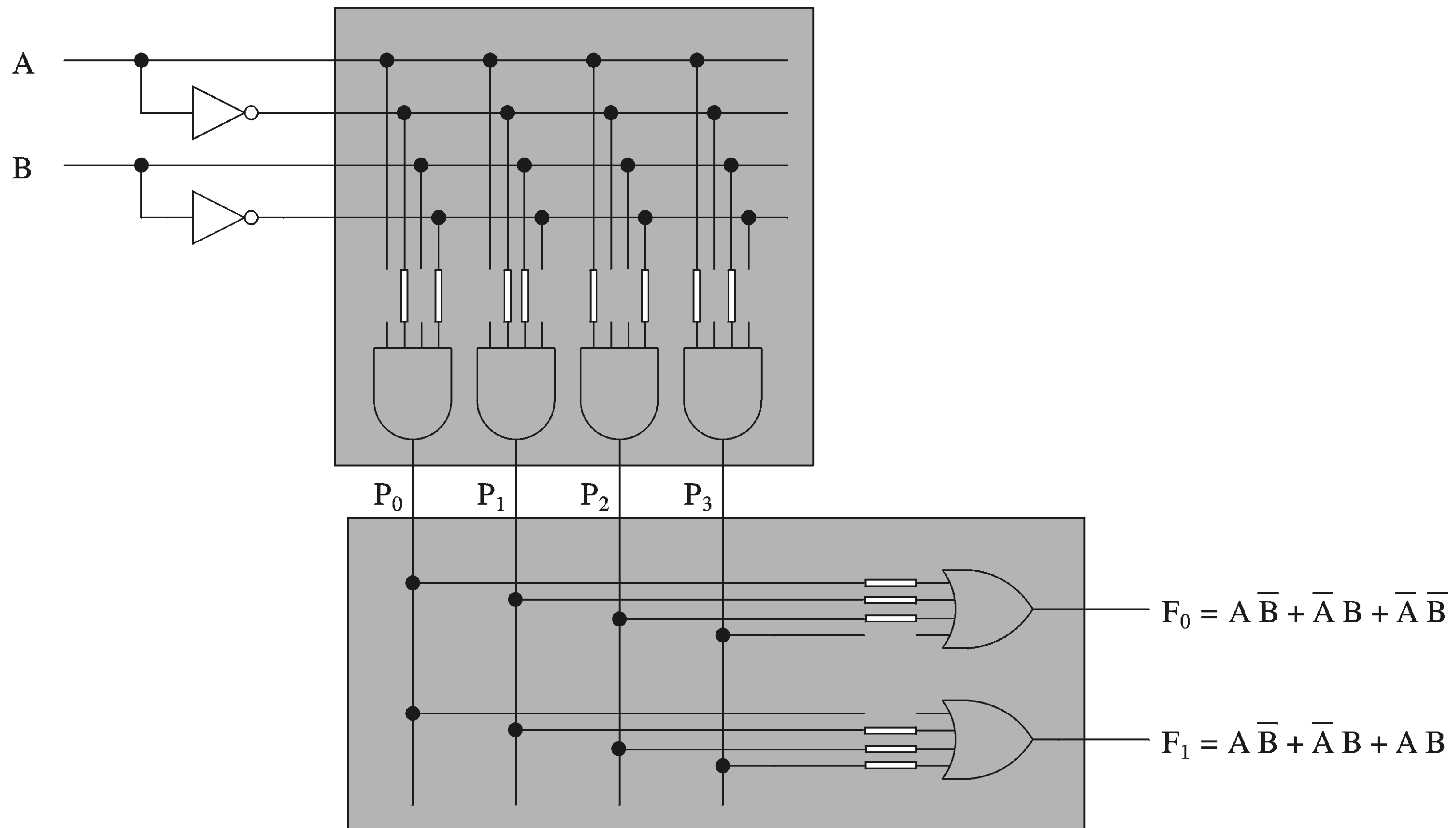
Programmable Logic Arrays (cont.)

A blank PLA with 2 inputs and 2 outputs



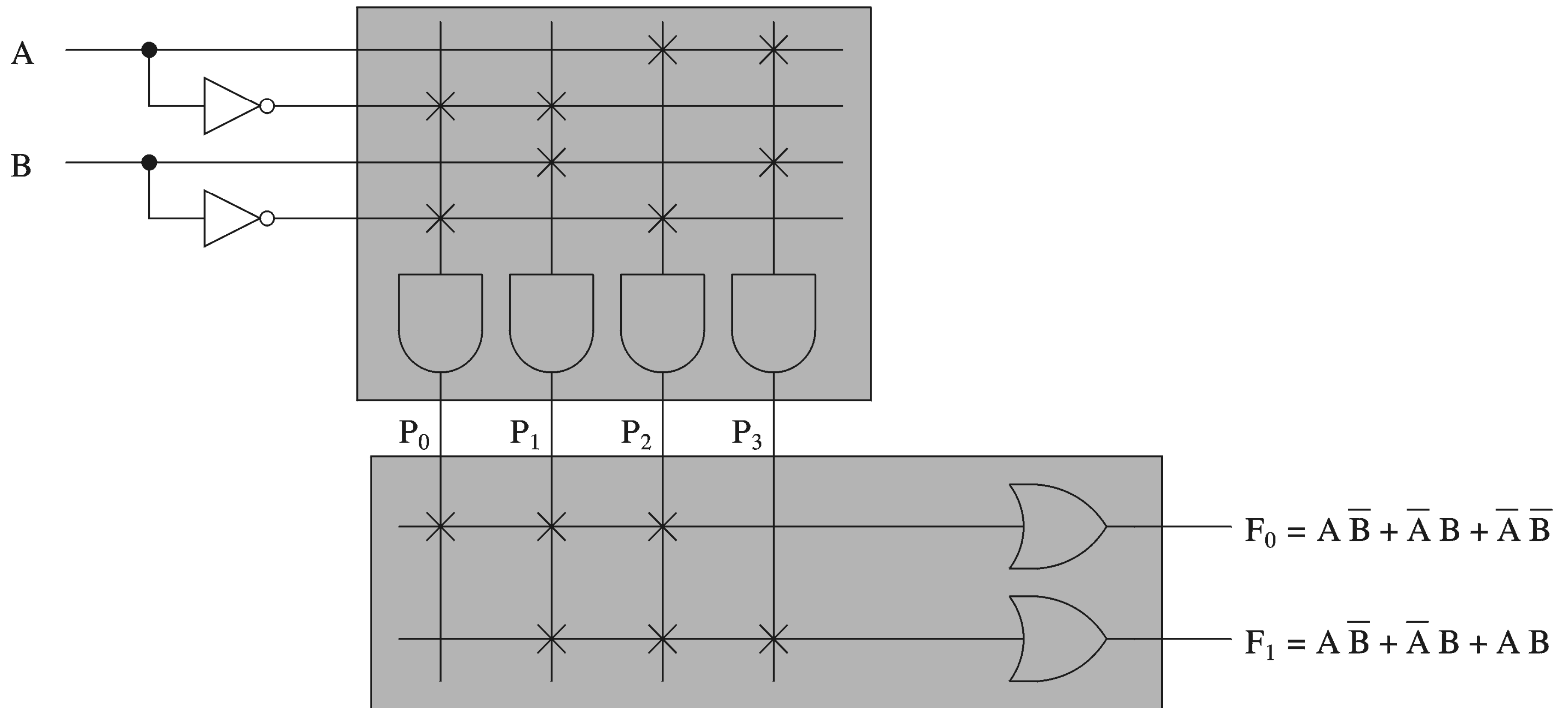
Programmable Logic Arrays (cont.)

Implementation examples



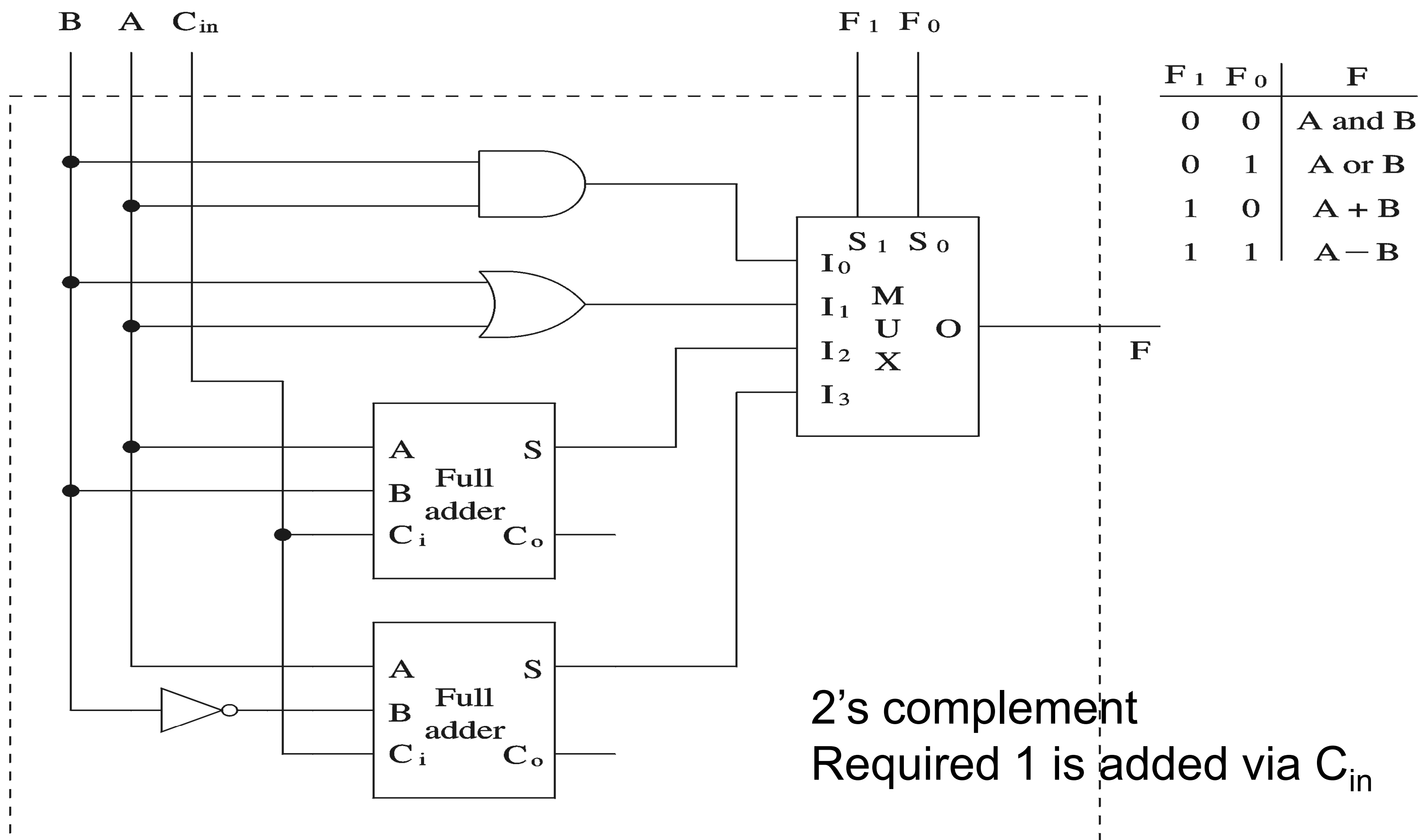
Programmable Logic Arrays (cont.)

Simplified notation



1-bit Arithmetic and Logic Unit

Preliminary ALU design

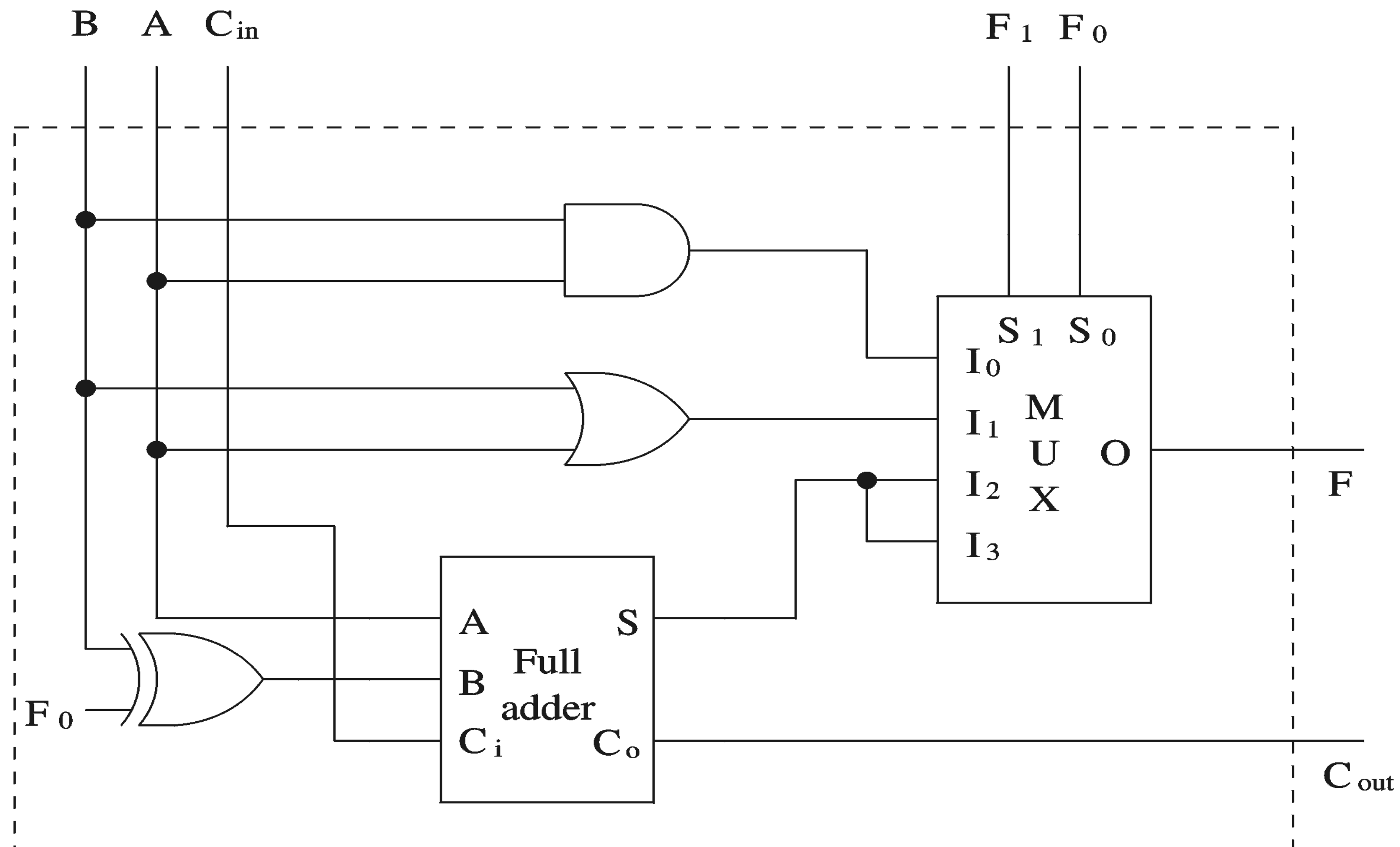


F_1	F_0	F
0	0	A and B
0	1	A or B
1	0	A + B
1	1	A − B

2's complement
Required 1 is added via C_{in}

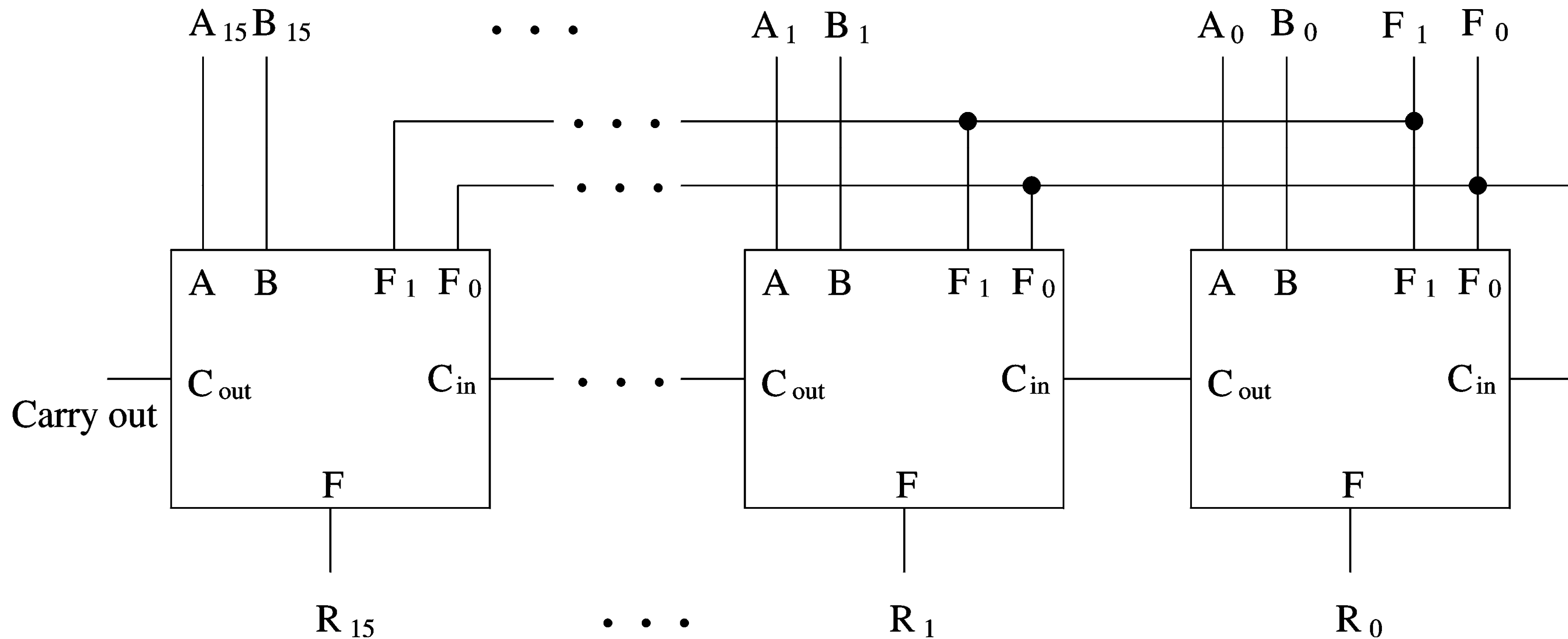
1-bit Arithmetic and Logic Unit (cont.)

Final design



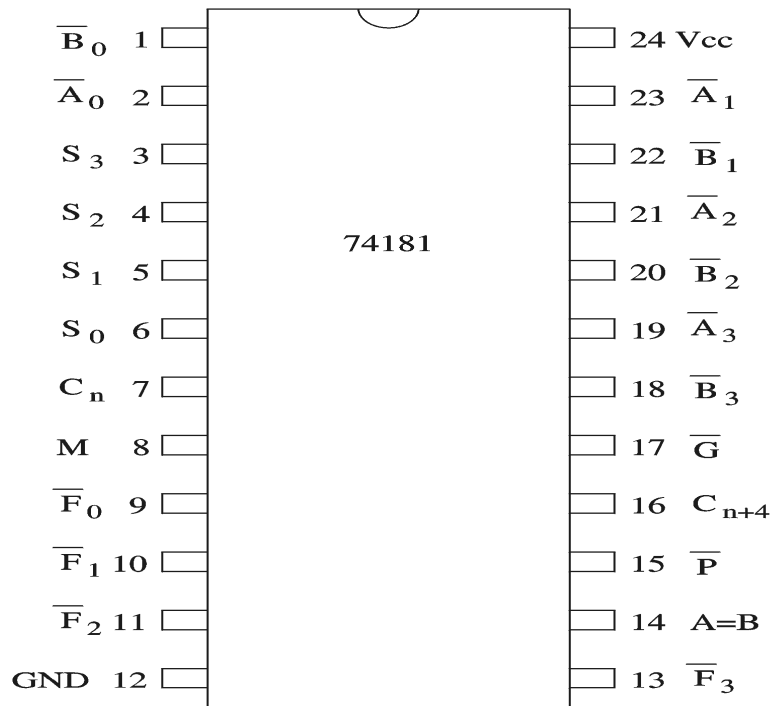
Arithmetic and Logic Unit (cont.)

16-bit ALU

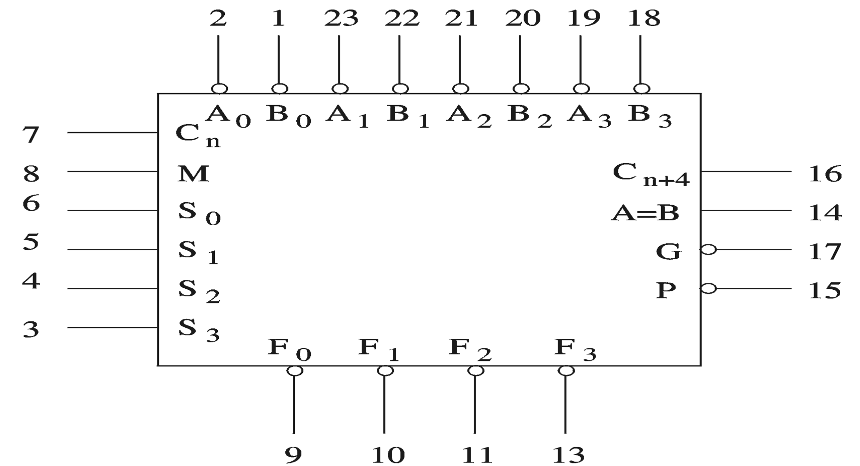


Arithmetic and Logic Unit (cont'd)

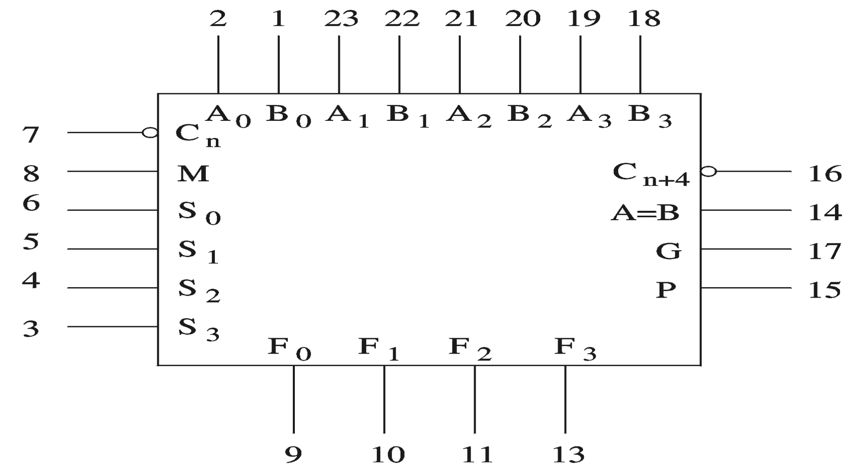
4-bit ALU



(a) Connection diagram



(b) Active low operands



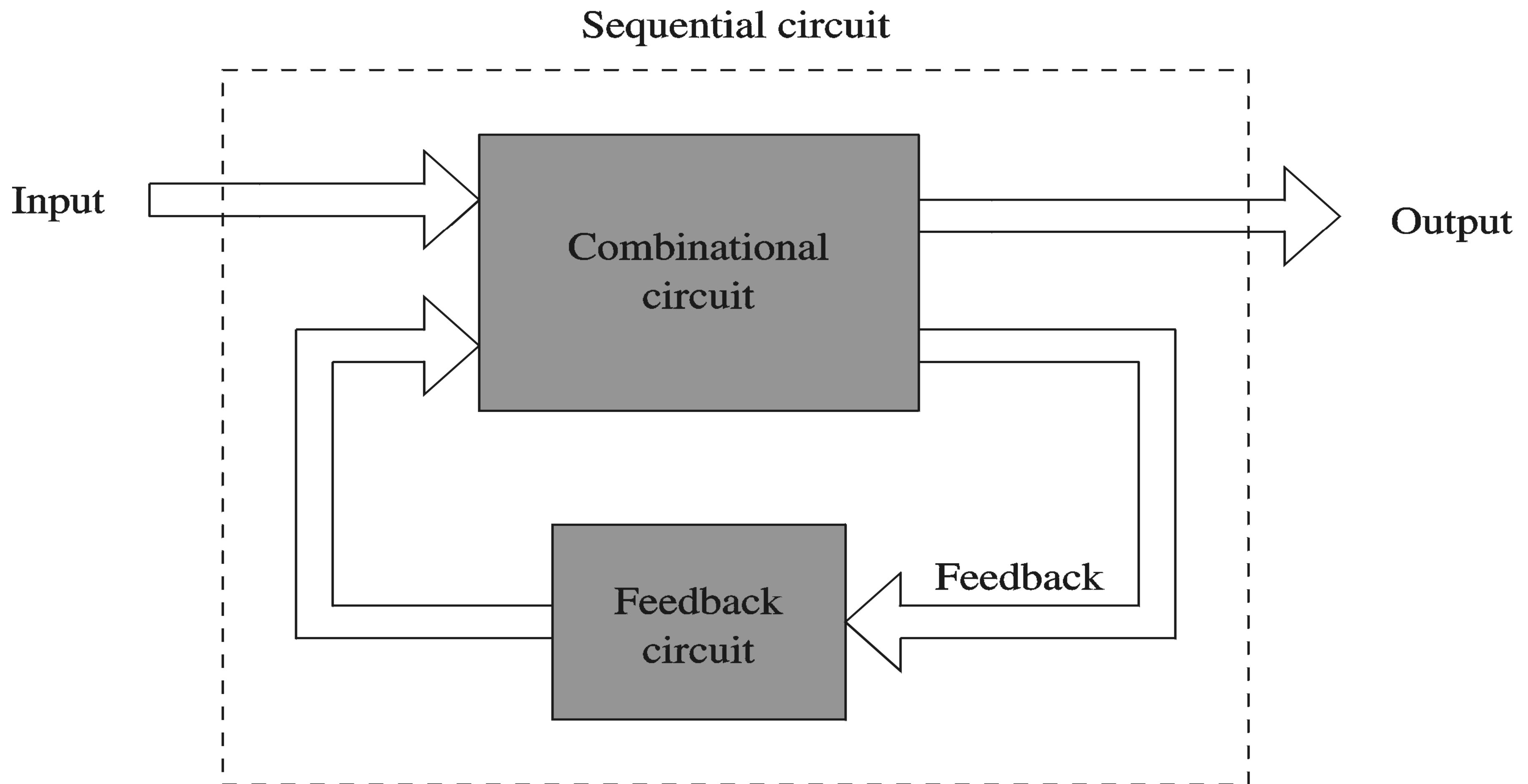
(c) Active high operands

Introduction to Sequential Circuits

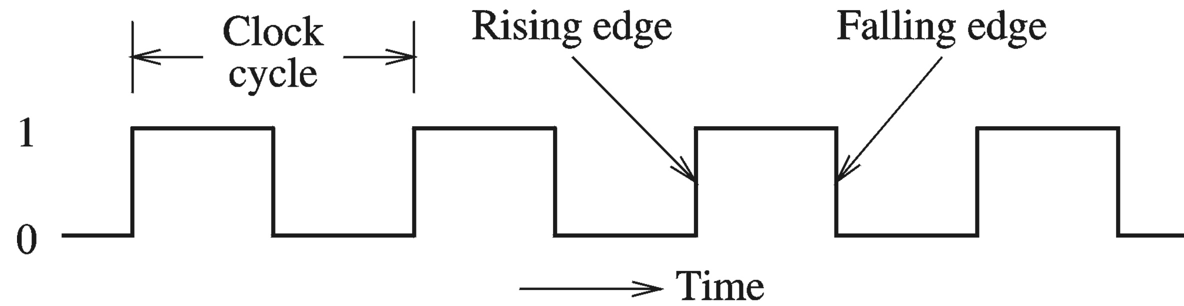
- Output depends on current as well as past inputs
  - Depends on the history
  - Have "memory" property
- Sequential circuit consists of
    - Combinational circuit
    - Feedback circuit
  - Past input is encoded into a set of state variables
    - Uses feedback (to feed the state variables)
      - Simple feedback
      - Uses flip flops

Introduction (cont.)

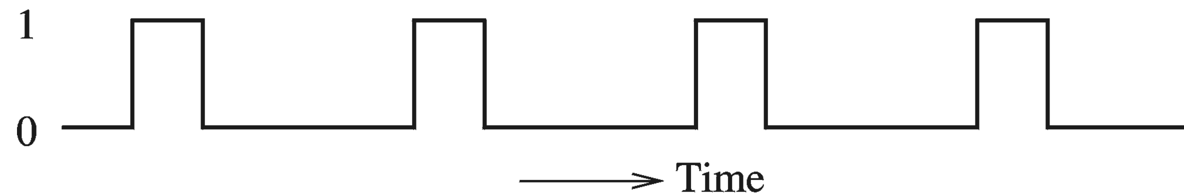
Main components of a sequential circuit



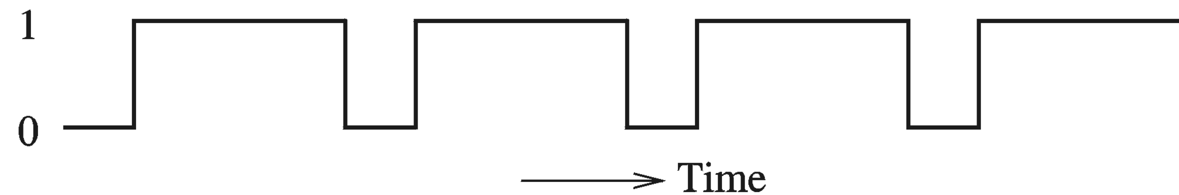
Clock Signal



(a) Symmetric



(b) Smaller ON period

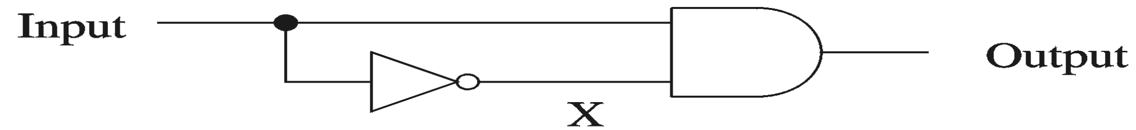


(c) Smaller OFF period

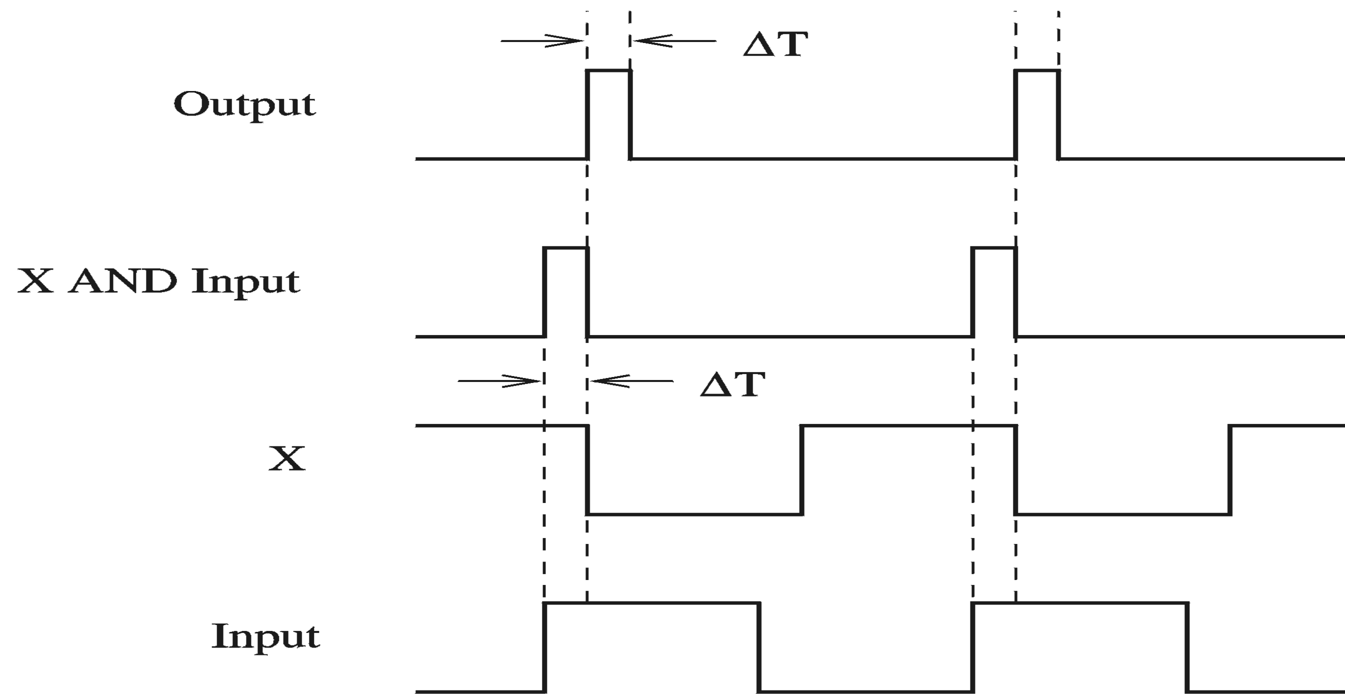
Clock Signal (cont.)

- Clock serves two distinct purposes
  - Synchronization point
    - Start of a cycle
    - End of a cycle
    - Intermediate point at which the clock signal changes levels
  - Timing information
    - Clock period, ON, and OFF periods
- Propagation delay
  - Time required for the output to react to changes in the inputs

Clock Signal (cont.)



(a) Circuit diagram

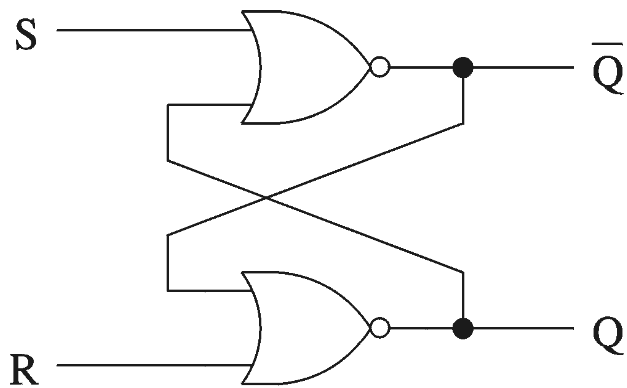


(b) Timing diagram

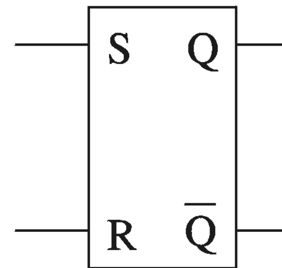
SR Latches

- Can remember a bit
- Level-sensitive (not edge-sensitive)

A NOR gate implementation of SR latch



(a) Circuit diagram



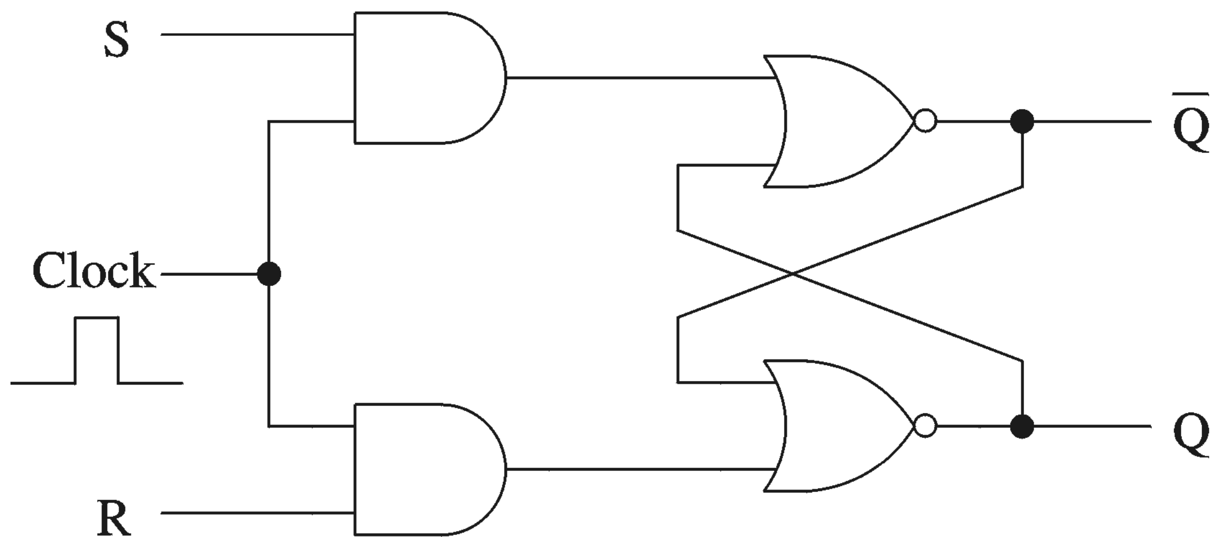
(b) Logic symbol

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
~~1~~	~~1~~	~~0~~

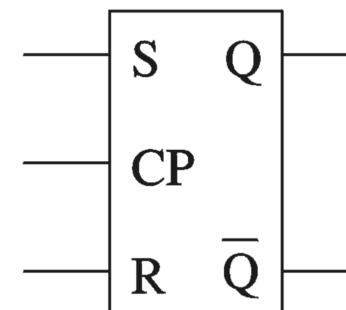
(c) Truth table

SR Latches (cont.)

- SR latch outputs follow inputs
- In clocked SR latch, outputs respond at specific instances
  - Uses a clock signal



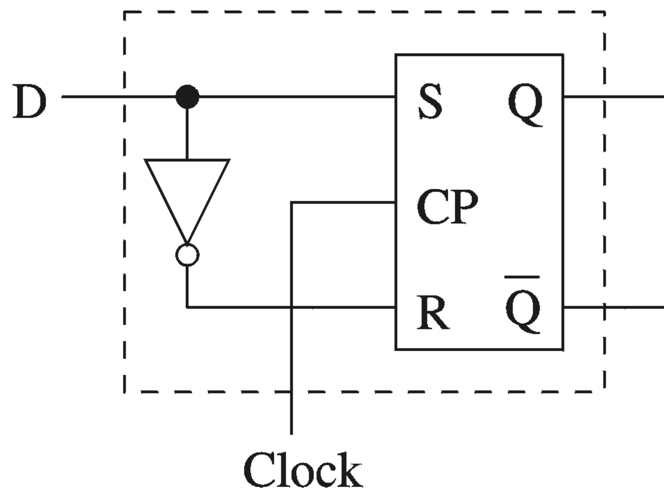
(a) Circuit diagram



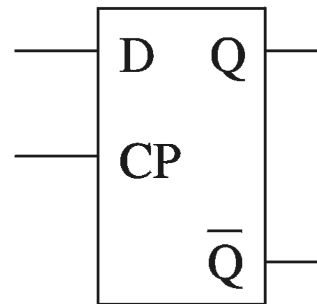
(b) Logic symbol

D Latches

- D Latch
  - Avoids the SR = 11 state



(a) Circuit diagram



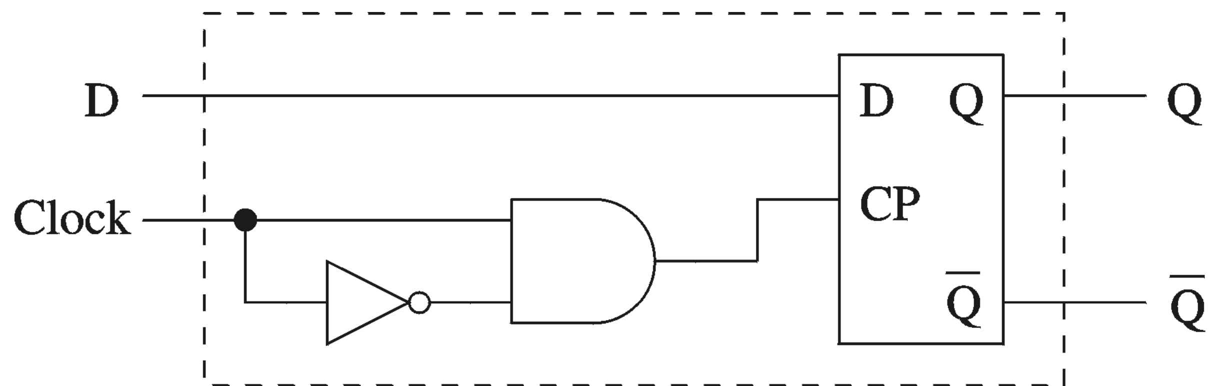
(b) Logic symbol

D	Q_{n+1}
0	0
1	1

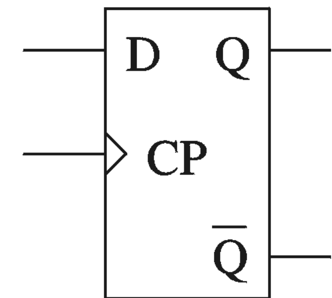
(c) Truth table

Positive Edge-Triggered D Flip-Flops

- Edge-sensitive devices
  - Changes occur either at positive or negative edges



(a) Circuit diagram



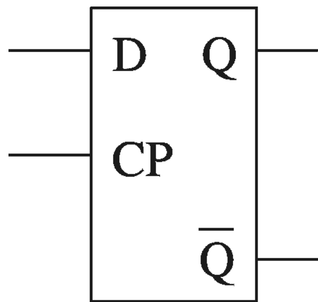
(b) Logic symbol

Notation for Latches & Flip-Flops

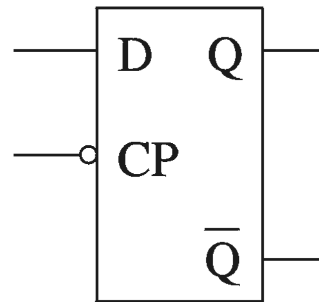
- Not strictly followed in the literature

Latches

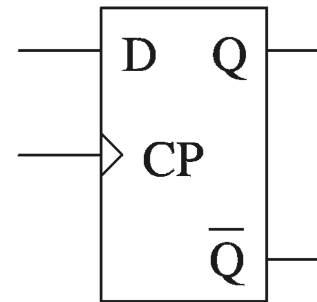
Flip-flops



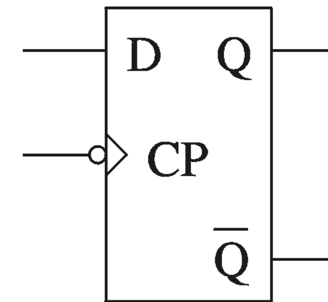
(a)
Low level



(b)
High level



(c)
Positive edge

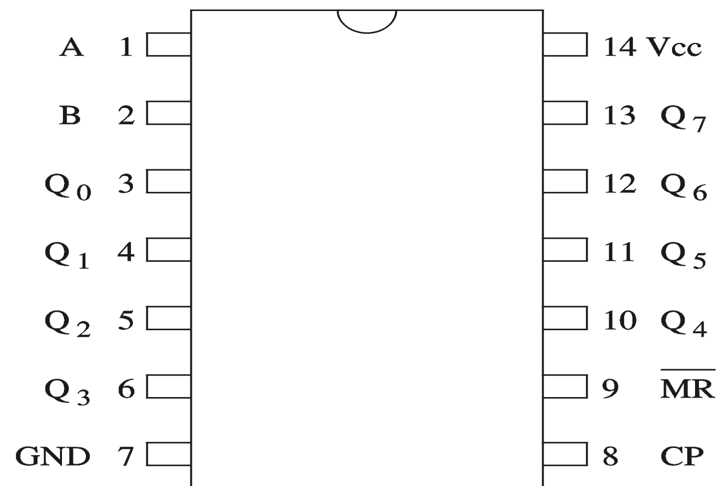


(d)
Negative edge

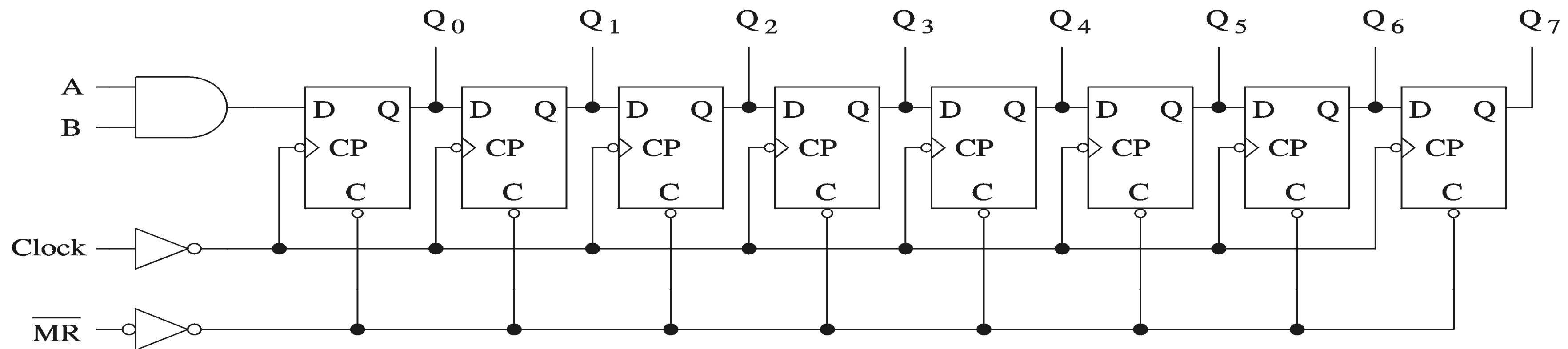
Example of Shift Register Using D Flip-Flops

74164 shift

Register chip



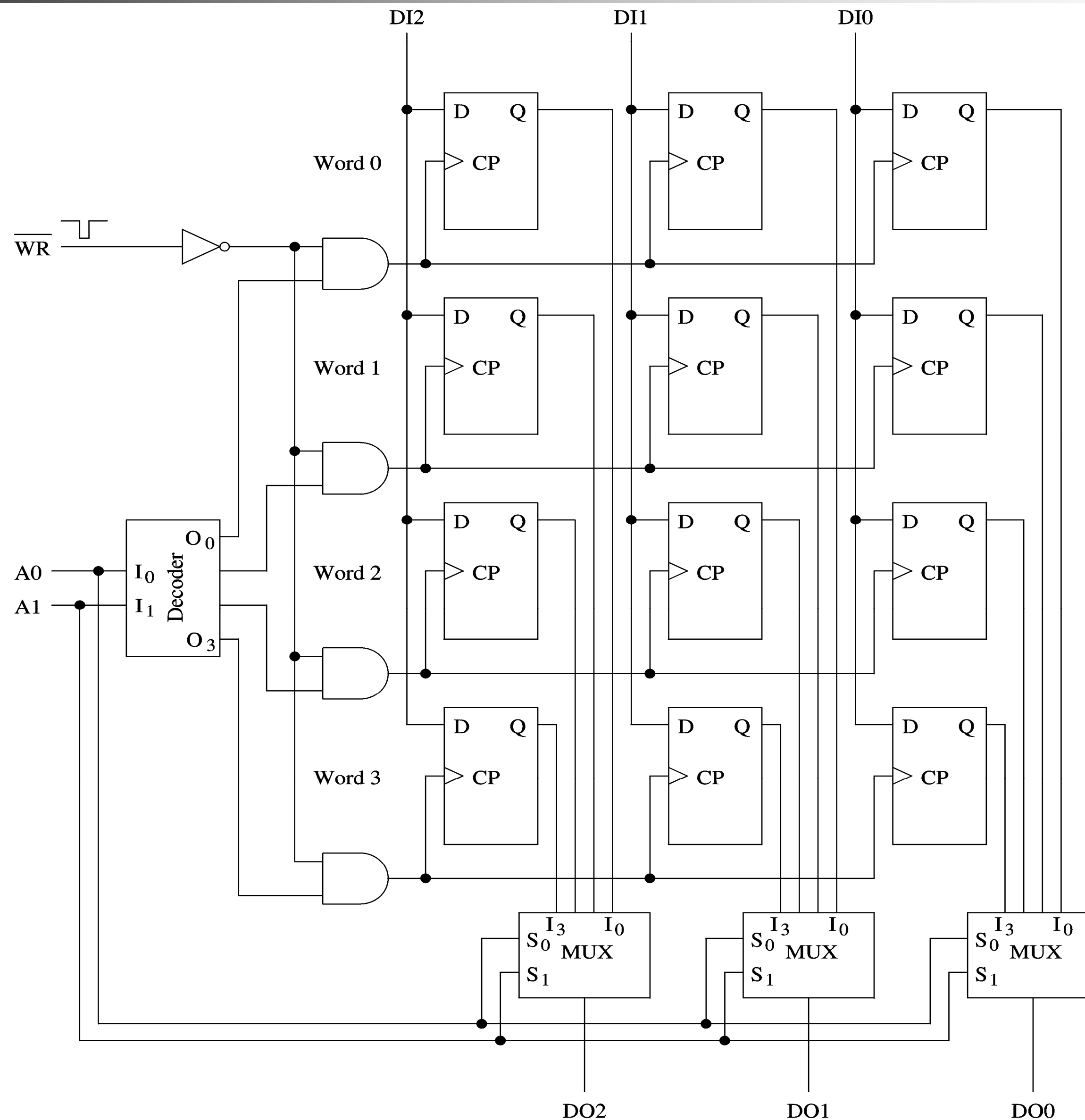
(a) Connection diagram



(b) Logic diagram

Memory Design Using D Flip-Flops

Require separate data in and out lines

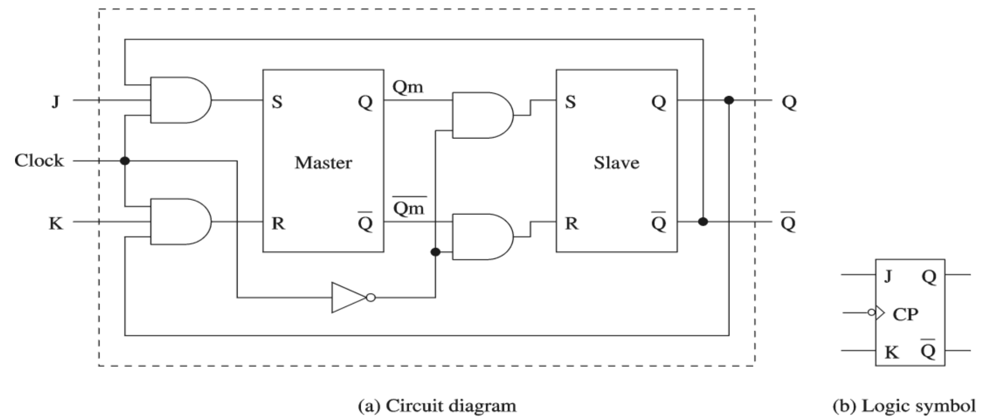


JK Flip-Flops

JK flip-flop

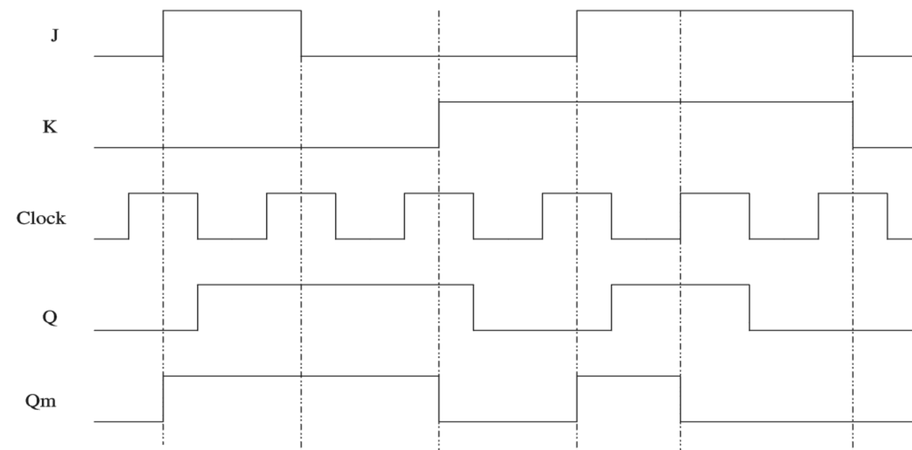
(master-slave)

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\overline{Q_n}$



(a) Circuit diagram

(b) Logic symbol



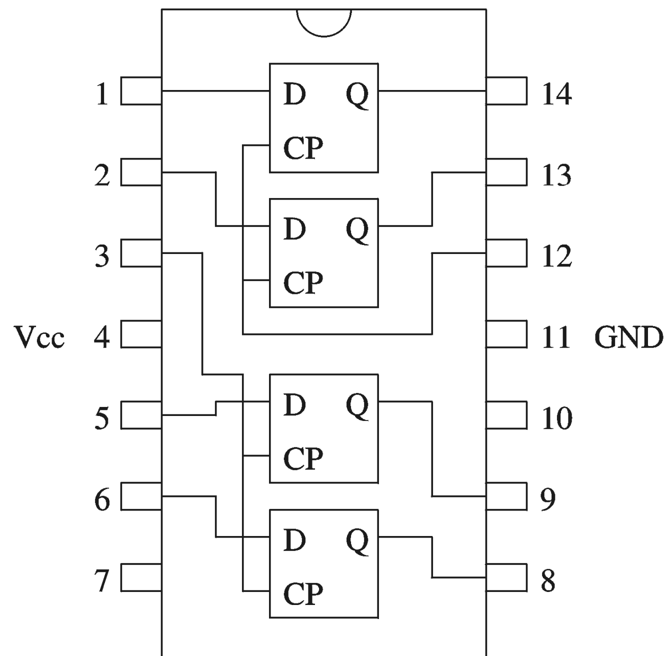
(c) Timing diagram

Examples of D & JK Flip-Flops

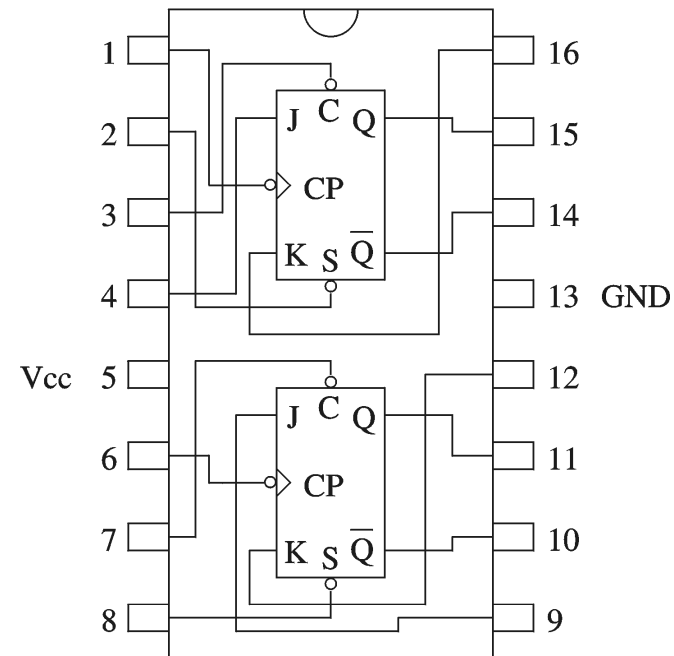
Two example chips

D latches

JK flip-flops



(a) 7477

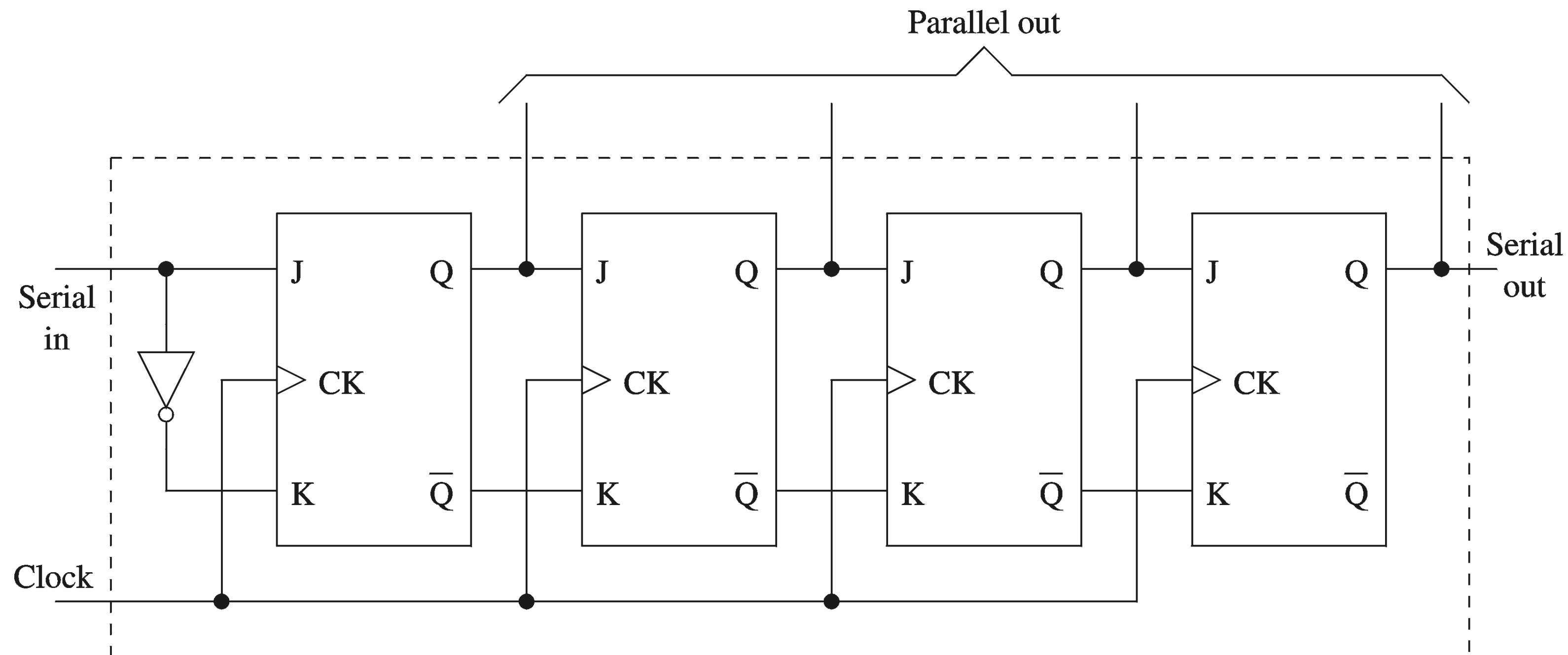


(b) 7476

Example of Shift Register Using JK Flip-Flops

- Shift Registers
  - Can shift data left or right with each clock pulse

A 4-bit shift register using JK flip-flops

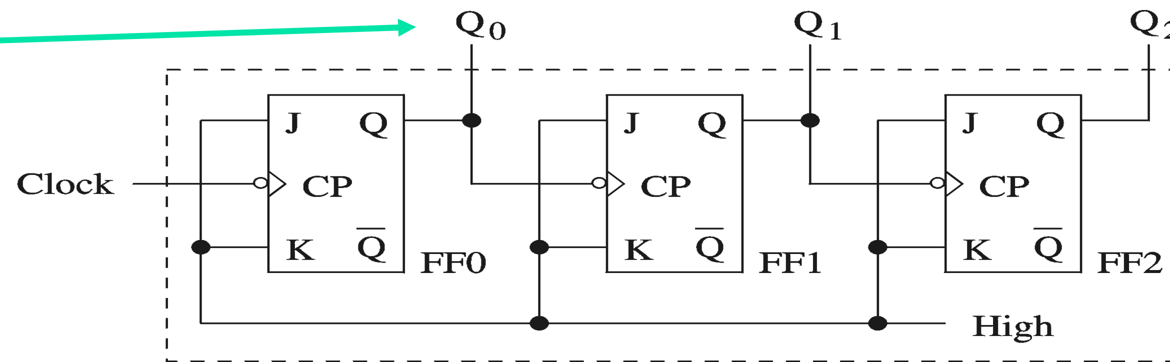


Example of Counter Using JK Flip-Flops

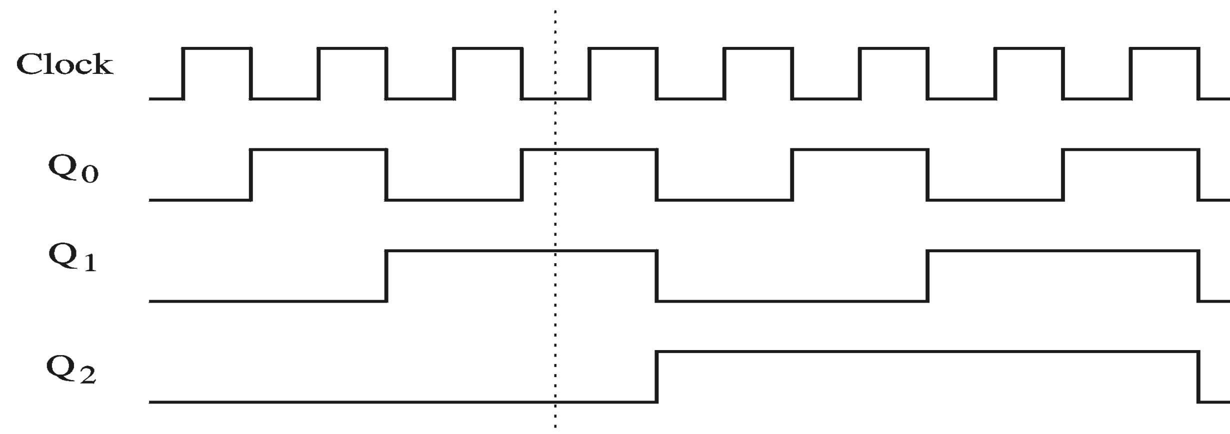
- Counters
  - Easy to build using JK flip-flops
    - Use the JK = 11 to toggle
  - Binary counters
    - Simple design
      - B bits can count from 0 to 2^B-1
    - Ripple counter
      - Increased delay as in ripple-carry adders
      - Delay proportional to the number of bits
    - Synchronous counters
      - Output changes more or less simultaneously
      - Additional cost/complexity

Modulo-8 Binary Ripple Counter Using JK Flip-Flops

LSB



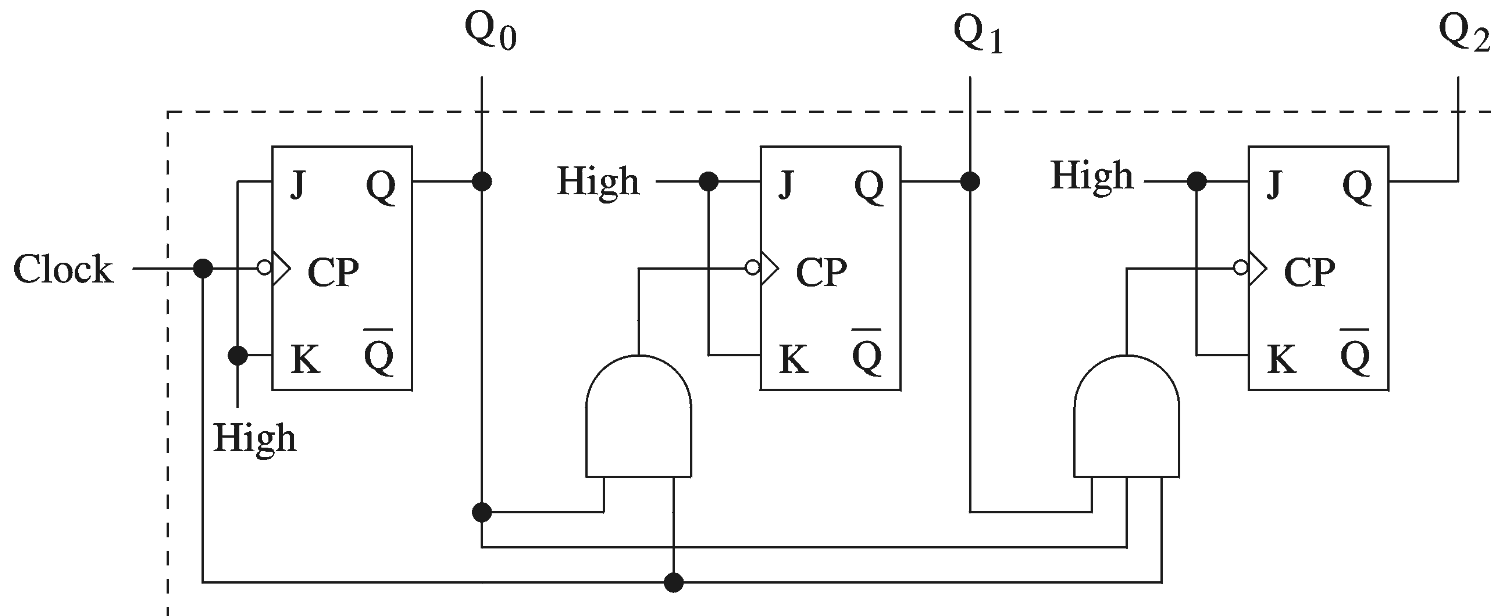
(a) Circuit diagram



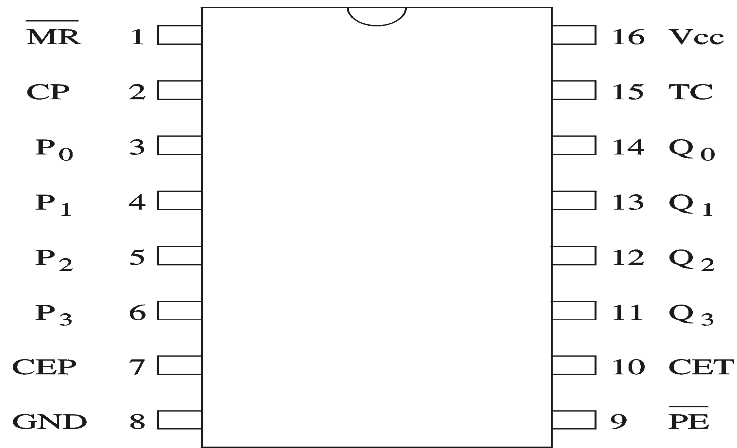
(b) Timing diagram

Synchronous Modulo-8 Counter

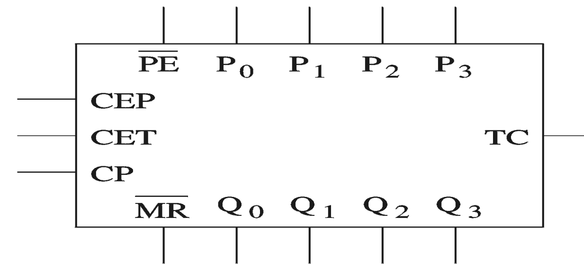
- Designed using the following simple rule
  - Change output if the preceding count bits are 1
    - Q1 changes whenever Q0 = 1
    - Q2 changes whenever Q1Q0 = 11



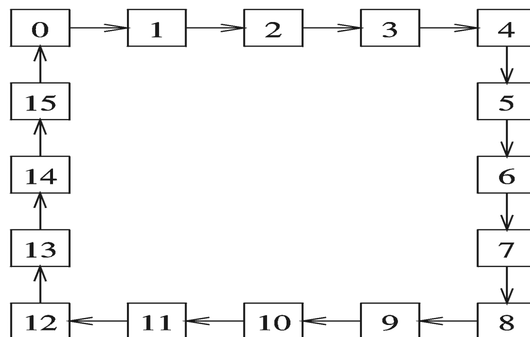
Example Counters



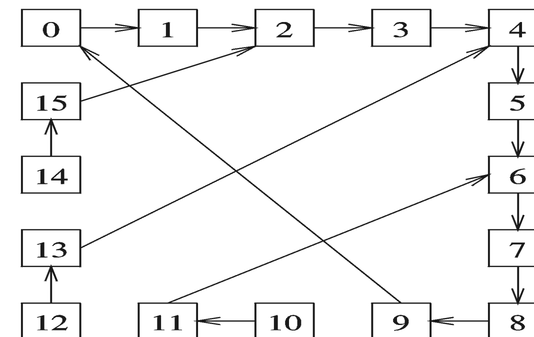
(a) Connection diagram



(b) Logic symbol



(c) State diagram of 74161



(d) State diagram of 74160

Sequential Circuit Design

- Sequential circuit consists of
  - A combinational circuit that produces output
  - A feedback circuit
    - We use JK flip-flops for the feedback circuit
- Simple counter examples using JK flip-flops
  - Provides alternative counter designs
  - We know the output
    - Need to know the input combination that produces this output
    - Use an excitation table
      - Built from the truth table

Sequential Circuit Design (cont.)

(a) JK flip-flop truth table

J	K	Q_n	Q_{n+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

(b) Excitation table for JK flip-flops

Q_n	Q_{n+1}	J	K
0	0	0	d
0	1	1	d
1	0	d	1
1	1	d	0

Sequential Circuit Design (cont.)

- Build a design table that consists of
  - Current state output
  - Next state output
  - JK inputs for each flip-flop
- Binary counter example
  - 3-bit binary counter
  - 3 JK flip-flops are needed
  - Current state and next state outputs are 3 bits each
  - 3 pairs of JK inputs

Sequential Circuit Design (cont.)

Design table for the binary counter example

Present state			Next state			JK flip-flop inputs					
A	B	C	A	B	C	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	0	1	0	d	0	d	1	d
0	0	1	0	1	0	0	d	1	d	d	1
0	1	0	0	1	1	0	d	d	0	1	d
0	1	1	1	0	0	1	d	d	1	d	1
1	0	0	1	0	1	d	0	0	d	1	d
1	0	1	1	1	0	d	0	1	d	d	1
1	1	0	1	1	1	d	0	d	0	1	d
1	1	1	0	0	0	d	1	d	1	d	1

Sequential Circuit Design (cont.)

Use K-maps to simplify expressions for JK inputs

A \ BC	00	01	11	10
0	0	0	1	0
1	d	d	d	d

$J_A = B\,C$

A \ BC	00	01	11	10
0	d	d	d	d
1	0	0	1	0

$K_A = B\,C$

A \ BC	00	01	11	10
0	0	1	d	d
1	0	1	d	d

$J_B = C$

A \ BC	00	01	11	10
0	d	d	1	0
1	d	d	1	0

$K_B = C$

A \ BC	00	01	11	10
0	1	d	d	1
1	1	d	d	1

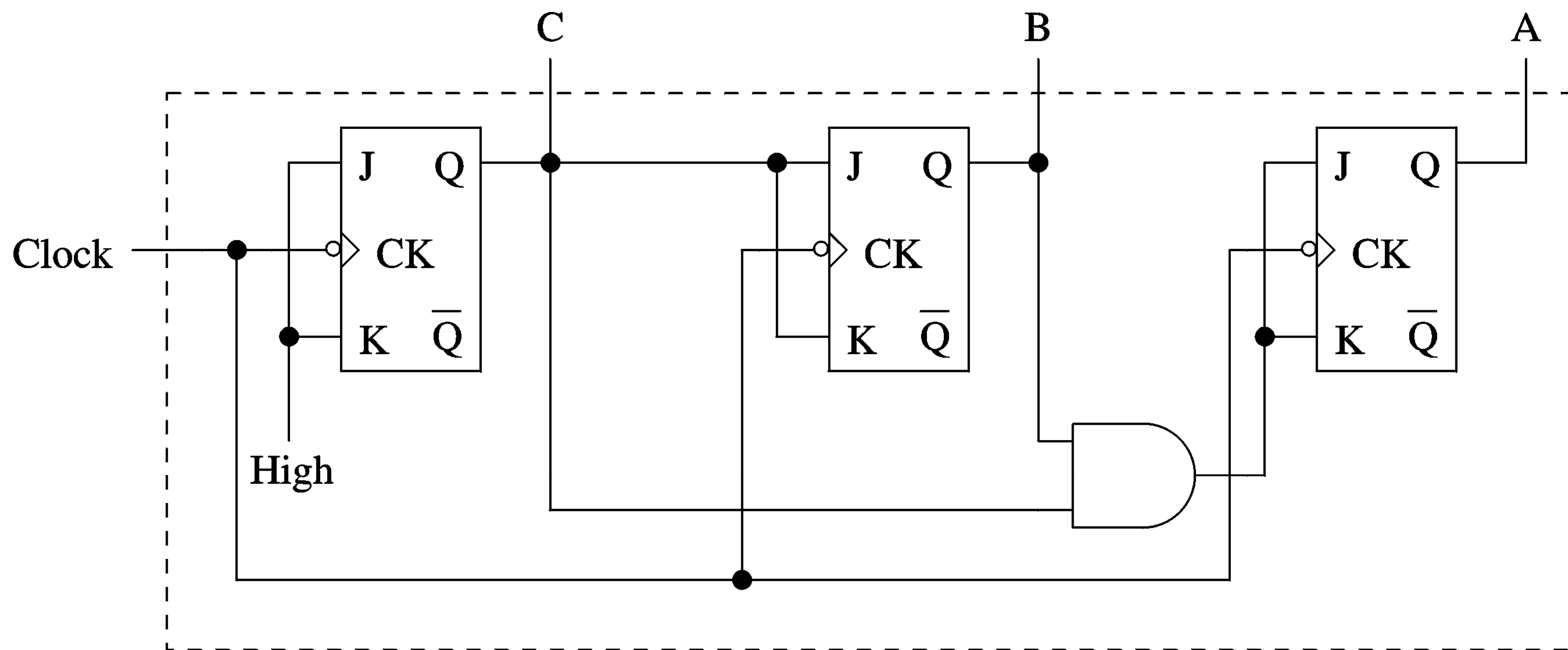
$J_C = 1$

A \ BC	00	01	11	10
0	d	1	1	d
1	d	1	1	d

$K_C = 1$

Sequential Circuit Design (cont.)

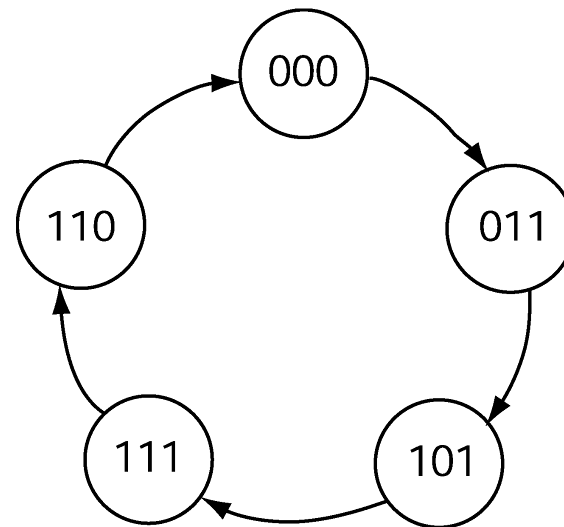
- Final circuit for the binary counter example
  - Compare this design with the synchronous counter design



Sequential Circuit Design (cont.)

- A more general counter design
  - Does not step in sequence

$0 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 6 \rightarrow 0$



- Same design process
- One significant change
  - Missing states
    - 1, 2, and 4
    - Use don’t cares for these states

Sequential Circuit Design (cont.)

Present state			Next state			JK flip-flop inputs					
A	B	C	A	B	C	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	1	1	0	d	1	d	1	d
0	0	1	–	–	–	d	d	d	d	d	d
0	1	0	–	–	–	d	d	d	d	d	d
0	1	1	1	0	1	1	d	d	1	d	0
1	0	0	–	–	–	d	d	d	d	d	d
1	0	1	1	1	1	d	0	1	d	d	0
1	1	0	0	0	0	d	1	d	1	0	d
1	1	1	1	1	0	d	0	d	0	d	1

Design table for the general counter example

Sequential Circuit Design (cont.)

K-maps to simplify JK input expressions

A \ BC	00	01	11	10
0	0	d	1	d
1	d	d	d	d

$J_A = B$

A \ BC	00	01	11	10
0	d	d	d	d
1	d	0	0	1

$K_A = \overline{C}$

A \ BC	00	01	11	10
0	1	d	d	d
1	d	1	d	d

$J_B = 1$

A \ BC	00	01	11	10
0	d	d	1	d
1	d	d	0	1

$K_B = \overline{A} + \overline{C}$

A \ BC	00	01	11	10
0	1	d	d	d
1	d	d	d	0

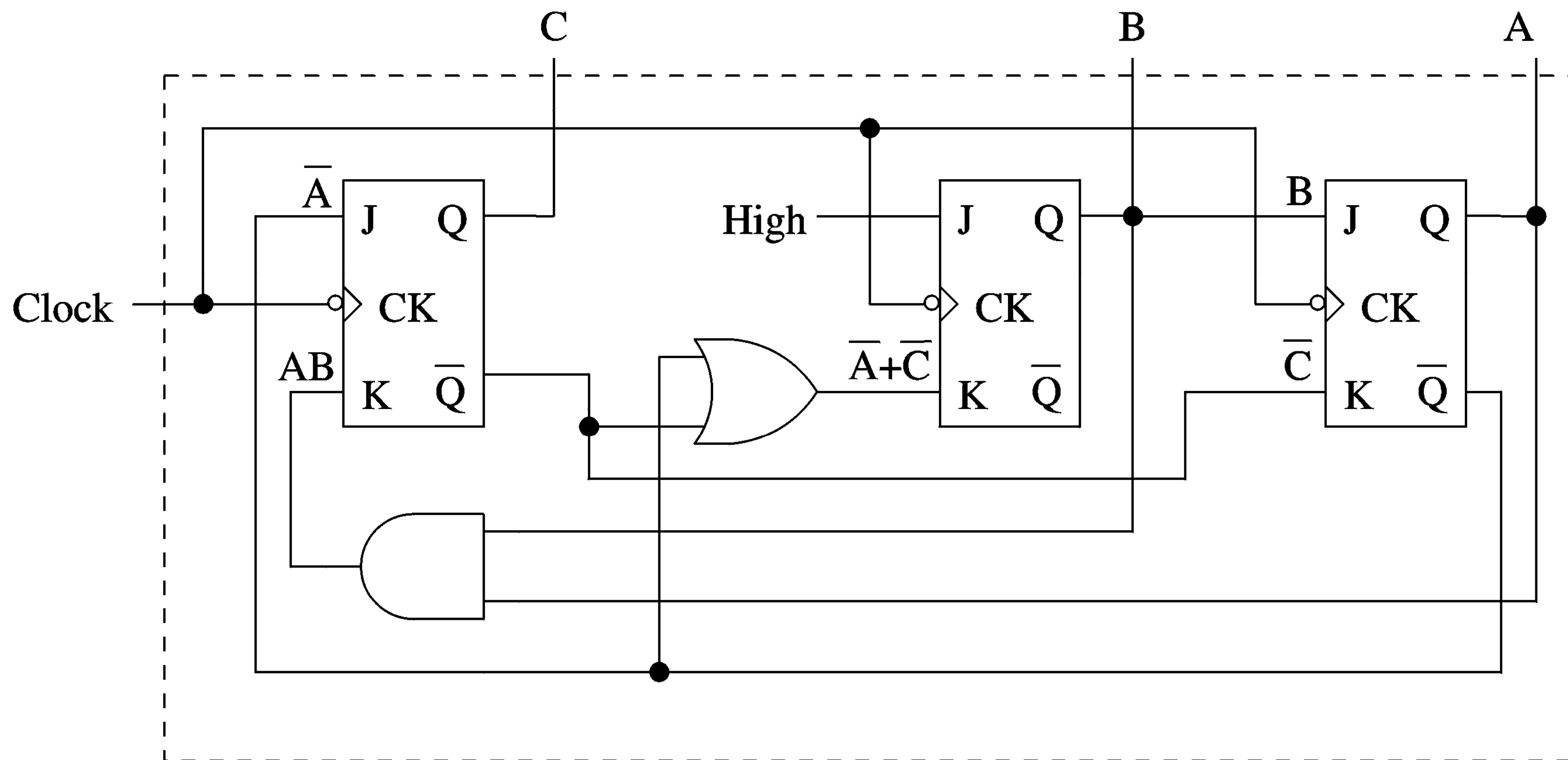
$J_C = \overline{A}$

A \ BC	00	01	11	10
0	d	d	0	d
1	d	0	1	d

$K_C = A\,B$

Sequential Circuit Design (cont.)

Final circuit for the general counter example

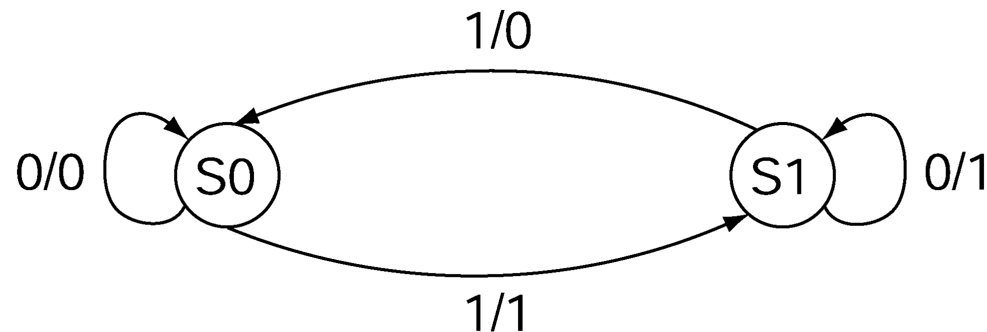


General Design Process

- FSM can be used to express the behavior of a sequential circuit
    - Counters are a special case
  - State transitions are indicated by arrows with labels X/Y
    - X: inputs that cause system state change
    - Y: output generated while moving to the next state
- Look at two examples
  - Even-parity checker
  - Pattern recognition

General Design Process (cont.)

- Even-parity checker
  - FSM needs to remember one of two facts
    - Number of 1's is odd or even
    - Need only two states
      - 0 input does not change the state
      - 1 input changes state
  - Simple example
    - Complete the design as an exercise

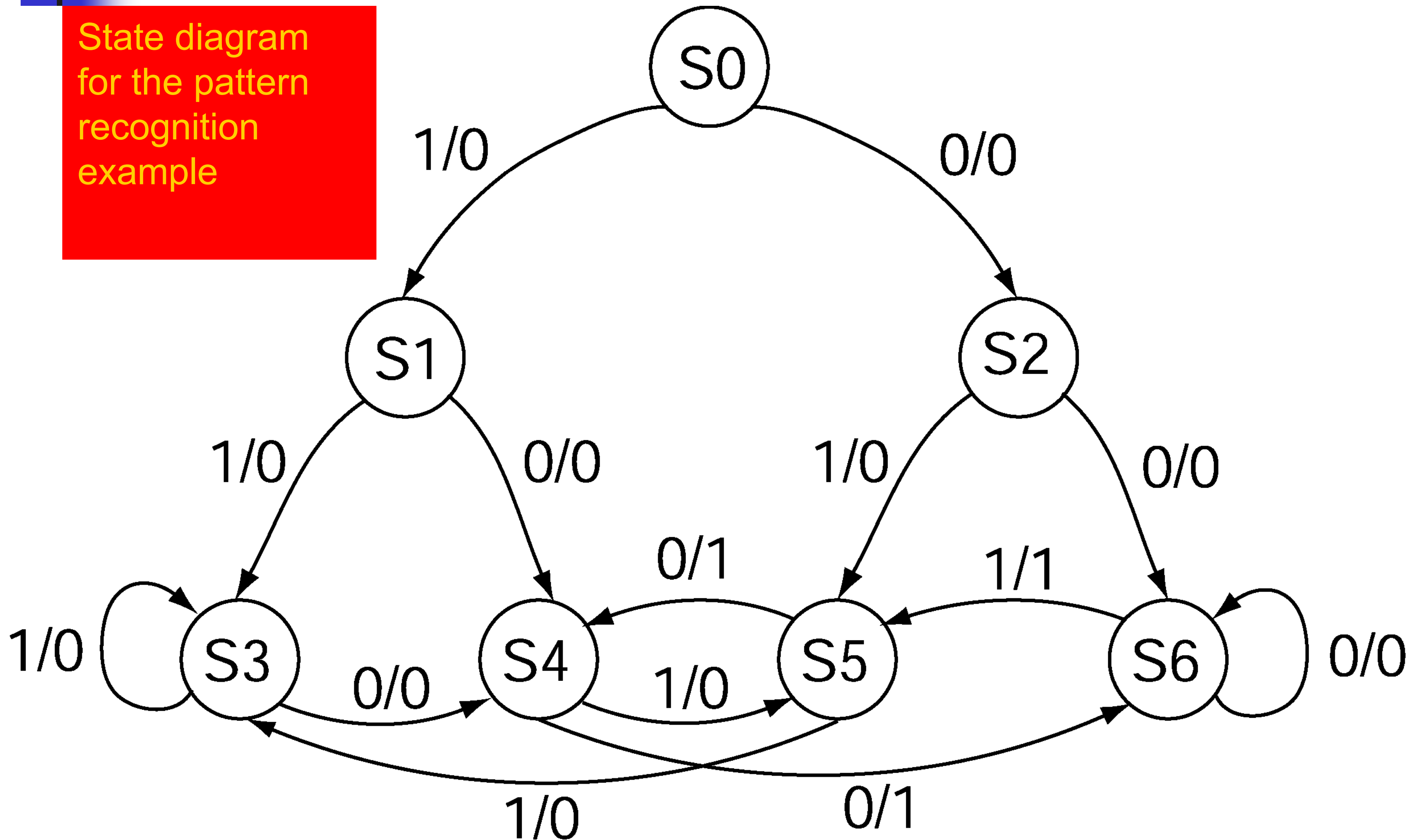


General Design Process (cont.)

- Pattern recognition example
  - Outputs 1 whenever the input bit sequence has exactly two 0s in the last three input bits
  - FSM requires thee special states to during the initial phase
    - S0 – S2
  - After that we need four states
    - S3: last two bits are 11
    - S4: last two bits are 01
    - S5: last two bits are 10
    - S6: last two bits are 00

General Design Process (cont.)

State diagram for the pattern recognition example



General Design Process (cont.)

- Steps in the design process
  1. Derive FSM
  2. State assignment
     - Assign flip-flop states to the FSM states
       - Necessary to get an efficient design
  3. Design table derivation
     - Derive a design table corresponding to the assignment in the last step
  4. Logical expression derivation
     - Use K-maps as in our previous examples
  5. Implementation

General Design Process (cont.)

- State assignment
  - Three heuristics
    - Assign adjacent states for
      - states that have the same next state
      - states that are the next states of the same state
      - States that have the same output for a given input
  - For our example
    - Heuristic 1 groupings: (S1, S3, S5)^2 (S2, S4, S6)^2
    - Heuristic 2 groupings: (S1, S2) (S3, S4)^3 (S5, S6)^3
    - Heuristic 1 groupings: (S4, S5)

General Design Process (cont.)

State table for the pattern recognition example

Present state	Next state		Output	
	X = 0	X = 1	X = 0	X = 1
S0	S2	S1	0	0
S1	S4	S3	0	0
S2	S6	S5	0	0
S3	S4	S3	0	0
S4	S6	S5	1	0
S5	S4	S3	1	0
S6	S6	S5	0	1

General Design Process (cont.)

K-map for state assignment

A \ BC	00	01	11	10
0	S0	S3	S5	S1
1		S4	S6	S2

State assignment

State		A	B	C
S0	=	0	0	0
S1	=	0	1	0
S2	=	1	1	0
S3	=	0	0	1
S4	=	1	0	1
S5	=	0	1	1
S6	=	1	1	1

General Design Process (cont.)

Design table

Present state			Present state	Next state			Present state	JK flip-flop inputs					
A	B	C	X	A	B	C	Y	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	1	1	0	0	1	d	1	d	0	d
0	0	0	1	0	1	0	0	0	d	1	d	0	d
0	0	1	0	1	0	1	0	1	d	0	d	d	0
0	0	1	1	0	0	1	0	0	d	0	d	d	0
0	1	0	0	1	0	1	0	1	d	d	1	1	d
0	1	0	1	0	0	1	0	0	d	d	1	1	d
0	1	1	0	1	0	1	1	1	d	d	1	d	0
0	1	1	1	0	0	1	0	0	d	d	1	d	0
1	0	1	0	1	1	1	1	d	0	1	d	d	0
1	0	1	1	0	1	1	0	d	1	1	d	d	0
1	1	0	0	1	1	1	0	d	0	d	0	1	d
1	1	0	1	0	1	1	0	d	1	d	0	1	d
1	1	1	0	1	1	1	0	d	0	d	0	d	0
1	1	1	1	0	1	1	1	d	1	d	0	d	0

General Design Process (cont.)

K-maps for JK inputs

AB \ CX	00	01	11	10
00	1	0	0	1
01	1	0	0	1
11	d	d	d	d
10	d	d	d	d

$J_A = \overline{X}$

AB \ CX	00	01	11	10
00	d	d	d	d
01	d	d	d	d
11	0	1	1	0
10	d	d	1	0

$K_A = X$

AB \ CX	00	01	11	10
00	1	1	0	0
01	d	d	d	d
11	d	d	d	d
10	d	d	1	1

$J_B = \overline{C} + A$

AB \ CX	00	01	11	10
00	d	d	d	d
01	1	1	1	1
11	0	0	0	0
10	d	d	d	d

$K_B = \overline{A}$

AB \ CX	00	01	11	10
00	0	0	d	d
01	1	1	d	d
11	1	1	d	d
10	d	d	d	d

$J_C = B$

AB \ CX	00	01	11	10
00	d	d	0	0
01	d	d	0	0
11	d	d	0	0
10	d	d	0	0

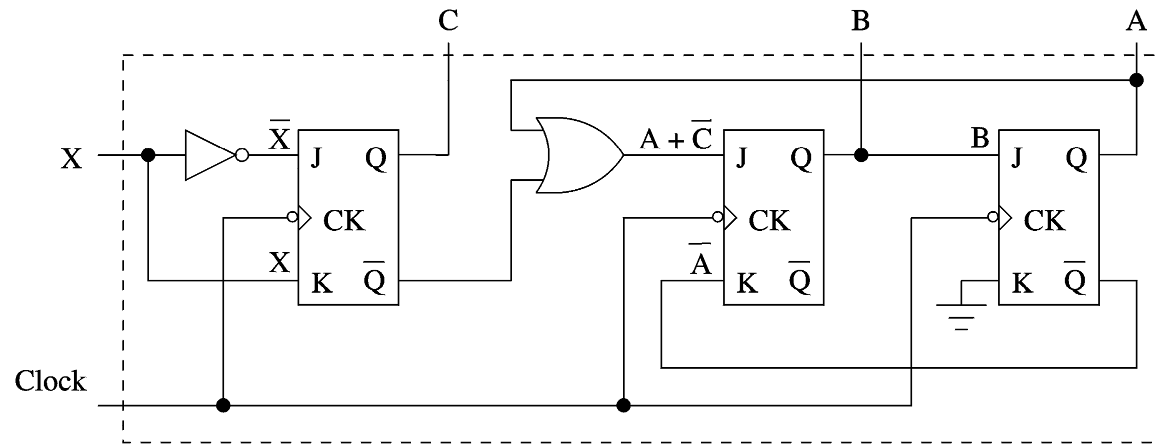
$K_C = 0$

K-map for the output

AB \ CX	00	01	11	10
00	0	0	0	0
01	0	0	0	1
11	0	0	1	0
10	d	d	0	1

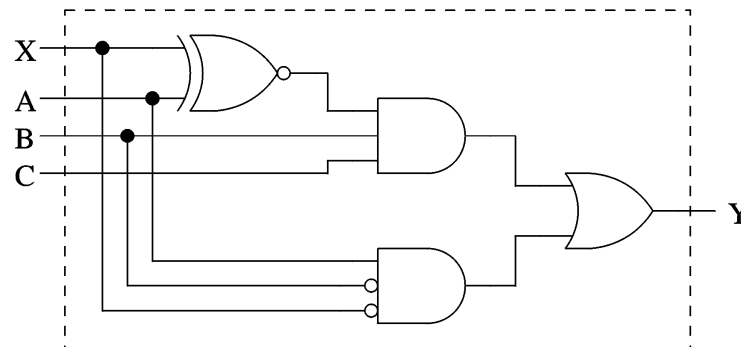
$Y = \overline{A}\,B\,C\,\overline{X} + A\,B\,C\,X + A\,\overline{B}\,\overline{X}$

General Design Process (cont.)



(a)

Final implementation



(b)