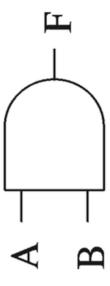


# **Basic Logic Review**

# Basic Concepts

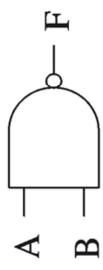
- Simple logic gates
  - AND  $\rightarrow$  0 if one or more inputs is 0
  - OR  $\rightarrow$  1 if one or more inputs is 1
  - NOT
- $\text{NAND} = \text{AND} + \text{NOT}$ 
  - 1 if one or more inputs is 0
- $\text{NOR} = \text{OR} + \text{NOT}$ 
  - 0 if one or more input is 1
- XOR implements exclusive-OR function
- NAND and NOR gates require fewer transistors than AND and OR in standard CMOS
- Functionality can be expressed by a truth table
  - A truth table lists output for each possible input combination

# Basic Logic Gates



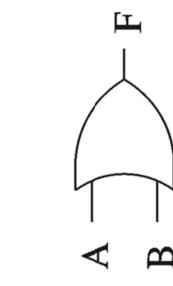
AND gate

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



NAND gate

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



OR gate

A	F
0	1
1	0



NOR gate

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

Truth table

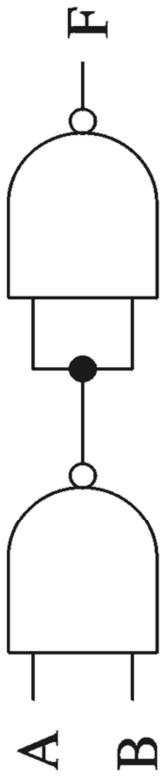
Logic symbol

# Complete Set of Gates

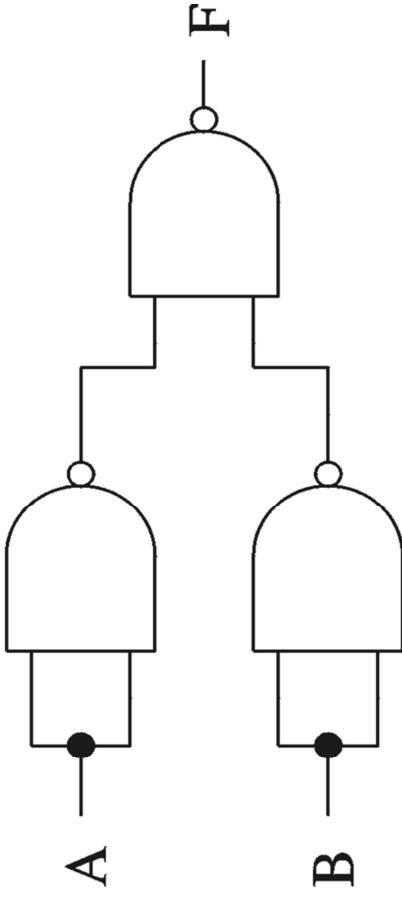
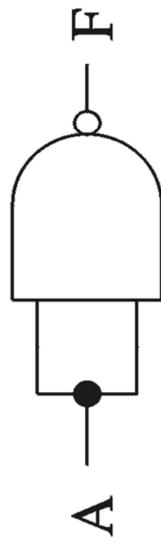
- Complete sets
  - A set of gates is complete
    - if we can implement any logical function using only the type of gates in the set
  - Some example complete sets
    - {AND, OR, NOT} → Not a minimal complete set
    - {AND, NOT}
    - {OR, NOT}
    - {NAND}
    - {NOR}
  - Minimal complete set
    - A complete set with no redundant elements.

# NAND as a Complete Set

- Proving NAND gate is universal



NOT gate



OR gate

# Logic Functions

---

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

# Logic Functions (cont'd)

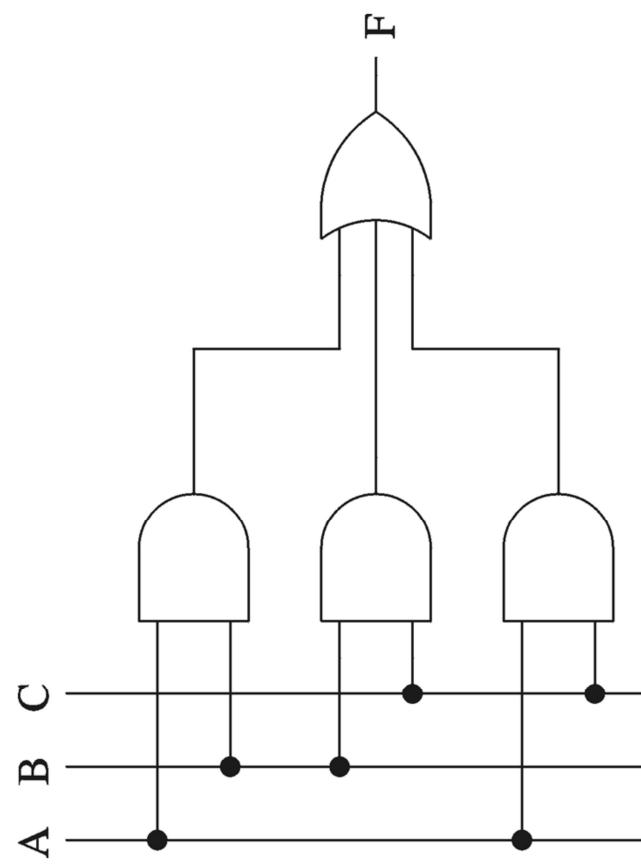
## Truth table

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

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

## Graphical schematic form



# Boolean Algebra

Name	AND version	OR version
Identity	$x \cdot 1 = x$	$x + 0 = x$
Complement	$x \cdot x' = 0$	$x + x' = 1$
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'd)

- Boolean identities (cont'd)

Name      AND version      OR version

---

Involution	$x = (x')$	---
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	$(x \cdot y)' = x' + y'$	$(x + y)' = x' \cdot y'$
(de Morgan's law in particular is <i>very useful</i> )		

# Majority Function Using Other Gates

---

- Using NAND gates
  - Get an equivalent expression
$$A \cdot B + C \cdot D = (A \cdot B + C \cdot D)''$$
  - Using de Morgan's law

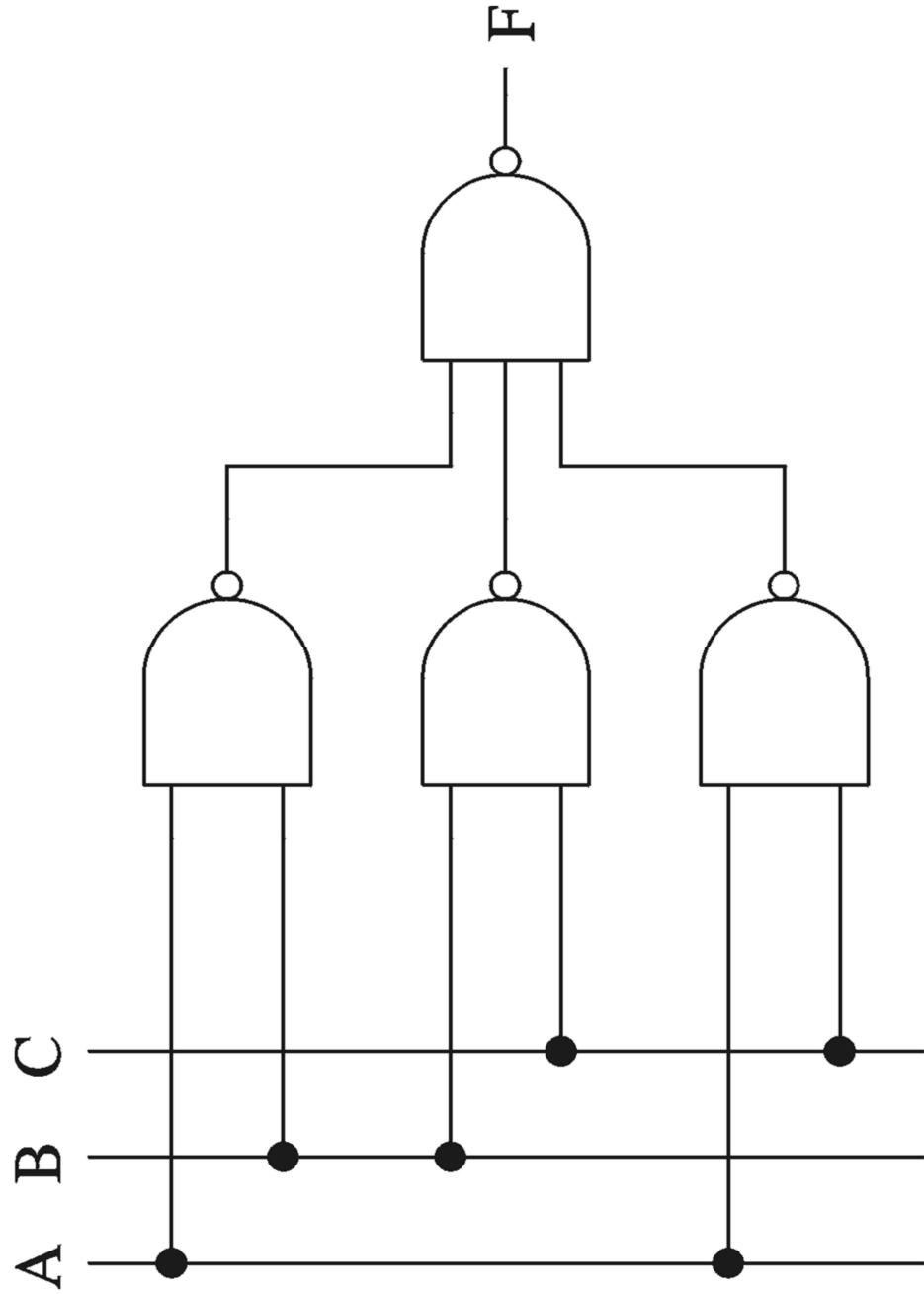
$$A \cdot B + C \cdot D = ((A \cdot B)' \cdot (C \cdot D))'$$

- Can be generalized
  - Example: Majority function

$$A \cdot B + B \cdot C + A \cdot C = ((A \cdot B)' \cdot (B \cdot C)' \cdot (A \cdot C)')'$$

## Majority Function Using Other Gates (cont'd)

- Majority function



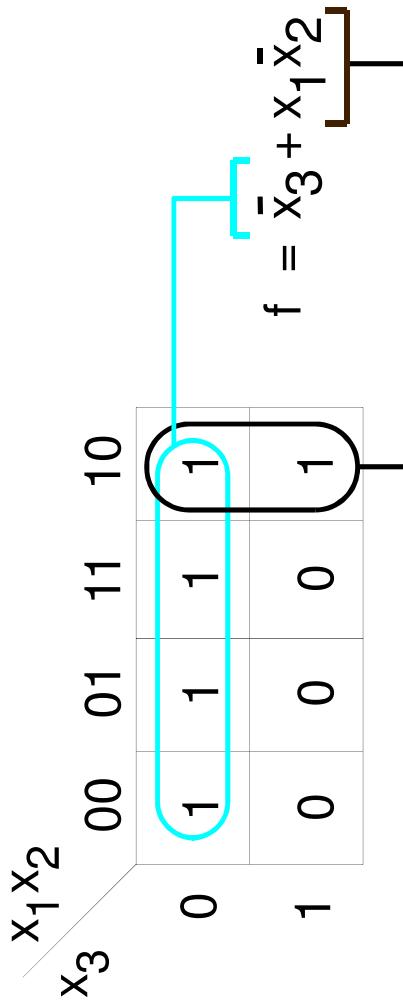
# Karnaugh Maps

$x_1$	$x_2$	$x_3$	
0	0	0	$m_0$
0	0	1	$m_1$
0	1	0	$m_2$
0	1	1	$m_3$
1	0	0	$m_4$
1	0	1	$m_5$
1	1	0	$m_6$
1	1	1	$m_7$

$x_3 \backslash x_1 x_2$

	00	01	11	10
0	$m_0$	$m_2$	$m_6$	$m_4$
1	$m_1$	$m_3$	$m_7$	$m_5$

(b) Karnaugh map



(a) Truth table

An example of three-variable Karnaugh maps

# Numbers

Decimal	Binary	Octal	Hexadecimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12

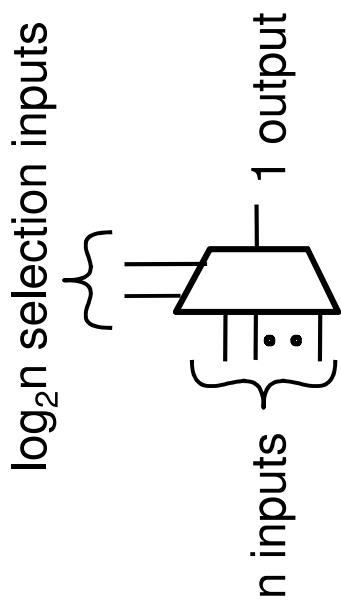
**Table 5.1** Interpretation of four-bit signed integers.

	$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
	0111	+7	+7	+7
	0110	+6	+6	+6
	0101	+5	+5	+5
	0100	+4	+4	+4
	0011	+3	+3	+3
	0010	+2	+2	+2
	0001	+1	+1	+1
	0000	+0	+0	+0
	1000	-0	-7	-8
	1001	-1	-6	-7
	1010	-2	-5	-6
	1011	-3	-4	-5
	1100	-4	-3	-4
	1101	-5	-2	-3
	1110	-6	-1	-2
	1111	-7	-0	-1

Numbers in different systems

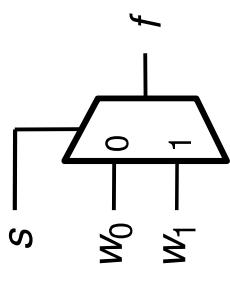
# Combinational Logic Building Blocks

# Multiplexers



- multiplexer
  - n binary inputs (binary input = 1-bit input)
  - log<sub>2</sub>n binary selection inputs
  - 1 binary output
- Function: One of n inputs is placed onto output
- Called **n-to-1** multiplexer

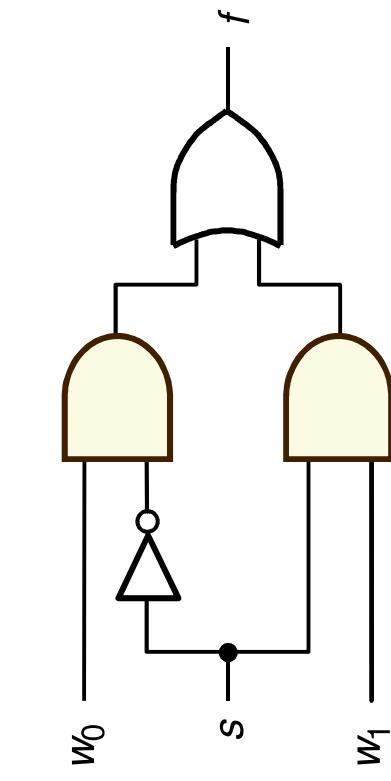
# 2-to-1 Multiplexer



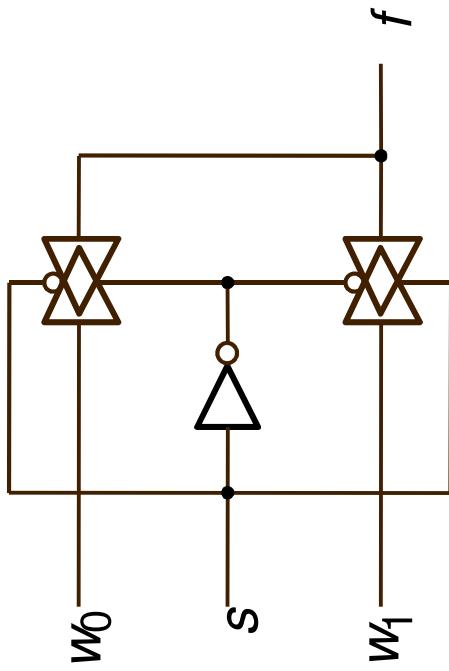
$S$	$f$
0	$w_0$
1	$w_1$

(a) Graphical symbol

(b) Truth table

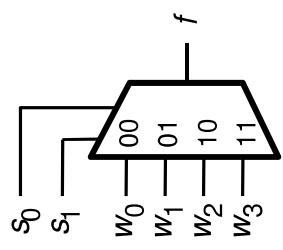


(c) Sum-of-products circuit



(d) Circuit with transmission gates

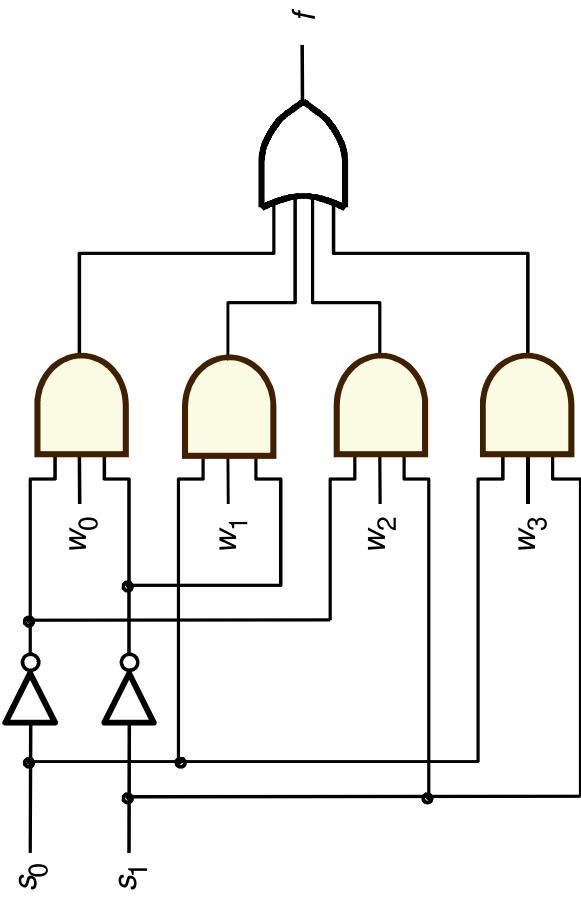
# 4-to-1 Multiplexer



$s_1$	$s_0$	$f$
0	0	$w_0$
0	1	$w_1$
1	0	$w_2$
1	1	$w_3$

(a) Graphic symbol

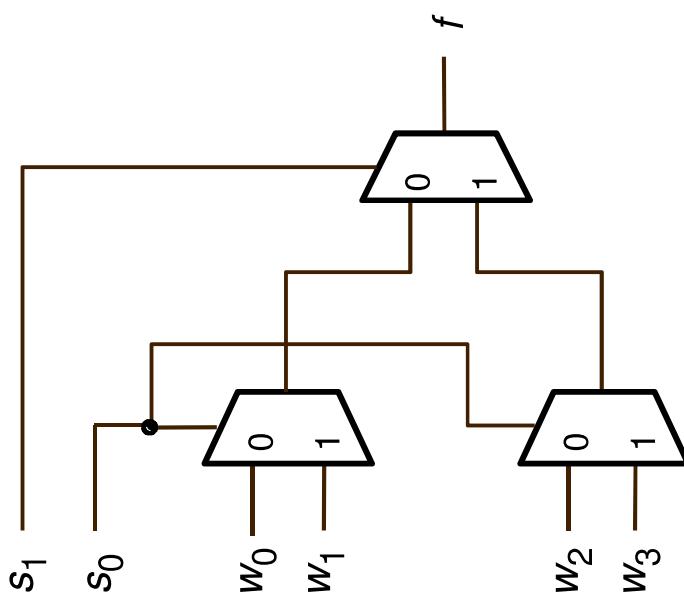
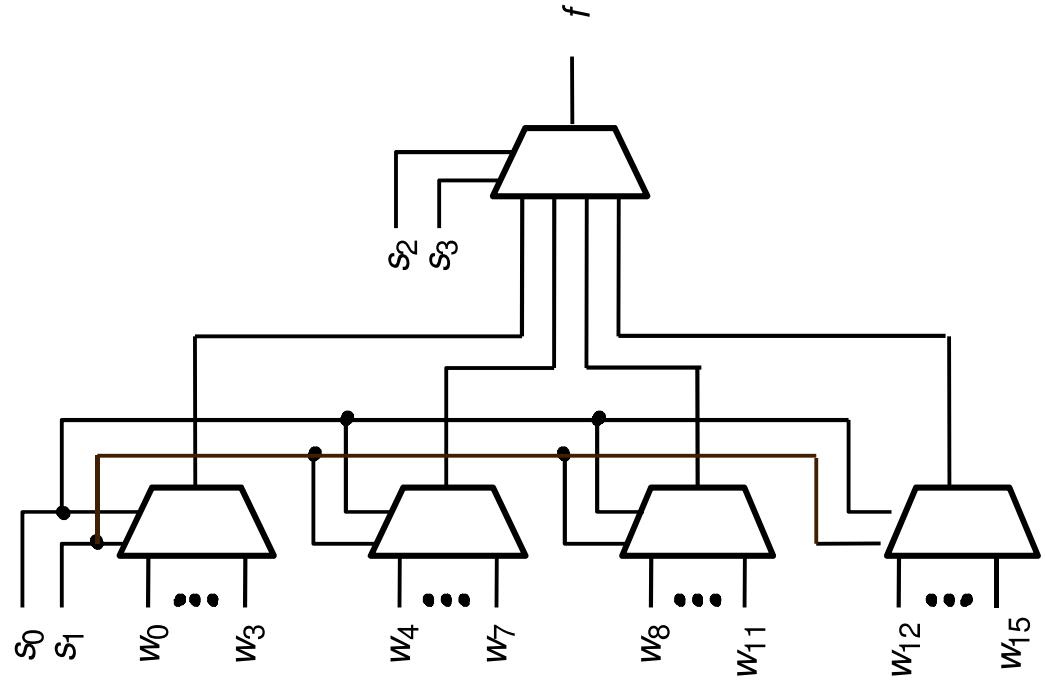
(b) Truth table



Source: Brown and Vranesic

(c) Circuit

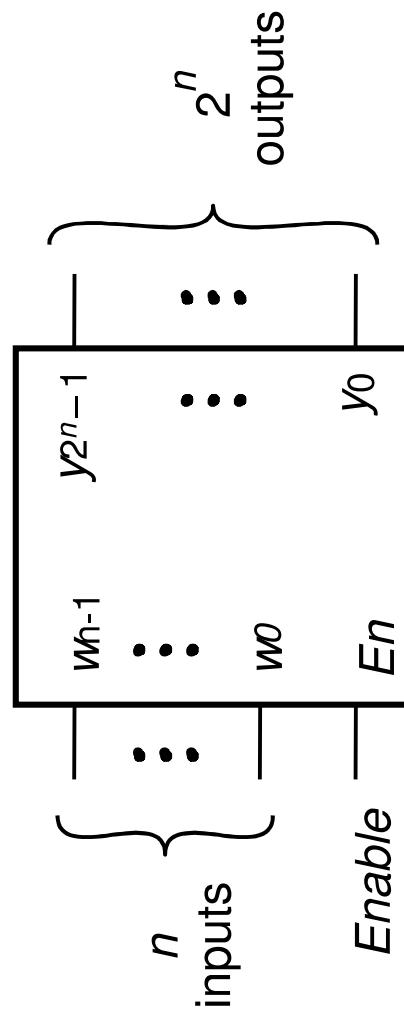
# Multiplexer



Using 2-to-1 multiplexers to  
build a 4-to-1 multiplexer.

A 16-to-1 multiplexer.

# Decoders



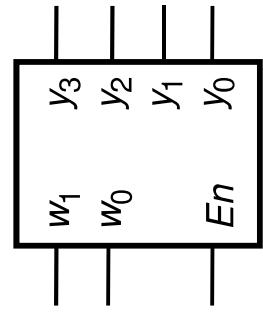
- Decoder

- $n$  binary inputs
- $2^n$  binary outputs
- Function: decode encoded information
  - If enable=1, one output is asserted high, the other outputs are asserted low
    - If enable=0, all outputs asserted low
  - Often, enable pin is not needed (i.e. the decoder is always enabled)
- Called **n-to- $2^n$**  decoder
  - Can consider  $n$  binary inputs as a single  $n$ -bit input
  - Can consider  $2^n$  binary outputs as a single  $2^n$ -bit output
- Decoders are often used for RAM/ROM addressing

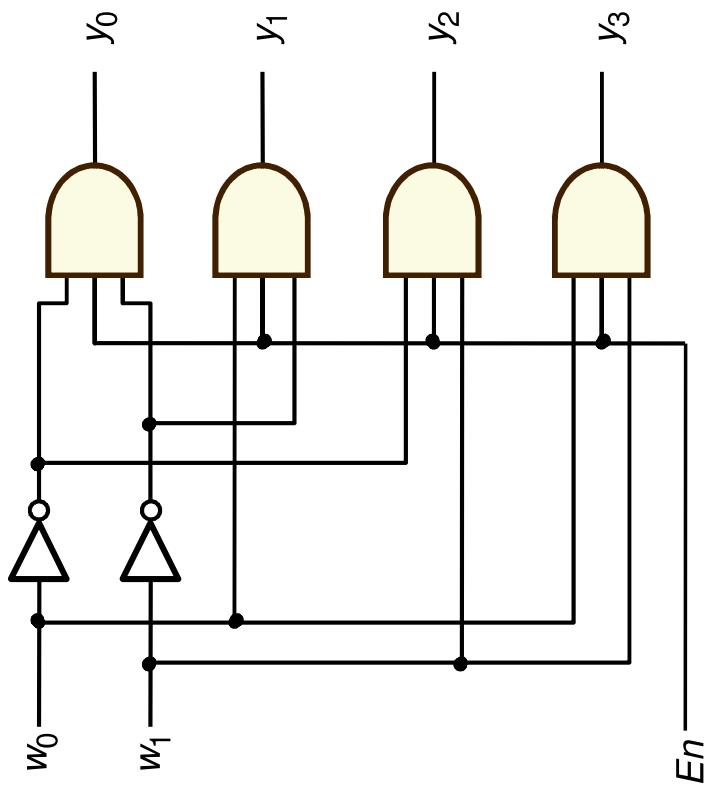
# 2-to-4 Decoder

$E_n$	$w_1$	$w_0$	$y_3$	$y_2$	$y_1$	$y_0$
1	0	0	0	0	1	
1	0	1	0	0	0	
1	1	0	0	1	0	
1	1	1	1	0	0	0
0	-	-	0	0	0	0

(a) Truth table



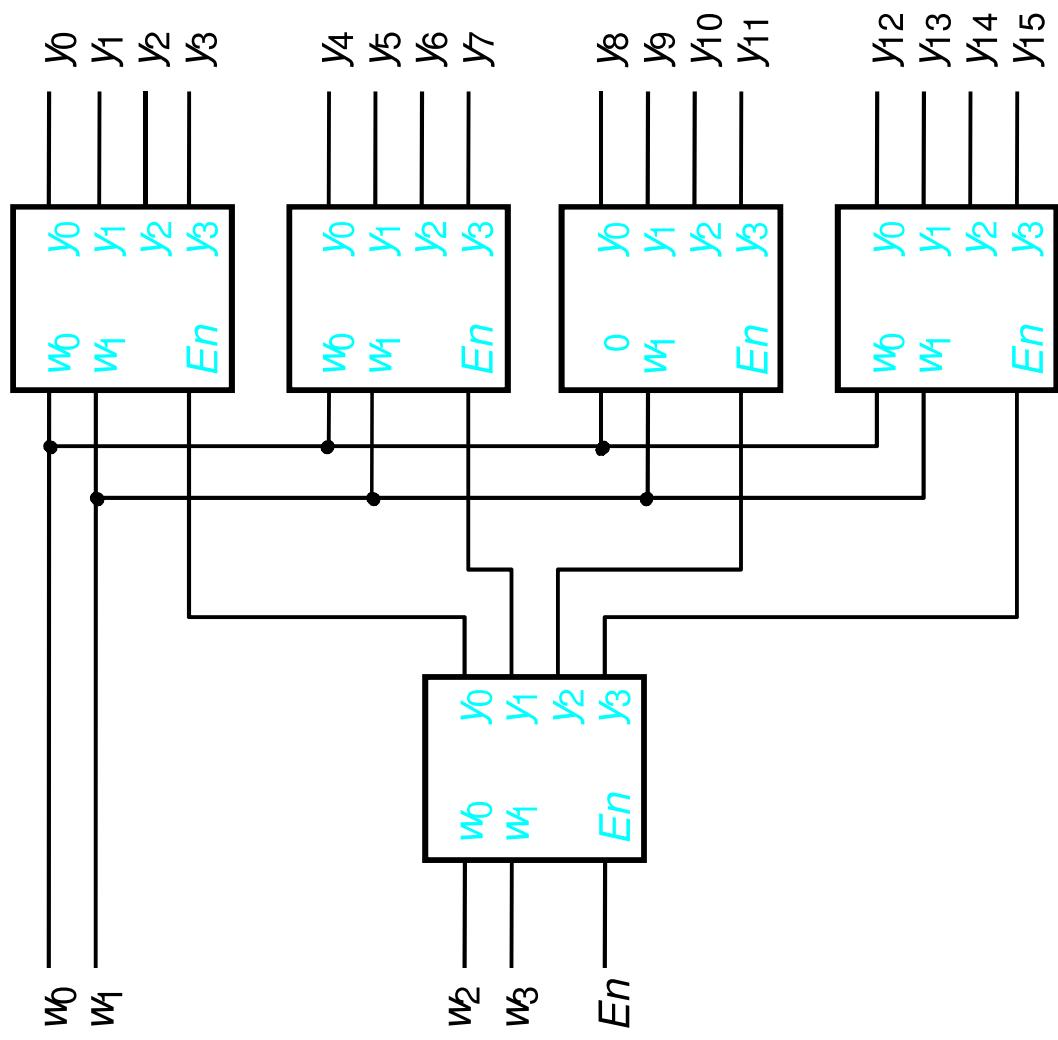
(b) Graphical symbol



Source: Brown and Vranesic

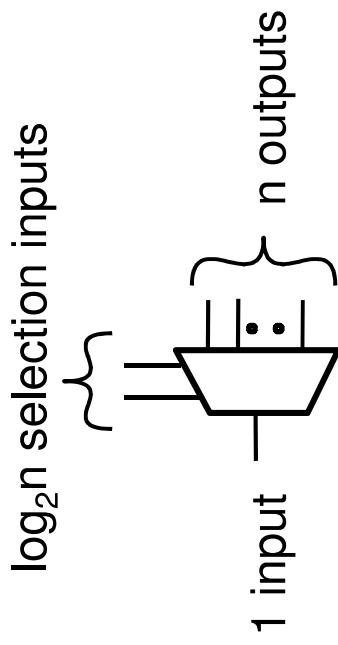
(c) Logic circuit

# Decoders



A 4-to-16 decoder built using a decoder tree

# Demultiplexers



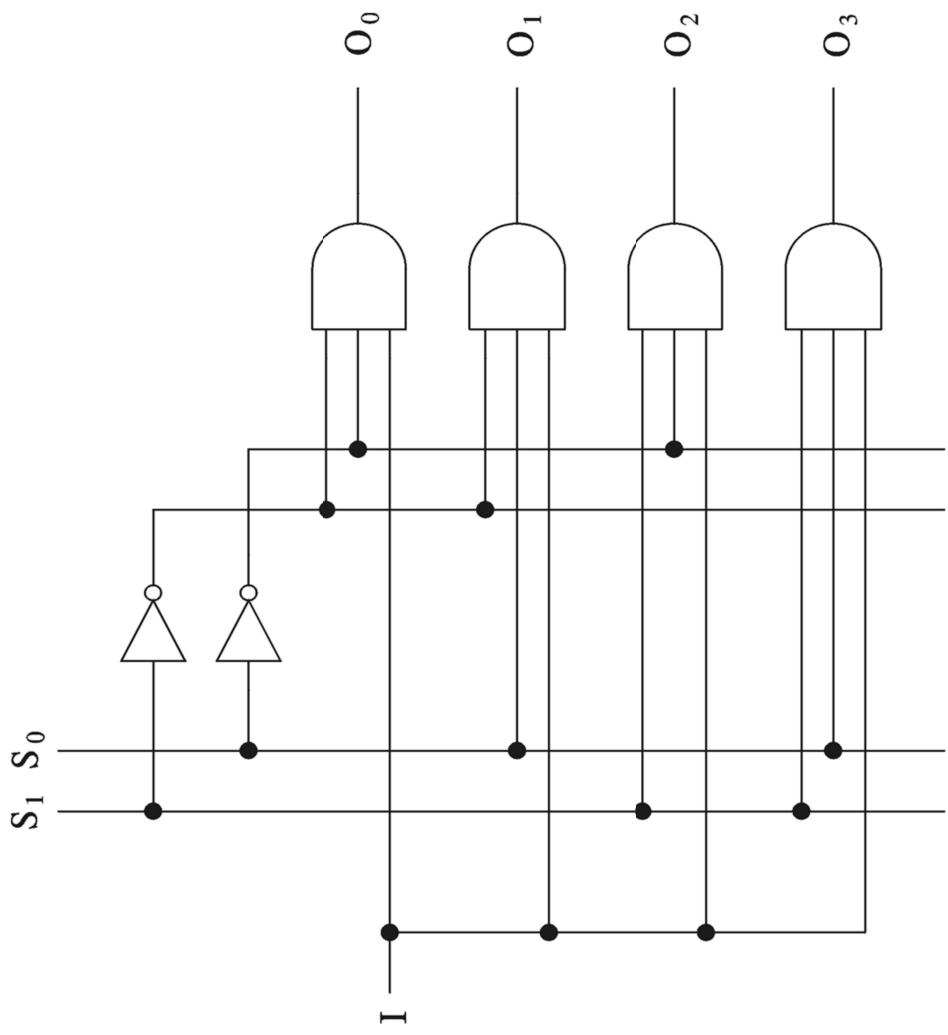
- Demultiplexer
  - 1 binary input
  - $n$  binary outputs
  - $\log_2 n$  binary selection inputs
  - Function: places input onto one of  $n$  outputs, with the remaining outputs asserted low
  - Called **1-to-n demultiplexer**
  - Closely related to decoder
    - Can build 1-to- $n$  demultiplexer from  $\log_2 n$ -to- $n$  decoder by using the decoder's enable signal as the demultiplexer's input signal, and using decoder's input signals as the demultiplexer's selection input signals.

# 1-to-4 Demultiplexer

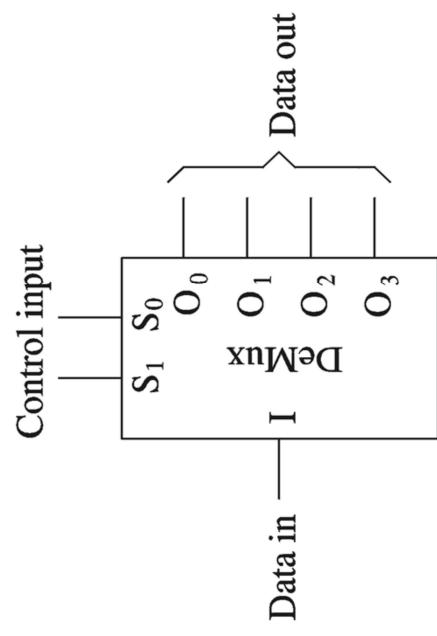
S S | Q Q Q Q

1	0	3	2	1	0
0	0	0	0	0	A
0	1	0	0	A	0
1	0	0	A	0	0
1	1	A	0	0	0
-	-	0	0	0	0

(a) Truth table

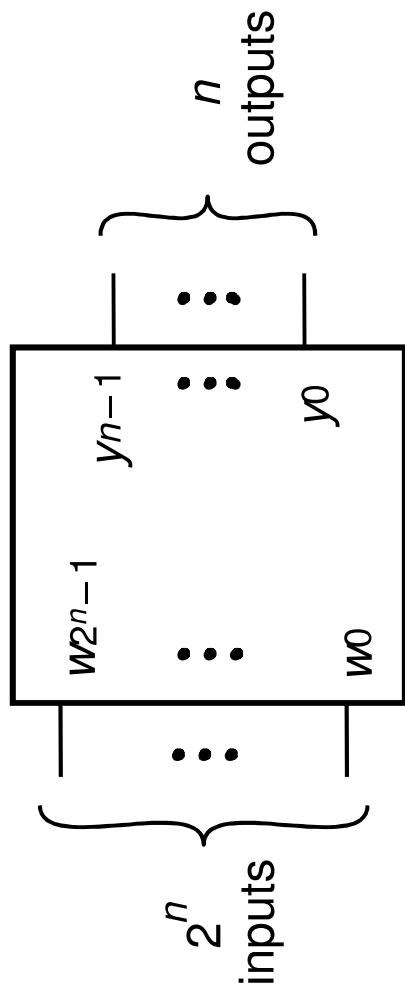


(b) Graphic symbol



(c) Circuit

# Encoders

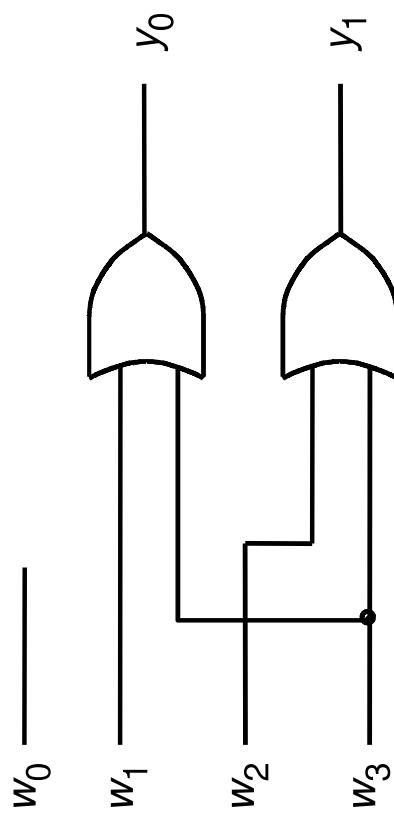


- Encoder
  - $2^n$  binary inputs
  - $n$  binary outputs
  - Function: encodes information into an  $n$ -bit code
  - Called  **$2^n$ -to- $n$**  encoder
    - Can consider  $2^n$  binary inputs as a single  $2^n$ -bit input
    - Can consider  $n$  binary output as a single  $n$ -bit output
  - Encoders only work when **exactly one** binary input is equal to 1

# 4-to-2 Encoder

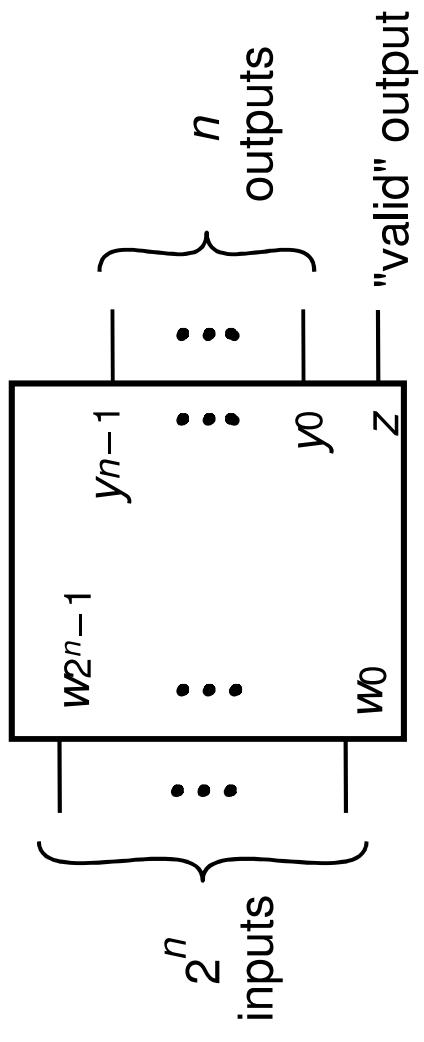
$w_3$	$w_2$	$w_1$	$w_0$	$y_1$	$y_0$
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Truth table



(b) Circuit

# Priority Encoders



- Priority Encoder
  - $2^n$  binary inputs
  - $n$  binary outputs
  - 1 binary "valid" output
  - Function: encodes information into an  $n$ -bit code based on priority of inputs
  - Called  **$2^n$ -to- $n$**  priority encoder
- Priority encoder allows for multiple inputs to have a value of '1', as it encodes the input with the highest priority (MSB = highest priority, LSB = lowest priority)
  - "valid" output indicates when priority encoder output is valid
  - Priority encoder is more common than an encoder

# 4-to-2 Priority Encoder

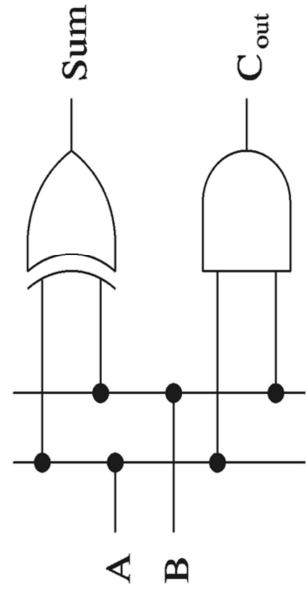
$w_3$	$w_2$	$w_1$	$w_0$	$y_1$	$y_0$	$z$
0	0	0	0	-	-	0
0	0	0	1	0	0	1
0	0	1	-	0	1	1
0	1	-	-	1	0	1
1	-	-	-	1	1	1

# Single-Bit Adders

- Half-adder
  - Adds two binary (i.e. 1-bit) inputs  $A$  and  $B$ 
    - Produces a *sum* and *carryout*
  - Problem: Cannot use it alone to build larger adders
- Full-adder
  - Adds three binary (i.e. 1-bit) inputs  $A$ ,  $B$ , and *carryin*
    - Like half-adder, produces a *sum* and *carryout*
  - Allows building  $M$ -bit adders ( $M > 1$ )
    - Simple technique
      - Connect  $C_{\text{out}}$  of one adder to  $C_{\text{in}}$  of the next
    - These are called *ripple-carry adders*
    - Shown in next section

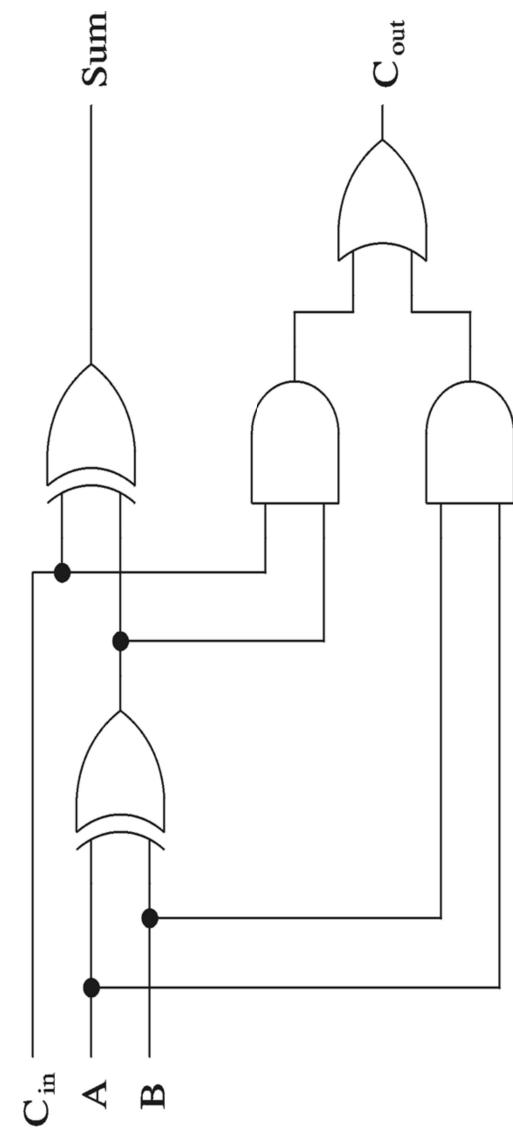
# Single-Bit Adders (cont'd)

A	B	Sum	C <sub>out</sub>
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



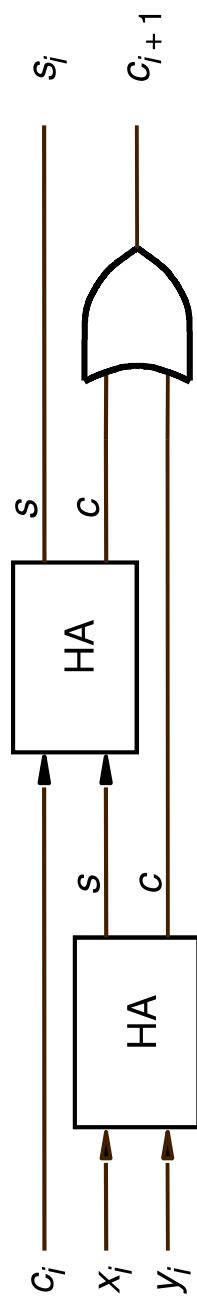
(a) Half-adder truth table and implementation

A	B	C <sub>in</sub>	Sum	C <sub>out</sub>
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

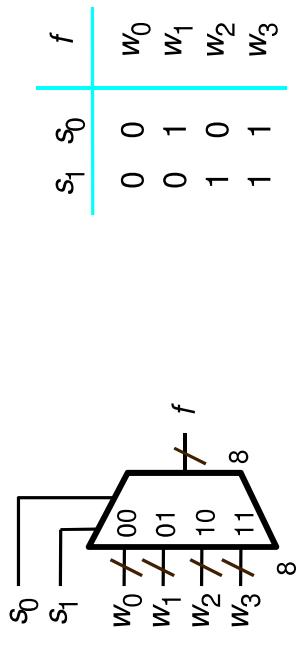
# Adders



A decomposed implementation of the full-adder circuit

# **Multi-Bit Combinational Logic Building Blocks**

# Multi-bit 4-to-1 Multiplexer



(a) Graphic symbol

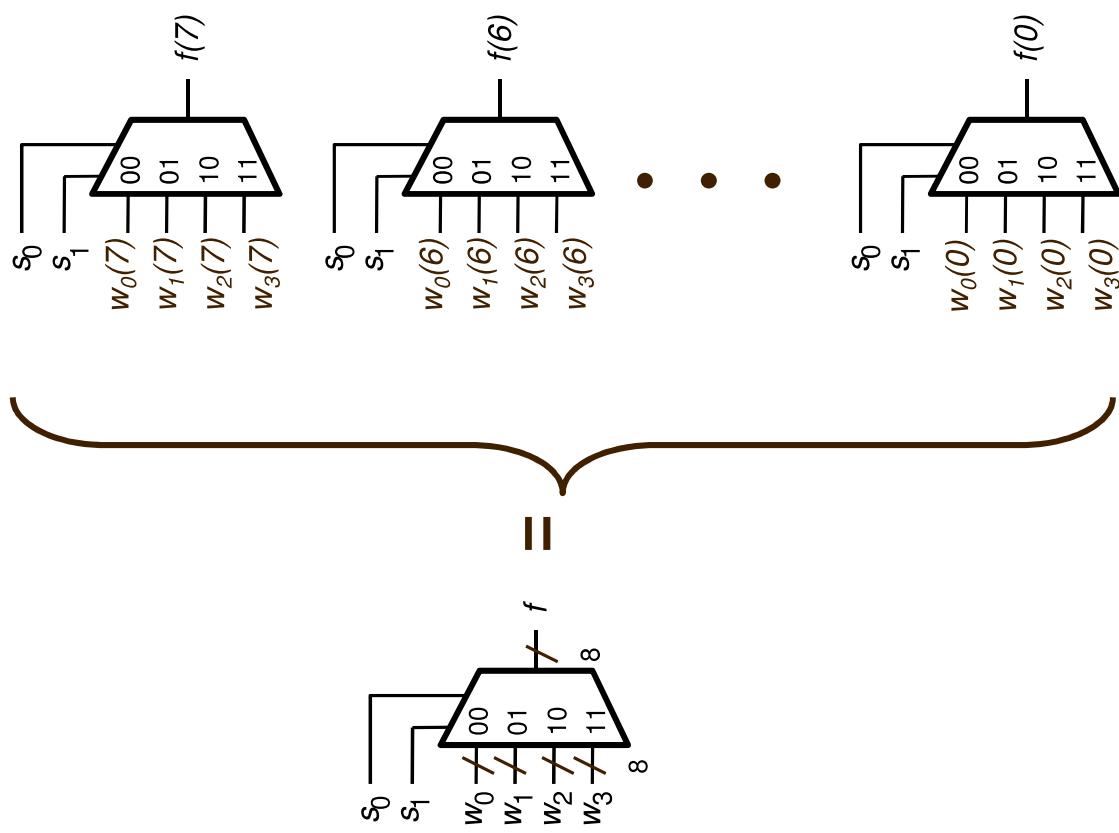
$s_1$	$s_0$	$f$
0	0	$w_0$
0	1	$w_1$
1	0	$w_2$
1	1	$w_3$

(b) Truth table

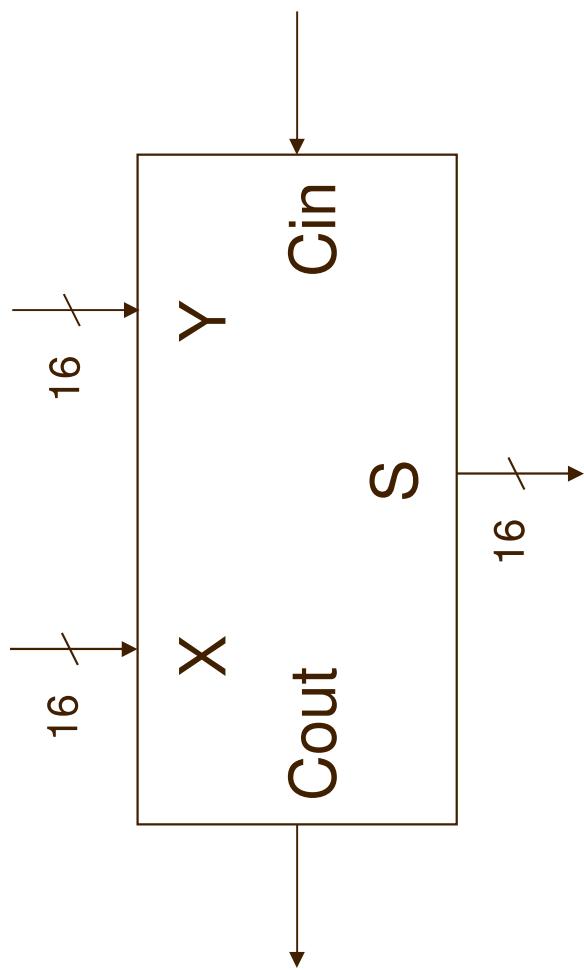
- When drawing schematics, can draw **multi-bit** multiplexers
- Example: 4-to-1 (8 bit) multiplexer
  - 4 inputs (each 8 bits)
  - 1 output (8 bits)
  - 2 selection bits
- Can also have multi-bit 2-to-1 muxes, 16-to-1 muxes, etc.

# 4-to-1 (8-bit) Multiplexer

A 4-to-1 (8-bit) multiplexer is composed  
of eight 4-to-1 (1-bit) multiplexers



# 16-bit Unsigned Adder

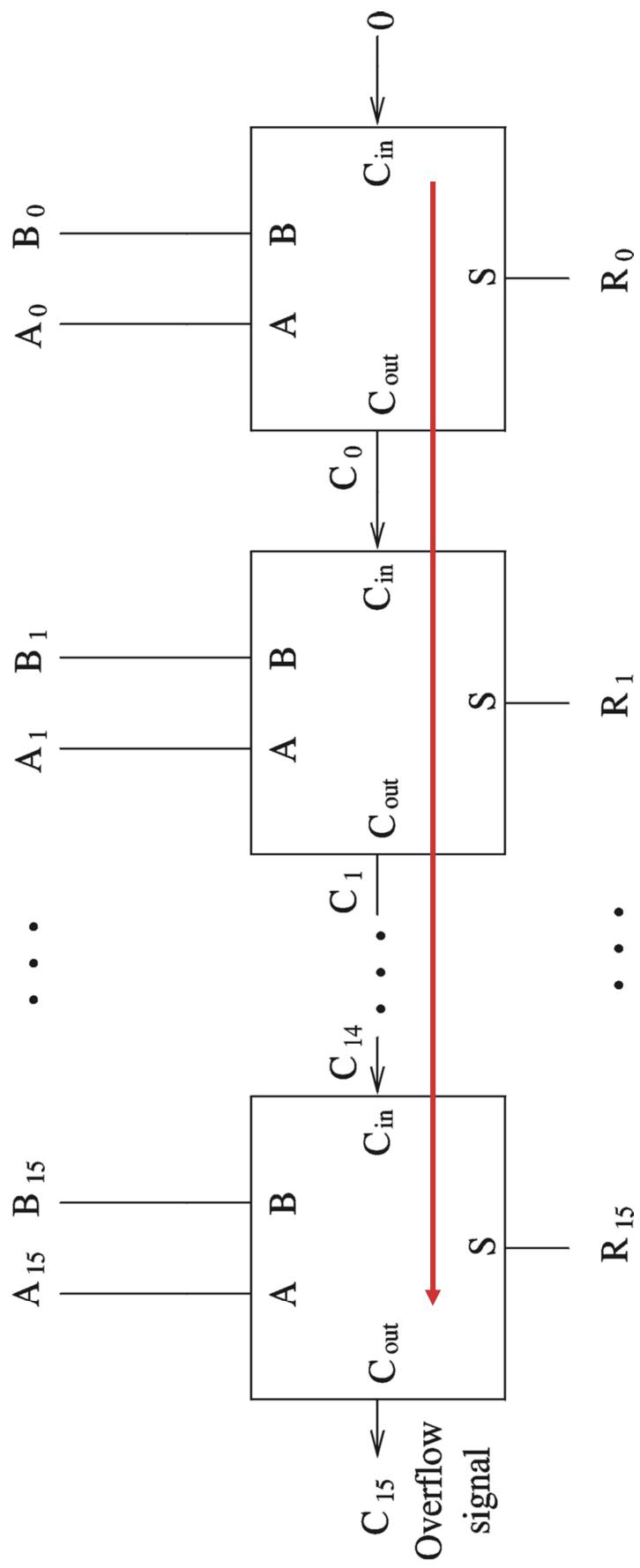


# Multi-Bit Ripple-Carry Adder

A 16-bit ripple-carry adder is composed of 16 (1-bit) full adders

Inputs: 16-bit A, 16-bit B, 1-bit carryin (set to zero in the figure below)

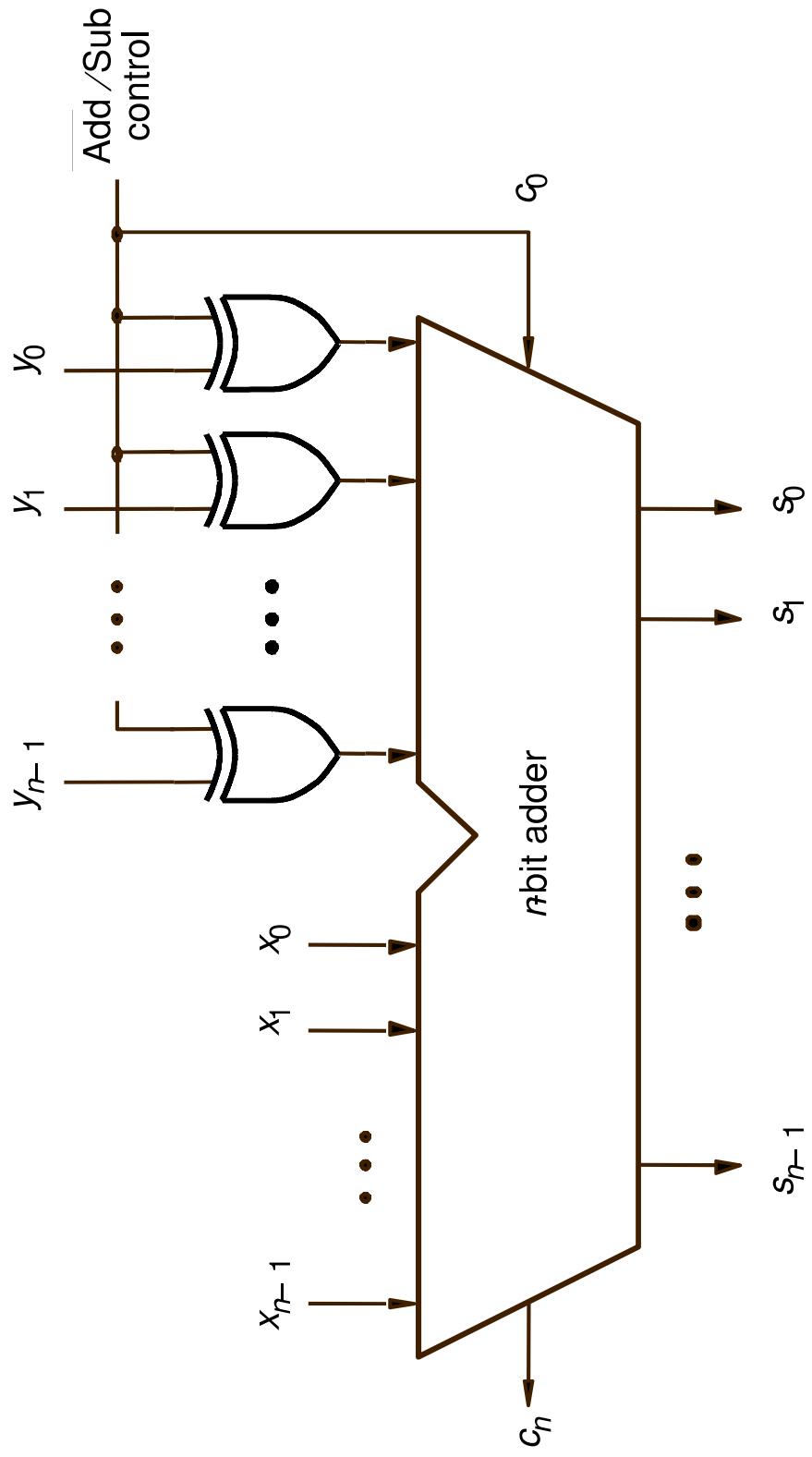
Outputs: 16-bit sum R, 1-bit overflow



Called a ripple-carry adder because carry ripples from one full-adder to the next.

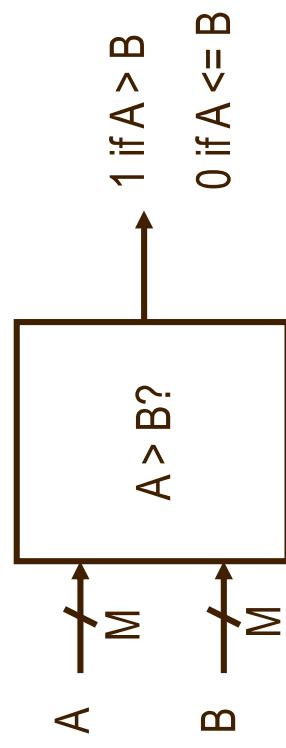
Critical path is 16 full-adders.

# Adder/subtractor unit

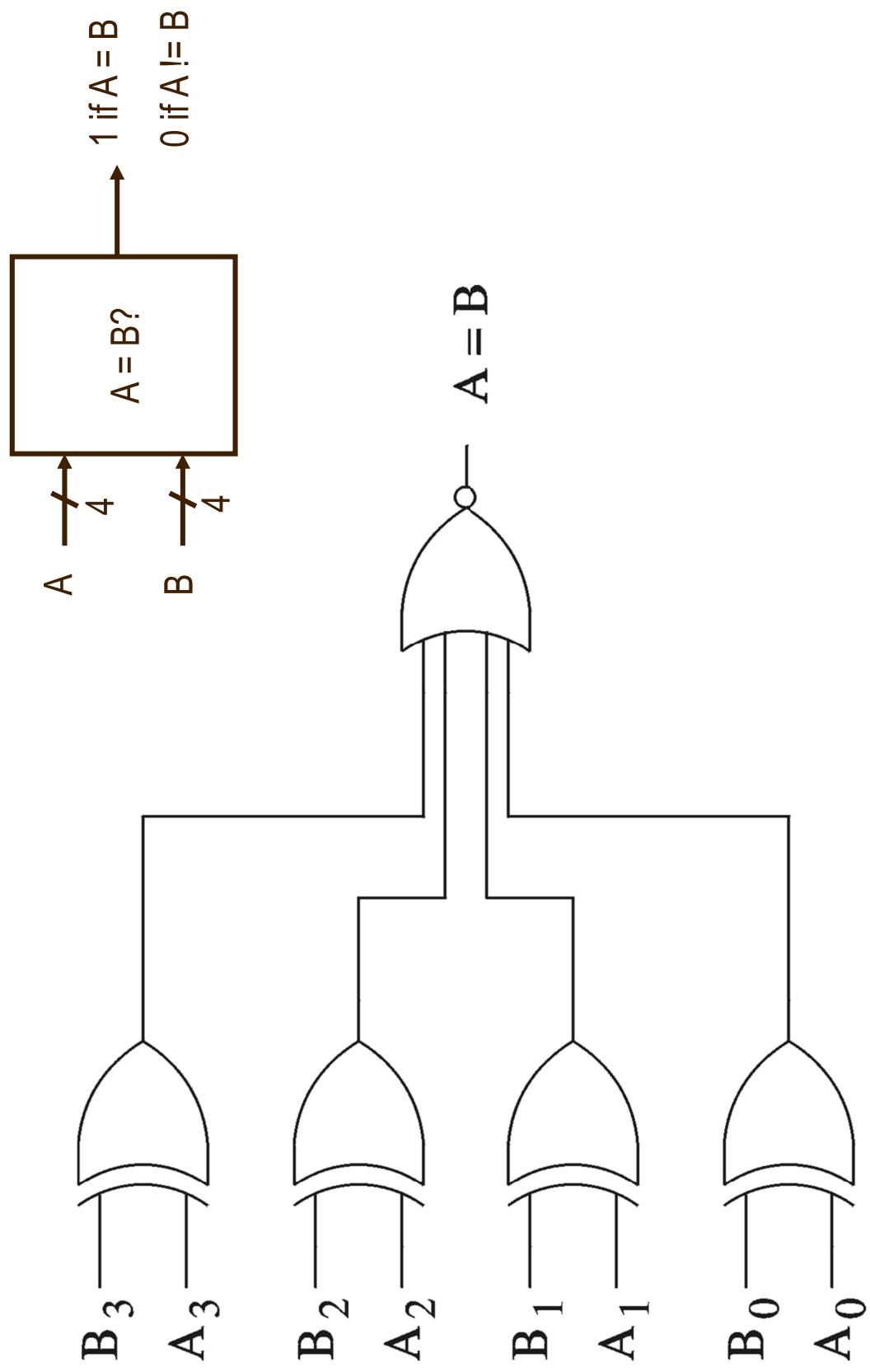


# Comparator

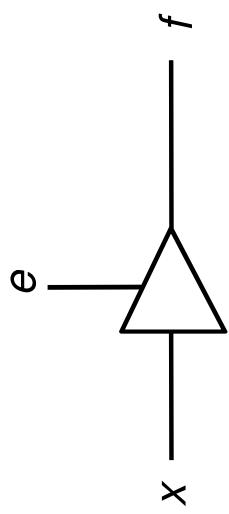
- Used two compare two M-bit numbers and produce a flag ( $M > 1$ )
  - Inputs: M-bit input A, M-bit input B
  - Output: 1-bit output flag
    - 1 indicates condition is met
    - 0 indicates condition is not met
  - Can compare:  $>$ ,  $\geq$ ,  $<$ ,  $\leq$ ,  $=$ , etc.



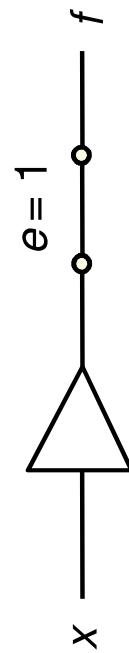
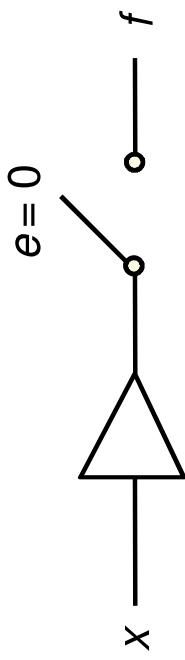
## Example: 4-bit comparator ( $A = B$ )



# Tri-state Buffer



(a) A tri-state buffer

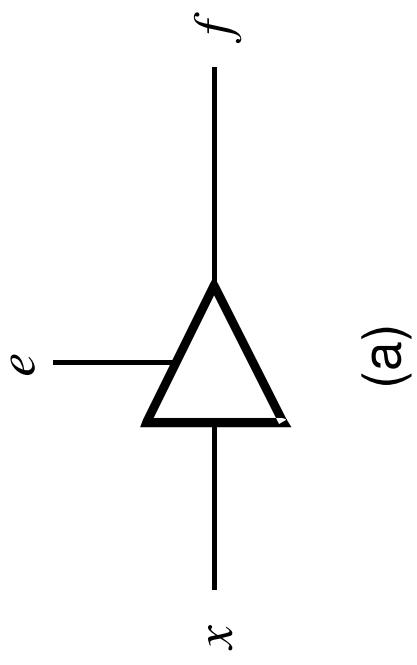


(b) Equivalent circuit

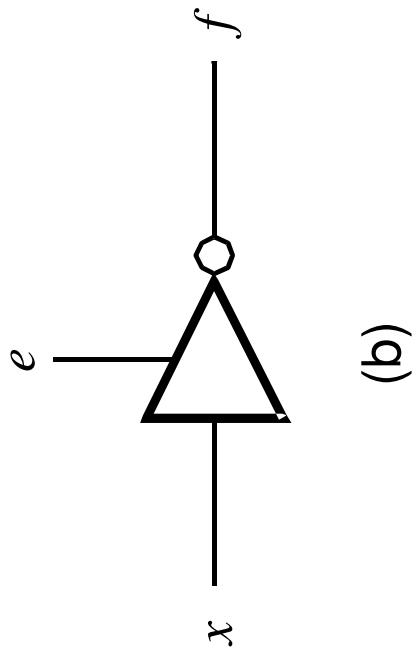
$e$	$x$	$f$
0	0	Z
0	1	Z
1	0	0
1	1	1

(c) Truth table

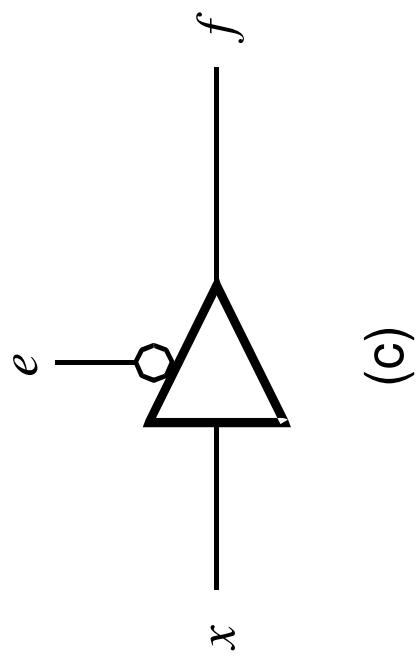
# Four types of Tri-state Buffers



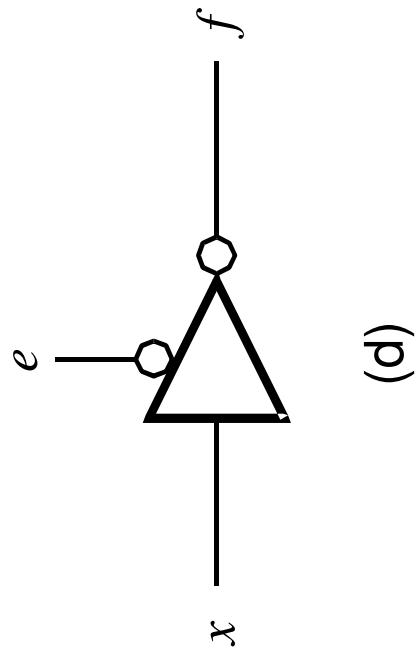
(a)



(b)



(c)



(d)

# Sequential Logic Building Blocks

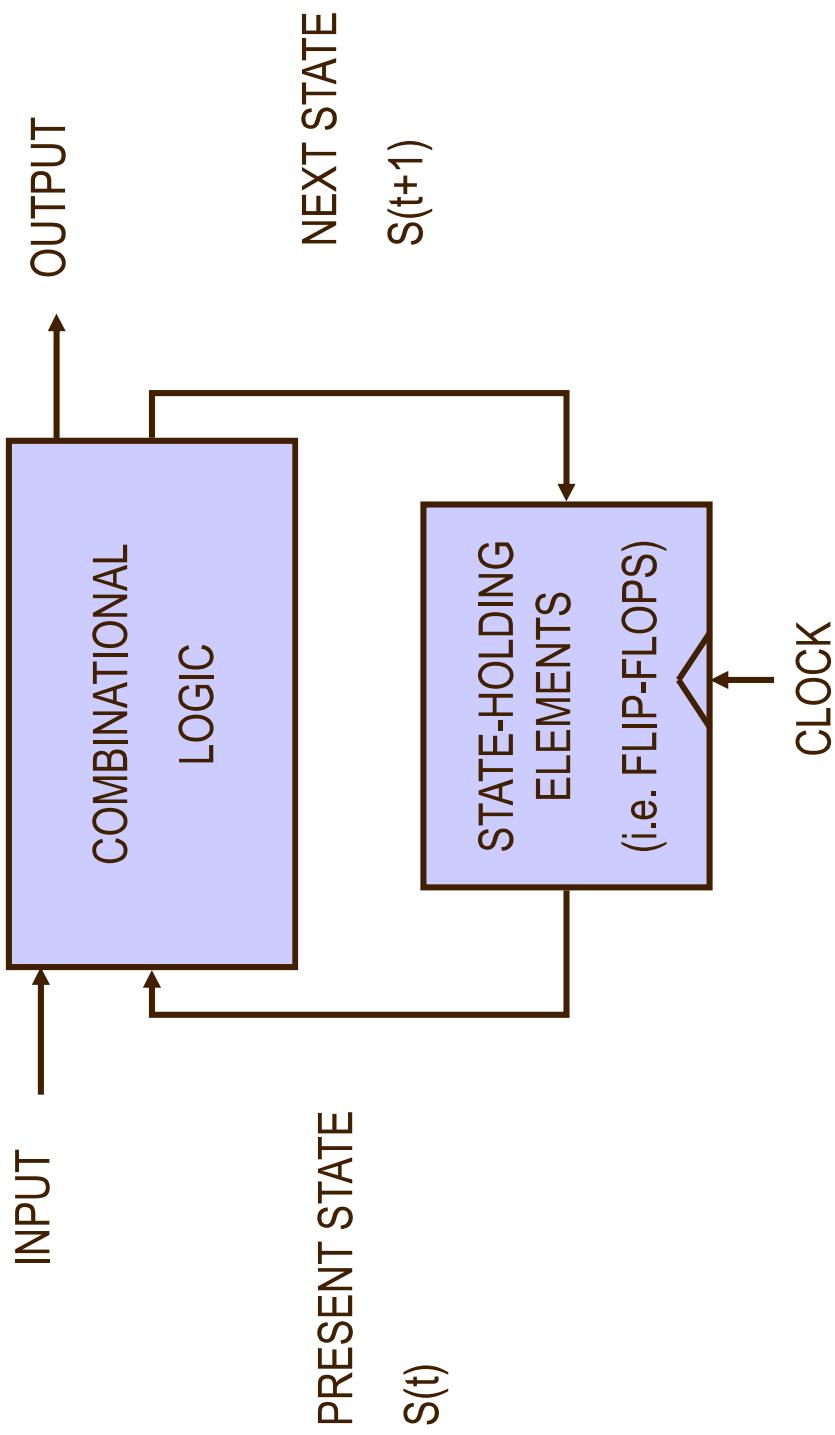
# Introduction to Sequential Logic

---

- 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'd)

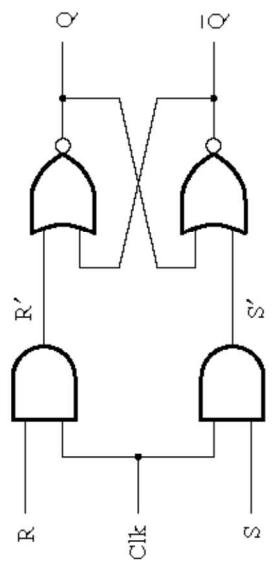
Main components of a typical synchronous sequential circuit  
(synchronous = uses a clock to keep circuits in lock step)



# State-Holding Memory Elements

- Latch versus Flip Flop
  - Latches are level-sensitive: whenever clock is high, latch is transparent
  - Flip-flops are edge-sensitive: data passes through (i.e. data is sampled) only on a rising (or falling) edge of the clock
  - Latches cheaper to implement than flip-flops
  - Flip-flops are easier to design with than latches
- In this course, primarily use D flip-flops

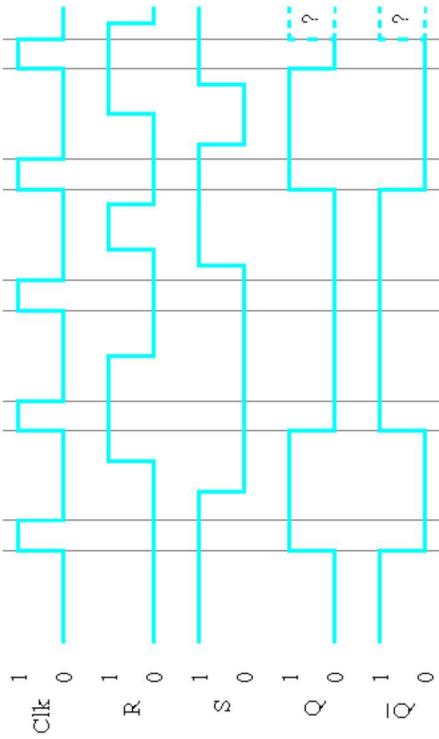
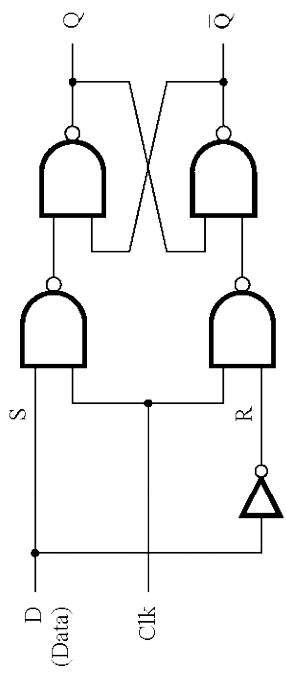
# Latches and Flip-Flops



(a) Circuit

Clk	S	R	Q(t+1)
0	x	x	Q(t) (no change)
1	0	0	Q(t) (no change)
1	0	1	0
1	1	0	1
1	1	1	x

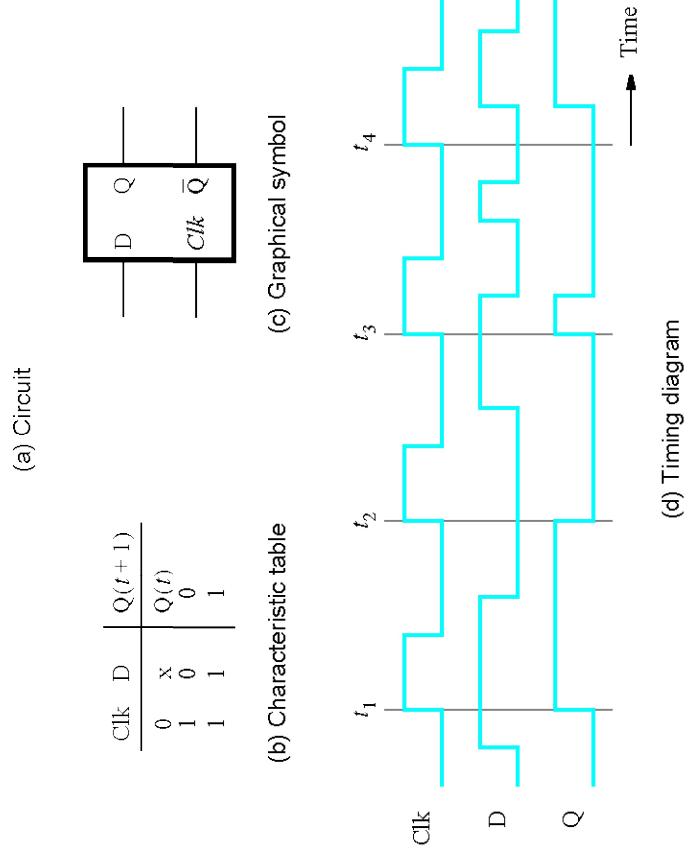
(b) Characteristic table



(c) Timing diagram

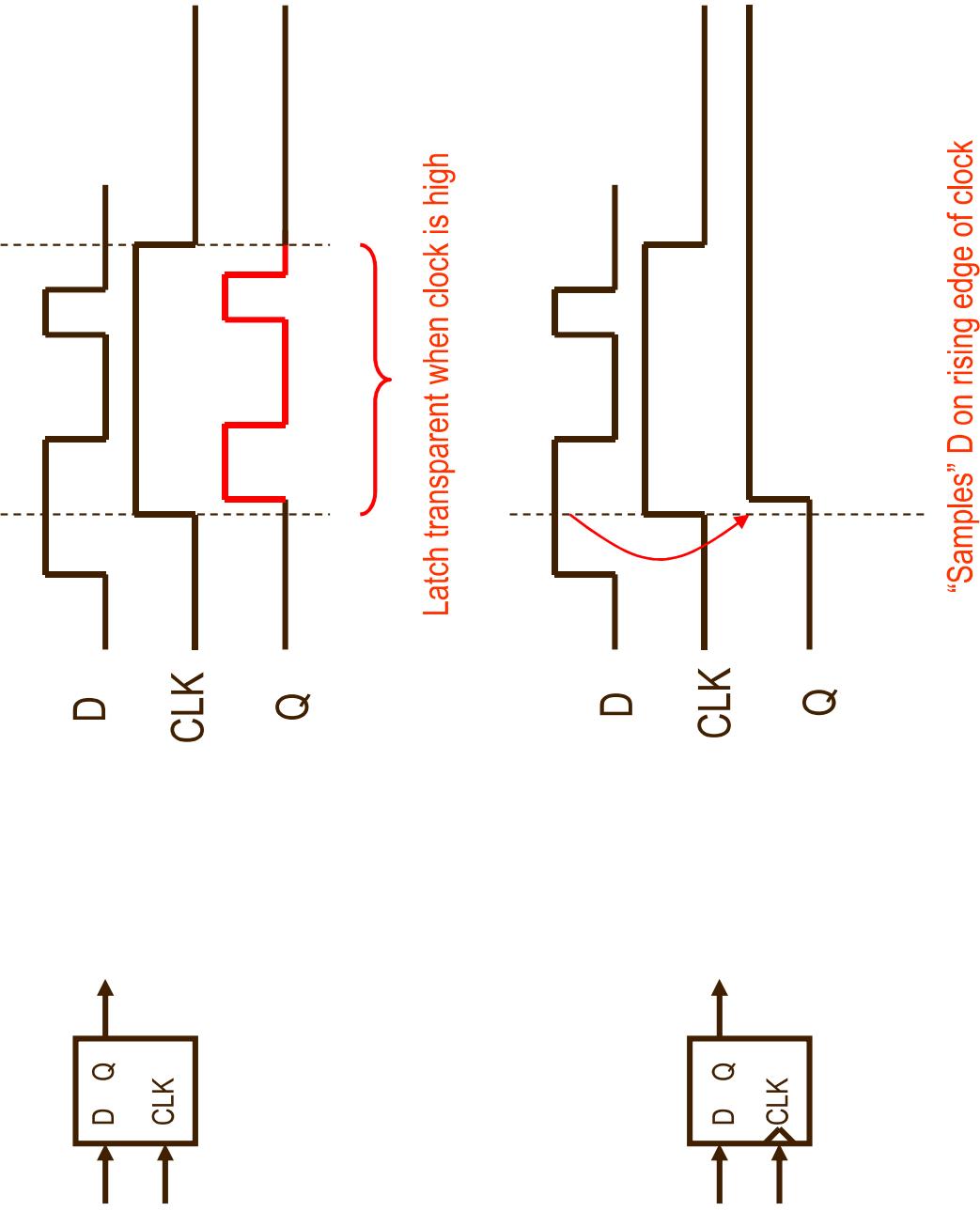
Clk	D	Q(t+1)
0	x	Q(t)
1	0	0
1	1	1

(b) Characteristic table



(d) Graphical symbol

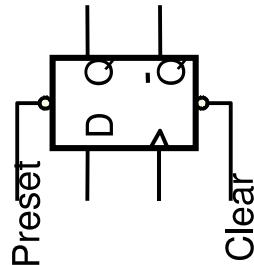
# D Latch vs. D Flip-Flop



# D Flip-Flop with Asynchronous Preset and Clear

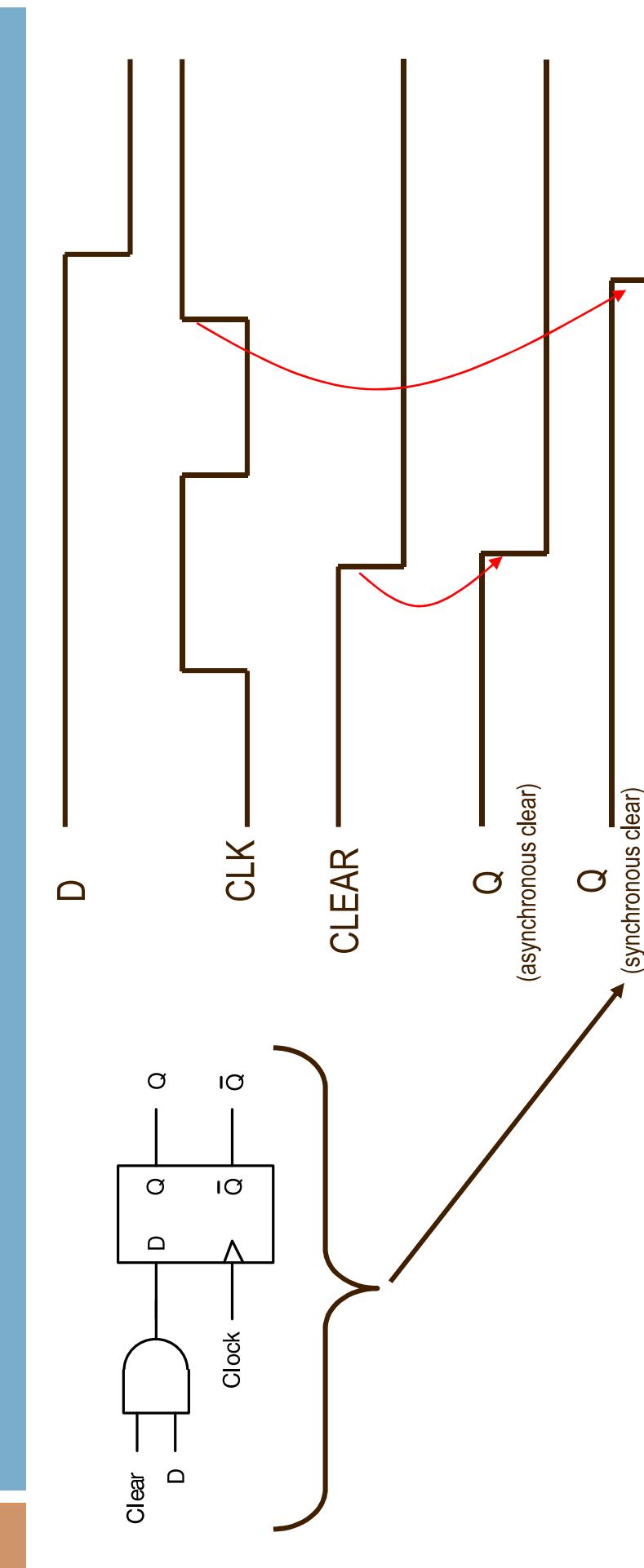
- Bubble on the symbol means “active-low”
  - When preset = 0, preset Q to 1
  - When preset = 1, do nothing
  - When clear = 0, clear Q to 0
  - When clear = 1, do nothing
- “Preset” and “Clear” also known as “Set” and “Reset” respectively
- In this circuit, preset and clear are asynchronous
  - Q changes immediately when preset or clear are active, regardless of clock

(a) Circuit



(b) Graphical symbol

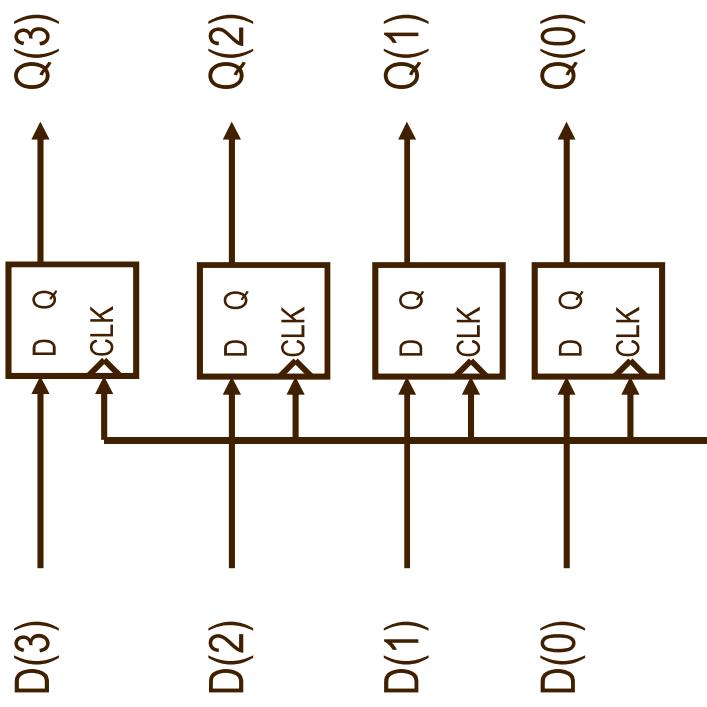
# D Flip-Flop with Synchronous Clear



- Asynchronous active-low clear: Q immediately clears to 0
- Synchronous active-low clear: Q clears to 0 on rising-edge of clock

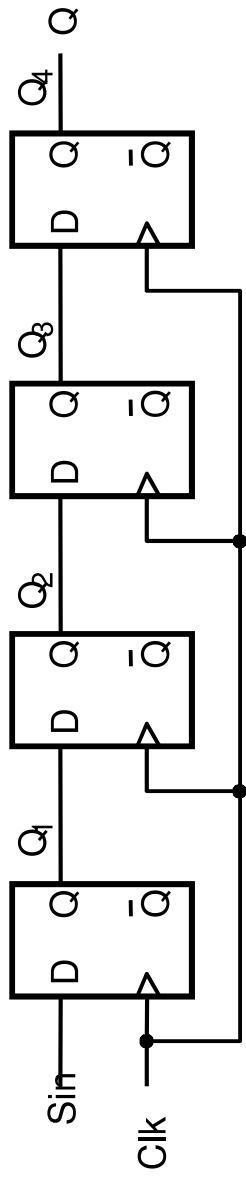
# Sequential Logic Circuits

# Register



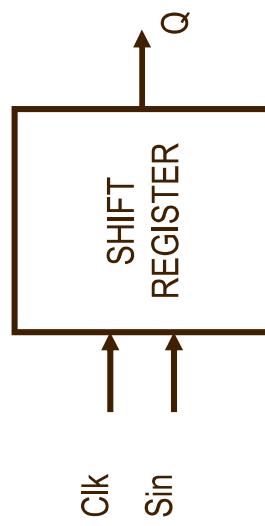
- In typical nomenclature, a register is a name for a collection of flip-flops used to hold a bus (i.e. `std_logic_vector`)

# Shift Register



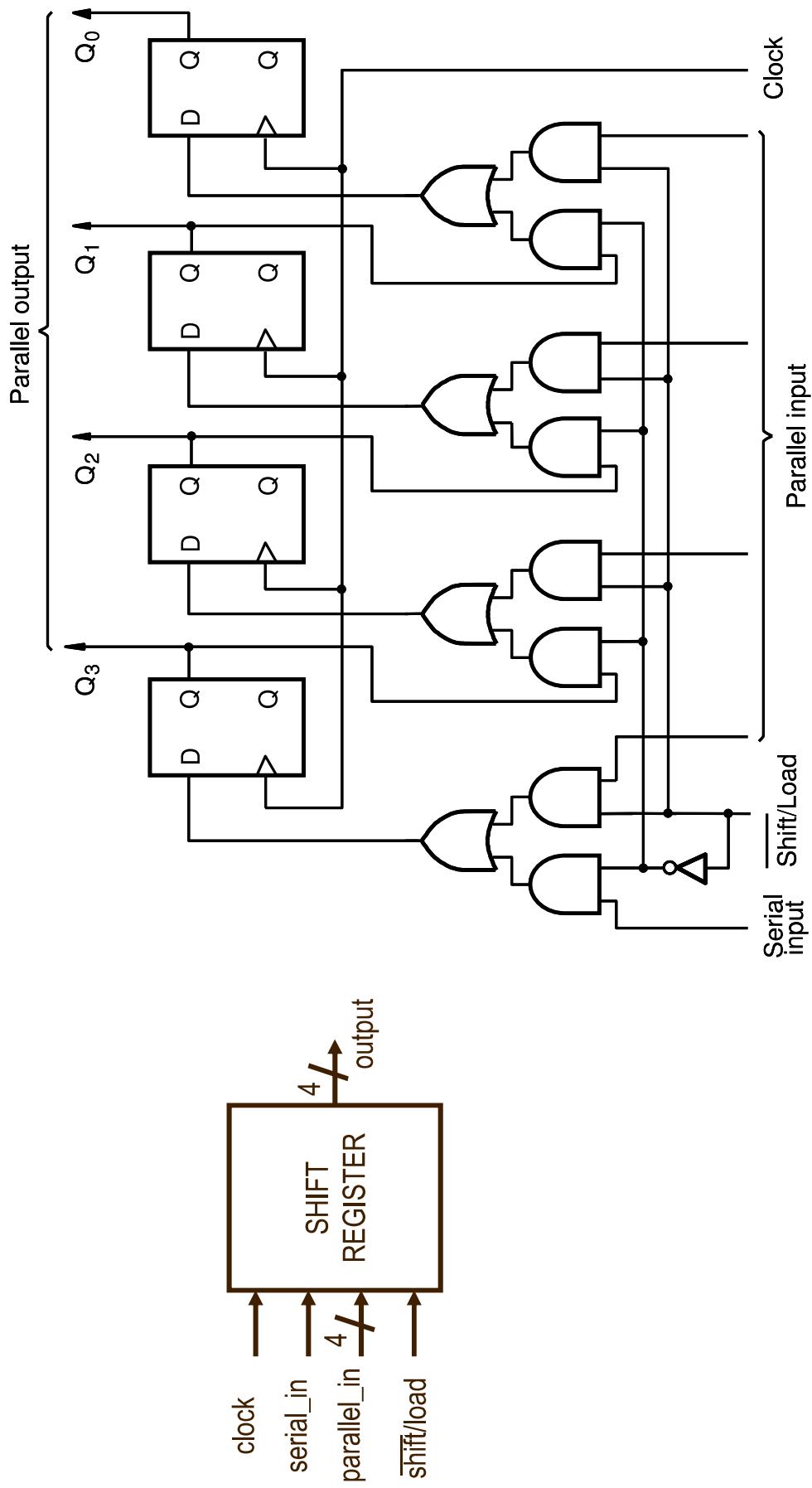
(a) Circuit

	Sin	$Q_1$	$Q_2$	$Q_3$	$Q_4 = Q$
$t_0$	1	0	0	0	0
$t_1$	0	1	0	0	0
$t_2$	1	0	1	0	0
$t_3$	1	1	0	1	0
$t_4$	1	1	1	0	1
$t_5$	0	1	1	1	0
$t_6$	0	0	1	1	1
$t_7$	0	0	0	1	1

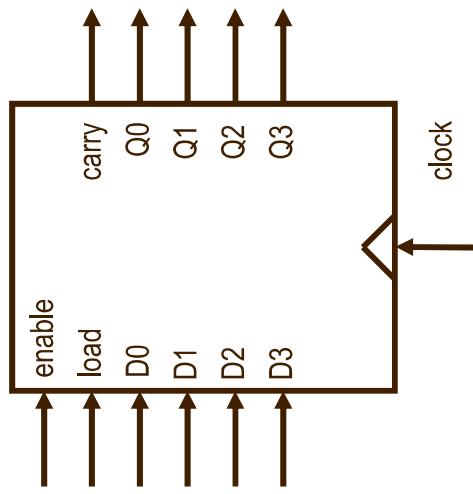


(b) A sample sequence

# Parallel Access Shift Register



# Synchronous Up Counter

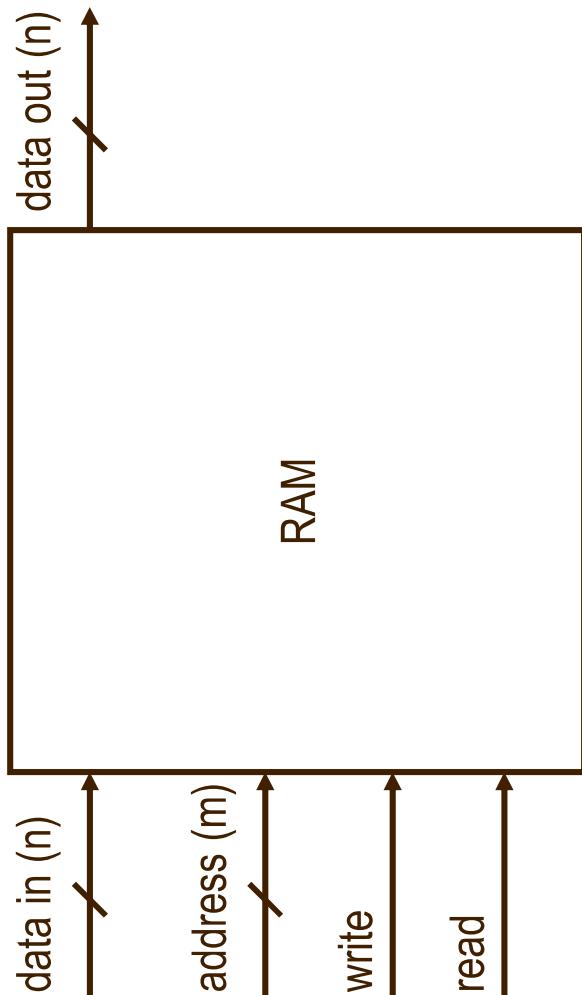


- Enable (synchronous): when high enables the counter, when low counter holds its value
- Load (synchronous) : when load = 1, load the desired value into the counter
- Output carry: indicates when the counter “rolls over”
- D3 downto D0, Q3 downto Q0 is how to interpret MSB to LSB

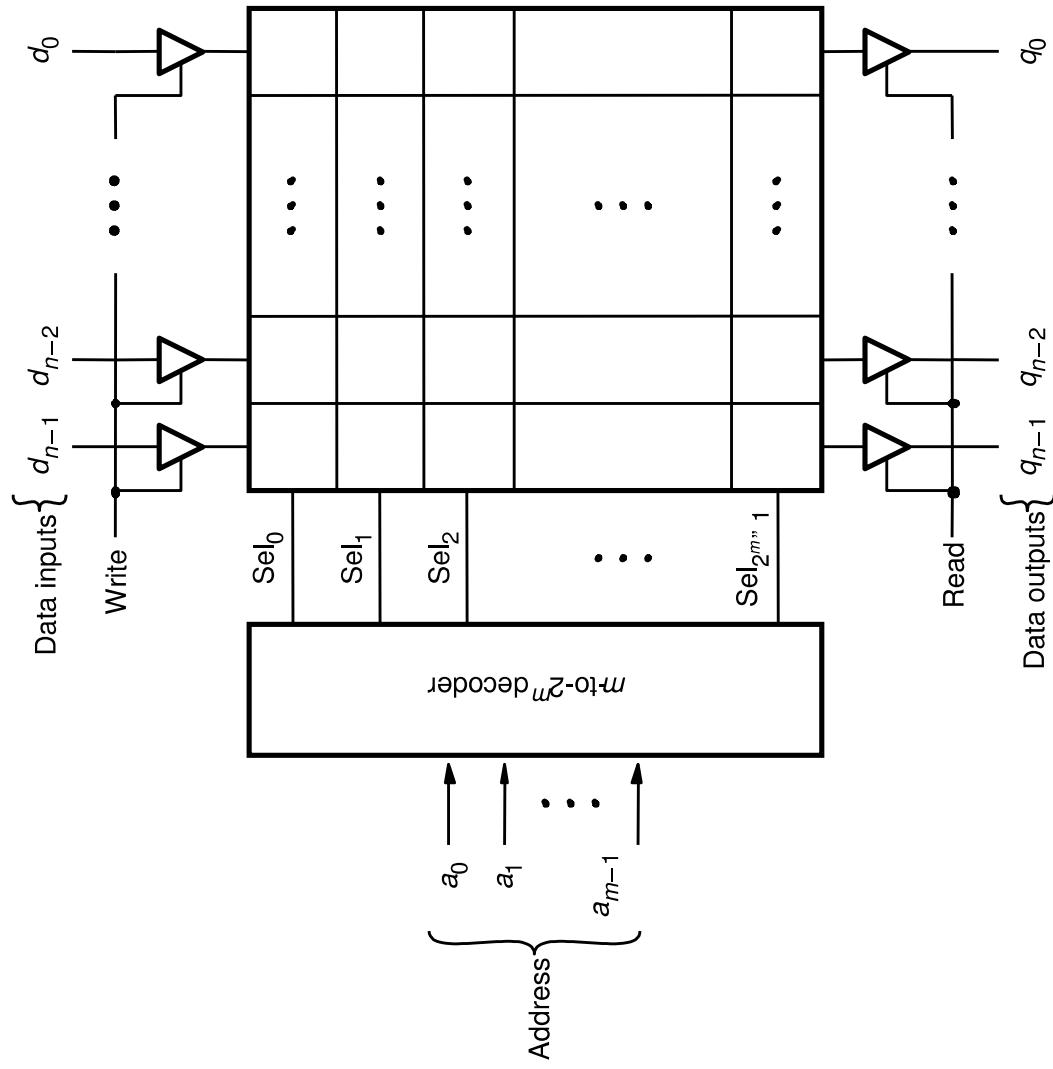
# Memories

# Random Access Memory (RAM)

- More efficient than registers for storing large amounts of data
- Can read and write to RAM
- Addressable memory
  - Can be synchronous (with clock) or asynchronous (no clock)
  - SRAM dimensions are:
    - (number of words) x (bits per word)  
SRAM
    - Address is m bits, data is n bits
      - $2^m \times n$ -bit RAM
  - Example: address is 5 bits, data is 8 bits
    - 32 x 8-bit RAM
- Write
  - Data\_in and address are stable
  - Assert write signal (then de-assert)
- Read
  - Address is stable
  - Assert read signal
  - Data\_out is valid

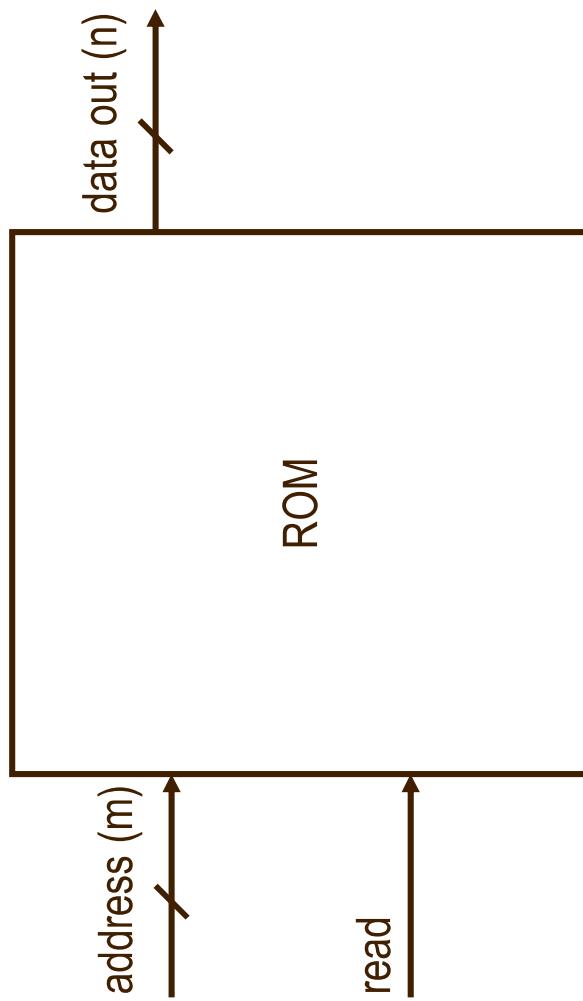


# Random Access Memory (RAM)



# Read Only Memory (ROM)

- Similar to RAM except read only
- Addressable memory
  - Can be synchronous (with clock) or asynchronous (no clock)



# Read-Only Memory (ROM)

