Logic and Computer Design Fundamentals

Part 1 – Registers, Microoperations and Implementations

Charles Kime & Thomas Kaminski

(Hyperlinks are active in View Show mode)

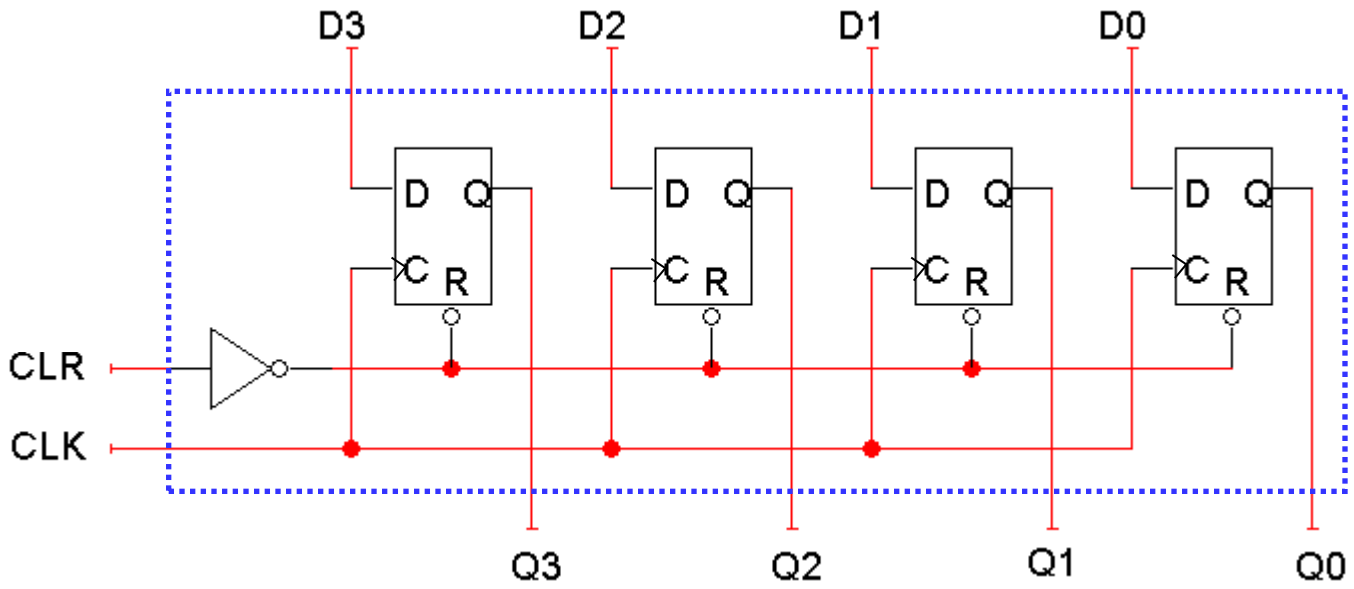
Overview

- **Part 1 - Registers, Microoperations and Implementations**
  - **Registers and load enable**
  - **Register transfer operations**
  - **Microoperations - arithmetic, logic, and shift**
  - **Microoperations on a single register**
    - **Multiplexer-based transfers**
    - **Shift registers**
- **Part 2 - Counters, Register Cells, Buses, & Serial Operations**
- **Part 3 – Control of Register Transfers**

Registers

- **Register – a collection of binary storage elements**
- **In theory, a register is sequential logic which can be defined by a state table**
- **More often, think of a register as storing a vector of binary values**
- **Frequently used to perform simple data storage and data movement and processing operations**

Registers



Example: 2-bit Register

- How many states are there? $2^2=4$
- How many input combinations? $2^2=4$
- How many output combinations? $2^2=4$
- What is the output function?

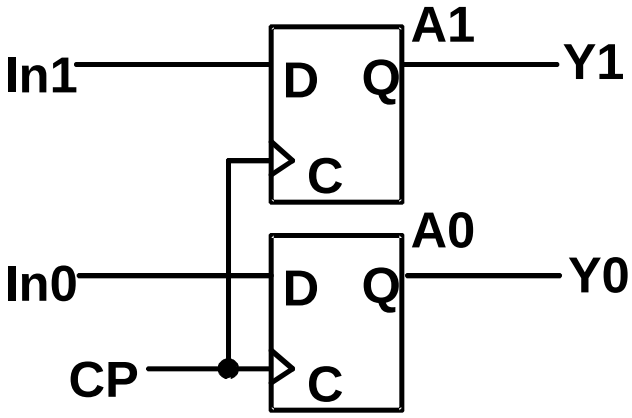
  $Y_1 = A_1$

  $Y_0 = A_0$
- What is the next state function?

  $A_1(t+1) = IN_1$

  $A_0(t+1) = IN_0$
- Moore or Mealy?

  Moore



State Table

Current State A_1 A_0	Next State $A_1(t+1)$ $A_0(t+1)$ For In_1 In_0 = 00 01 10 11	Output ($=A_1$ A_0) Y_1 Y_0
0 0	00 01 10 11	0 0
0 1	00 01 10 11	0 1
1 0	00 01 10 11	1 0
1 1	00 01 10 11	1 1

- What are the quantities above for an *n*-bit register?

  States = 2^n

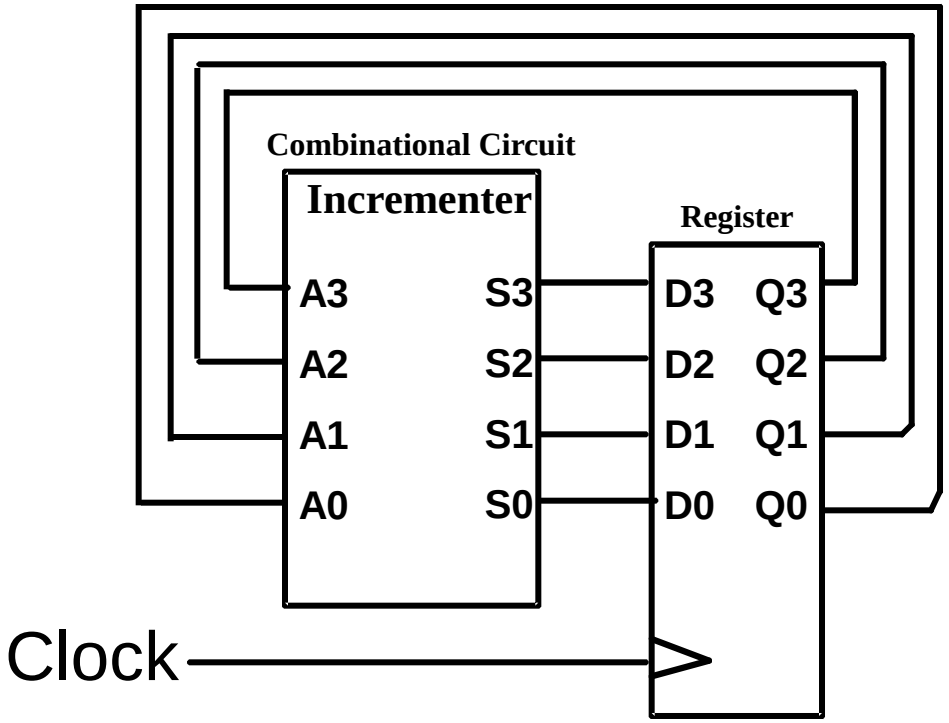
  Input Combinations = 2^n

  Output Combinations = 2^n

Register Design Models

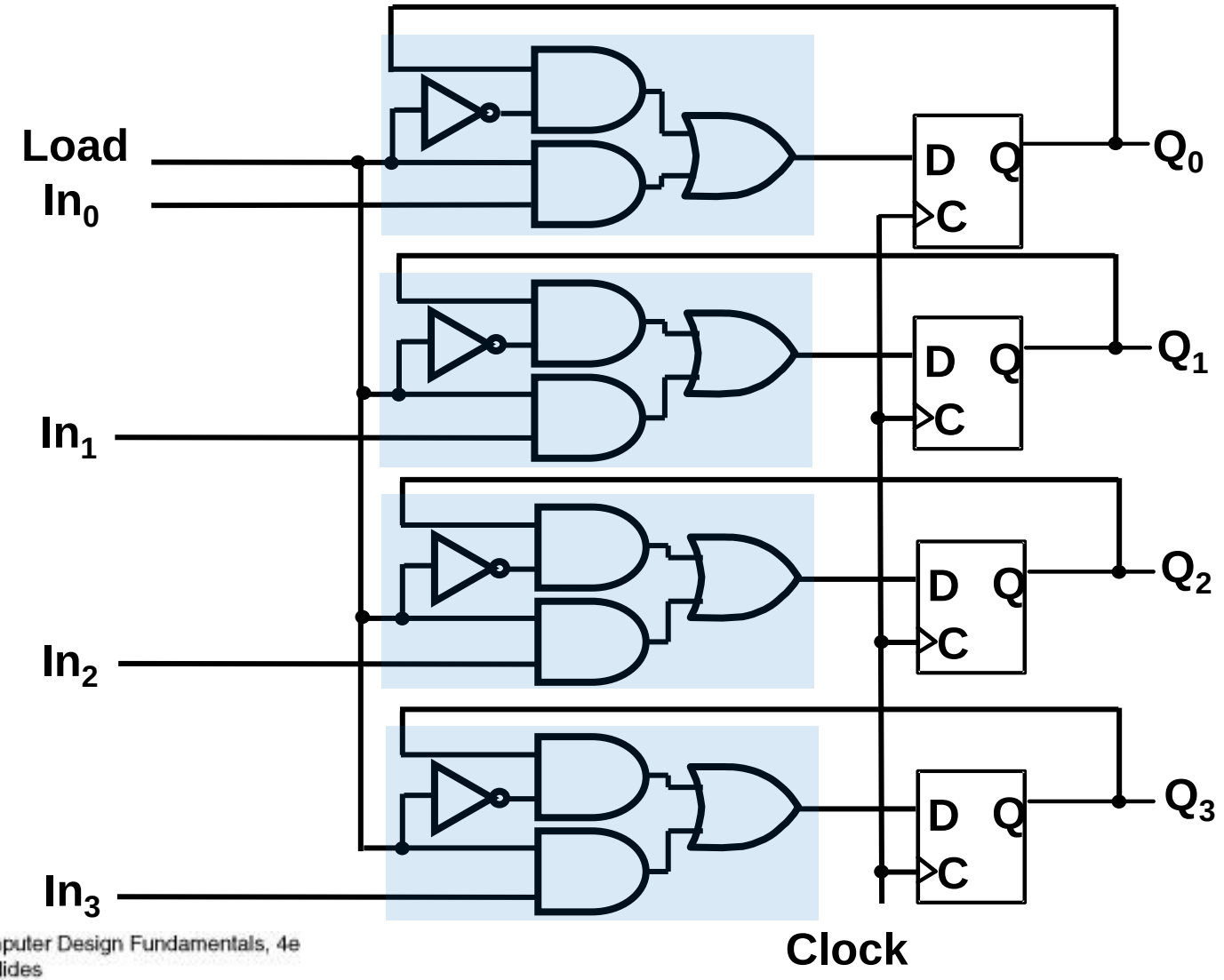
- **Due to the large numbers of states and input combinations as *n* becomes large, the state diagram/state table model is not feasible!**
- **What are methods we can use to design registers?**
  - **Add predefined combinational circuits to registers**
    - **Example: To count up, connect the register flip-flops to an incrementer**
  - **Design individual cells using the state diagram/state table model and combine them into a register**
    - **A 1-bit cell has just two states**
    - **Output is usually the state variable**

Add Combinational Circuits to Registers



Counter

Design Individual Cells

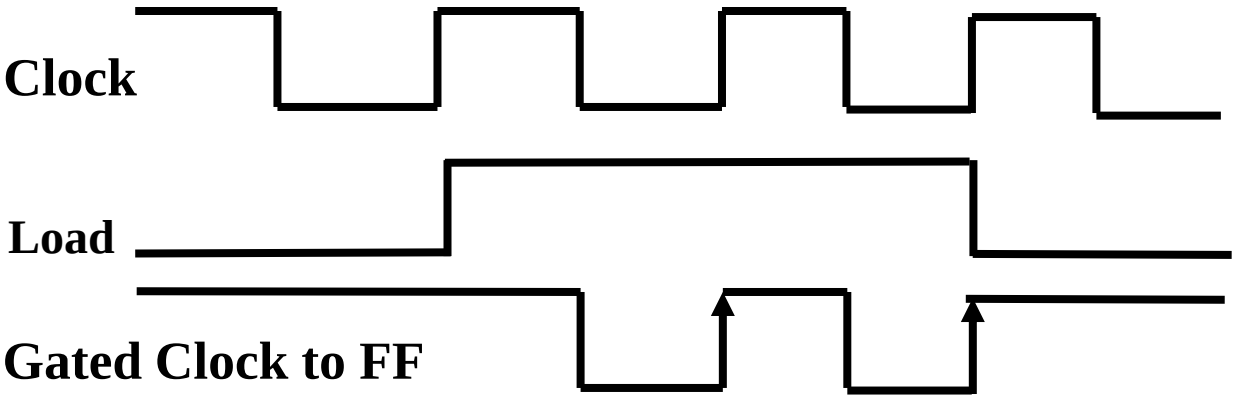


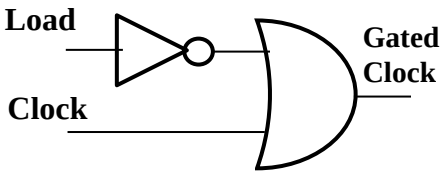
Register Storage

- **Expectations:**
  - **A register can store information for multiple clock cycles**
  - **To "store" or "load" information should be controlled by a signal**
- **Reality:**
  - **A D flip-flop register loads information on every clock cycle**
- **Realizing expectations:**
  1. **Use a signal to block the clock to the register,**
  2. **Use a signal to control feedback of the output of the register back to its inputs, or**
  3. **Use other SR or JK flip-flops, that for (0,0) applied, store their state**
- **Load is a frequent name for the signal that controls register storage and loading**
  - **Load = 1: Loads input values (load new values)**
  - **Load = 0: Loads register contents (hold current values)**

Registers with Clock Gating

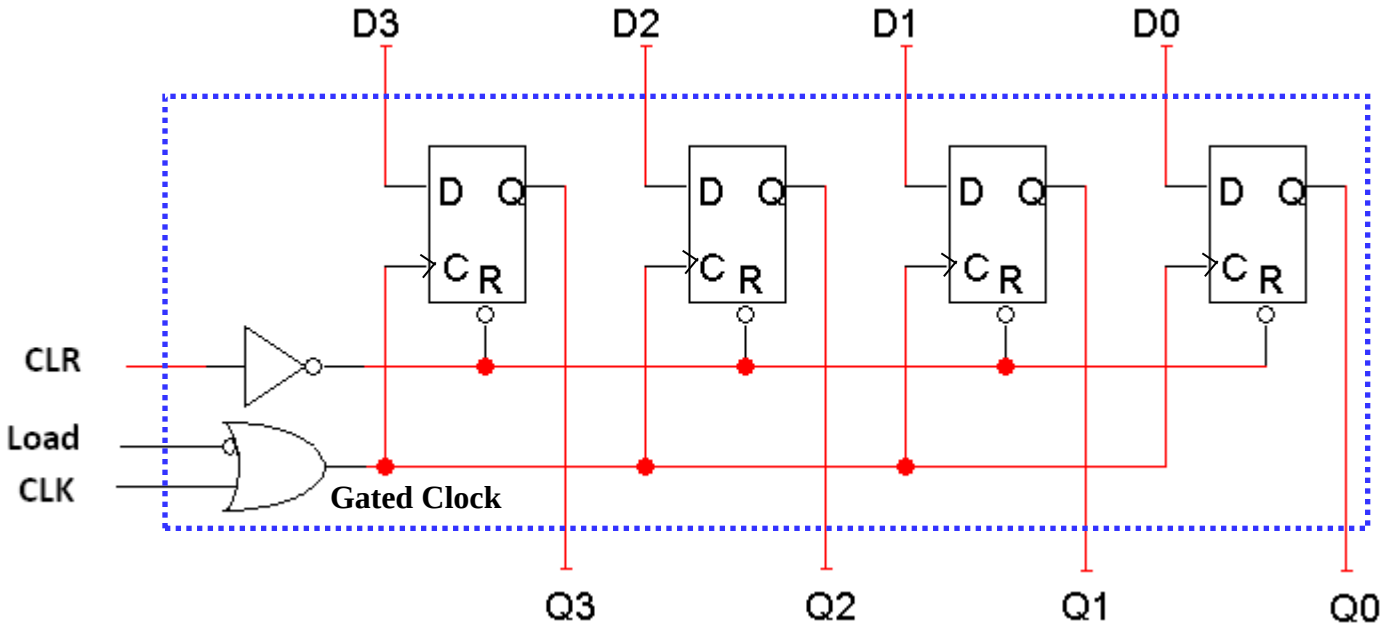
- The $\overline{Load}$ signal enables the clock signal to pass through if 1 and prevents the clock signal from passing through if 0.
- Example: For Positive Edge-Triggered or Negative Pulse Master-Slave Flip-flop:





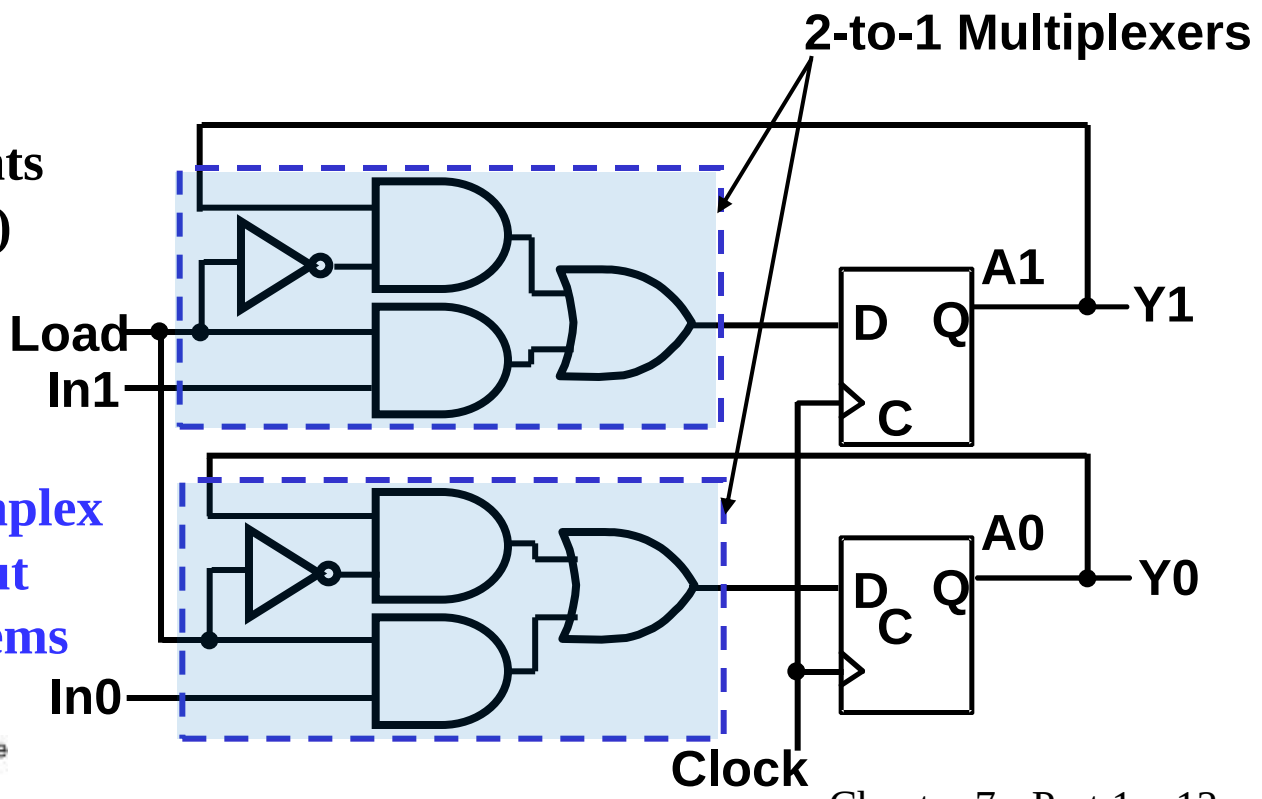
- What logic is needed for gating? Gated Clock = Clock + $\overline{Load}$
- What is the problem? Clock Skew of gated clocks with respect to clock or each other

Registers with Clock Gating

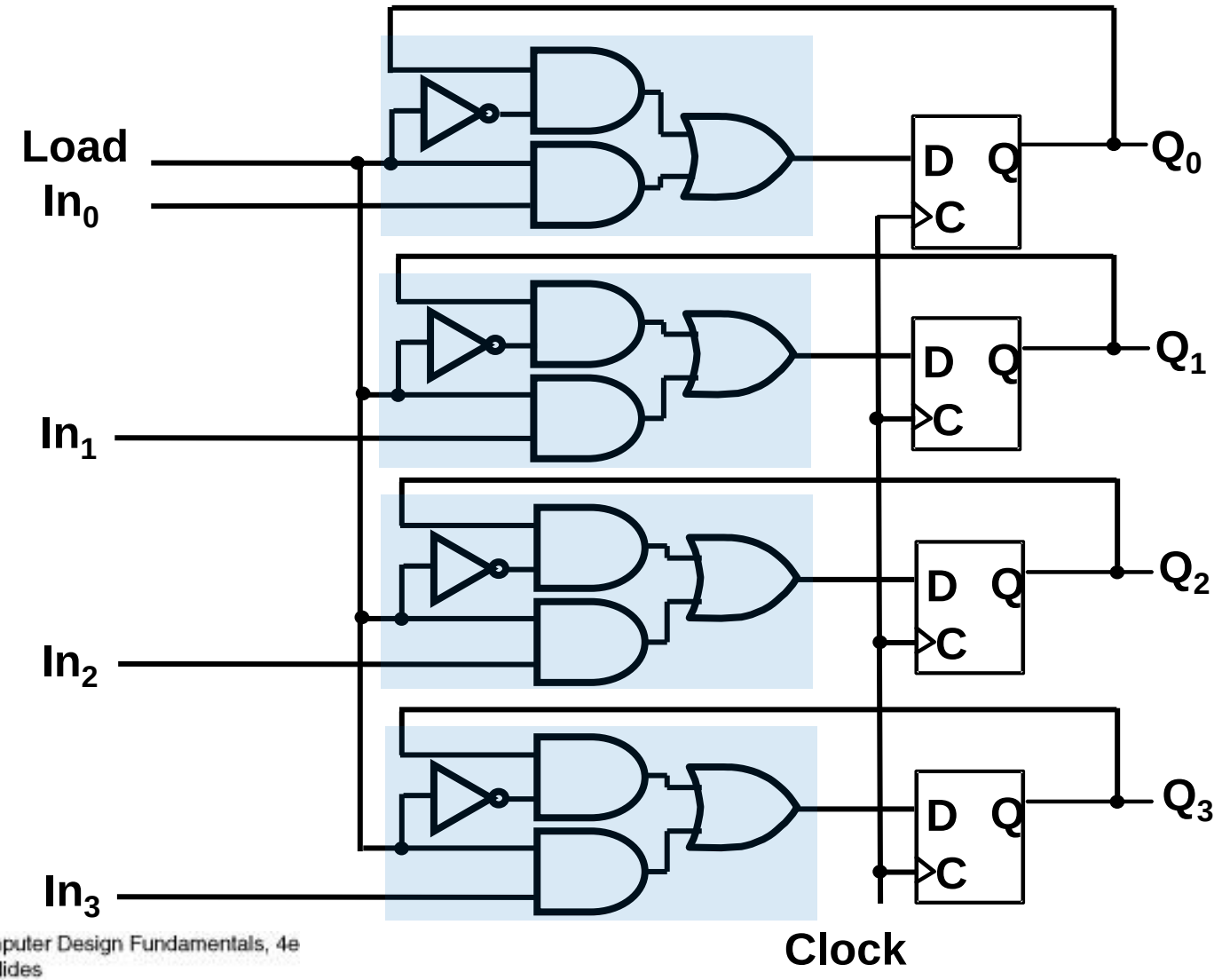


Registers with Load-Controlled Feedback

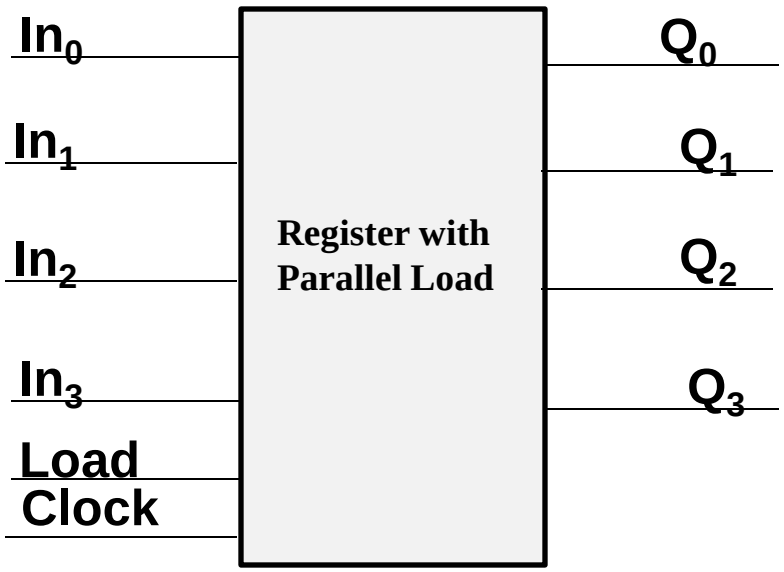
- **A more reliable way to selectively load a register:**
  - **Run the clock continuously, and**
  - **Selectively use a load control to change the register contents.**
- **Example: 2-bit register with Load Control:**
- **For Load = 0, loads register contents (hold current values)**
- **For Load = 1, loads input values (load new values)**
- **Hardware more complex than clock gating, but free of timing problems**

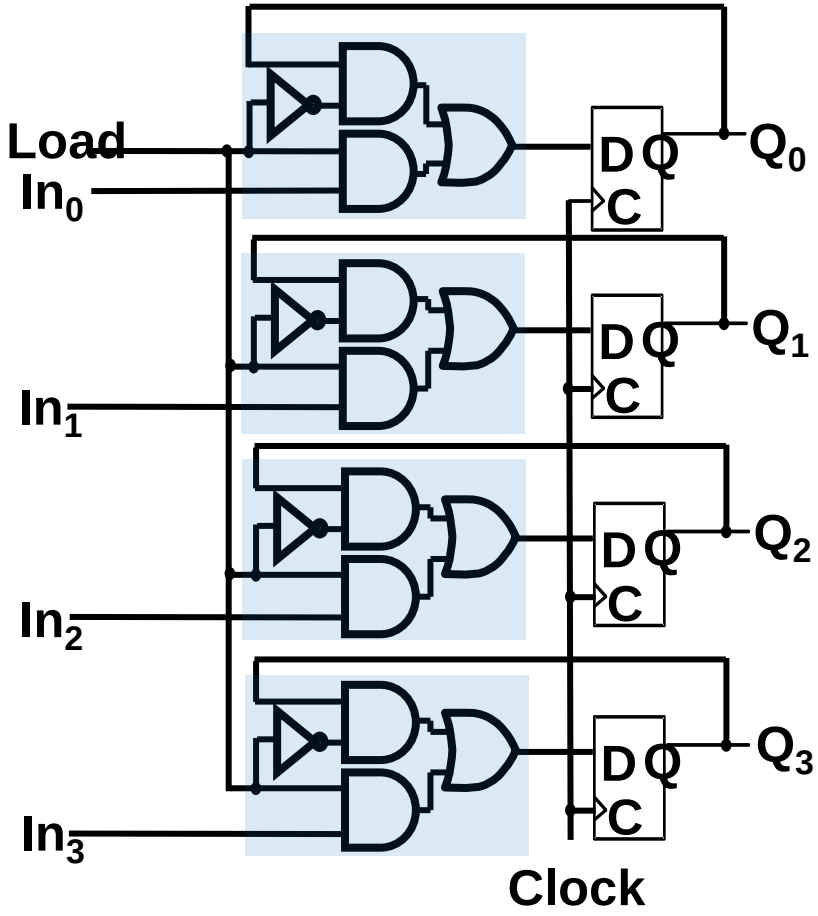


Registers with Load-Controlled Feedback



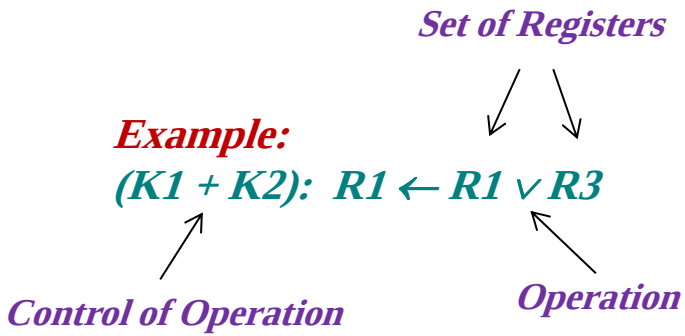
Registers with Load-Controlled Feedback





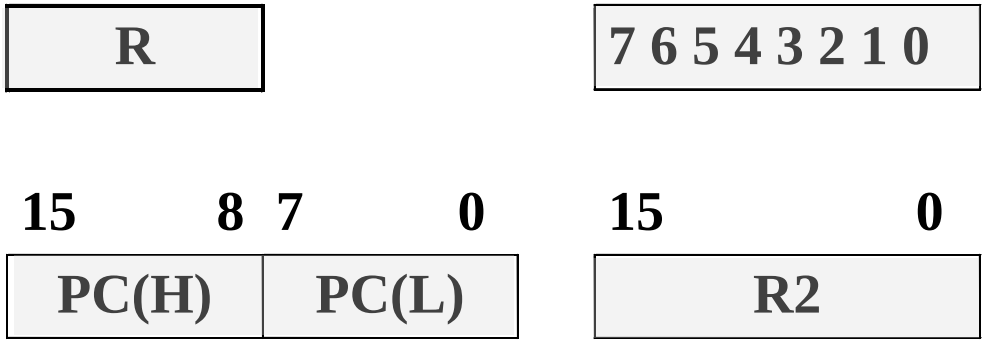
Register Transfer Operations

- **Register Transfer Operations – The movement and processing of data stored in registers**
- **Three basic components:**
  - **Set of Registers**
  - **Operations**
  - **Control of Operations**



- **Elementary Operations:**
  - **load, count, shift, add, bitwise "OR", etc.**
- **Elementary operations called *microoperations***

Register Notation



- **Letters and numbers – denotes a register (ex. R2, PC, IR)**
- **Parentheses ( ) – denotes a range of register bits**
  Example: R1(1), PC(7:0), PC(L)
- **Arrow (←) – denotes data transfer**
  Example: R1 ← R2, PC(L) ← R0
- **Comma – separates parallel operations**
- **Brackets [ ] – Specifies a memory address**
  Example: R0 ← M[AR], R3 ← M[PC]

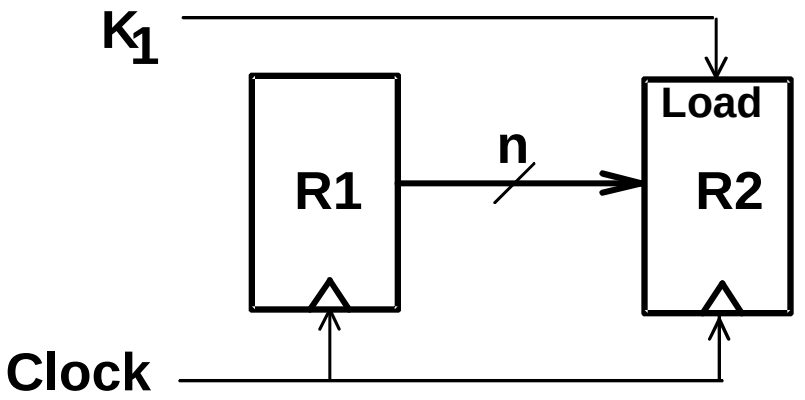
Conditional Transfer

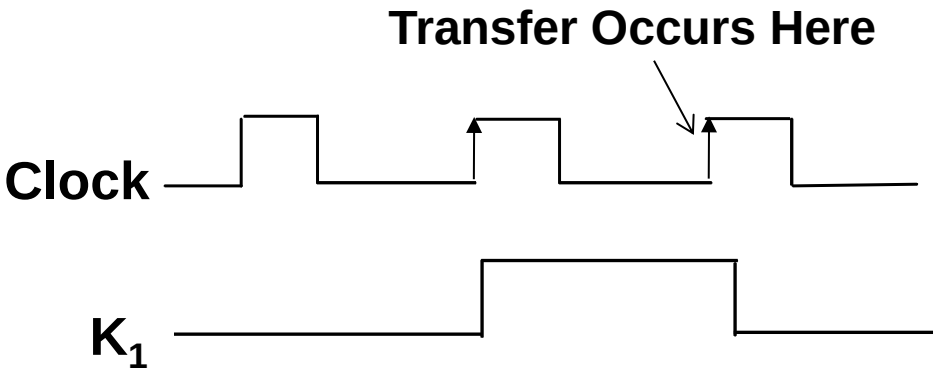
- **If (K_1 =1) then (R2 ← R1)**

  is shortened to

  K_1: (R2 ← R1)

where K_1 is a control variable specifying a conditional execution of the microoperation.





Microoperations

- **Logical Groupings:**
  - **Transfer - move data from one register to another**
  - **Arithmetic - perform arithmetic on data in registers**
  - **Logic - manipulate data or use bitwise logical operations**
  - **Shift - shift data in registers**

Arithmetic operations

+ Addition
– Subtraction
*** Multiplication**
/ Division

Logical operations

$\vee$ Logical OR
$\wedge$ Logical AND
$\oplus$ Logical Exclusive OR
$\overline{\ }$ Not

Example Microoperations

- **Add the content of R1 to the content of R2 and place the result in R1.**

  $R1 \leftarrow R1 + R2$

- **Multiply the content of R1 by the content of R6 and place the result in PC.**

  $PC \leftarrow R1 * R6$

- **Exclusive OR the content of R1 with the content of R2 and place the result in R1.**

  $R1 \leftarrow R1 \oplus R2$

Example Microoperations (Continued)

- **Take the 1's Complement of the contents of R2 and place it in the PC.**

$$PC \leftarrow \overline{R2}$$

- **On condition K_1 OR K_2, the content of R1 is Logic bitwise Ored with the content of R3 and the result placed in R1.**

$$(K1 + K2):\ R1 \leftarrow R1 \vee R3$$

- **NOTES:**

  "+" (as in $K_1 + K_2$) and means "OR."

  In R1 ← R1 + R3, + means "plus."

Control Expressions

- The control expression for an operation appears to the left of the operation and is separated from it by a colon
- Control expressions specify the logical condition for the operation to occur
- Control expression values of:
  - Logic "1" -- the operation occurs.
  - Logic "0" -- the operation is does not occur.

- Example:

  $\overline{X}\ K_1$: R1 ← R2 + R1

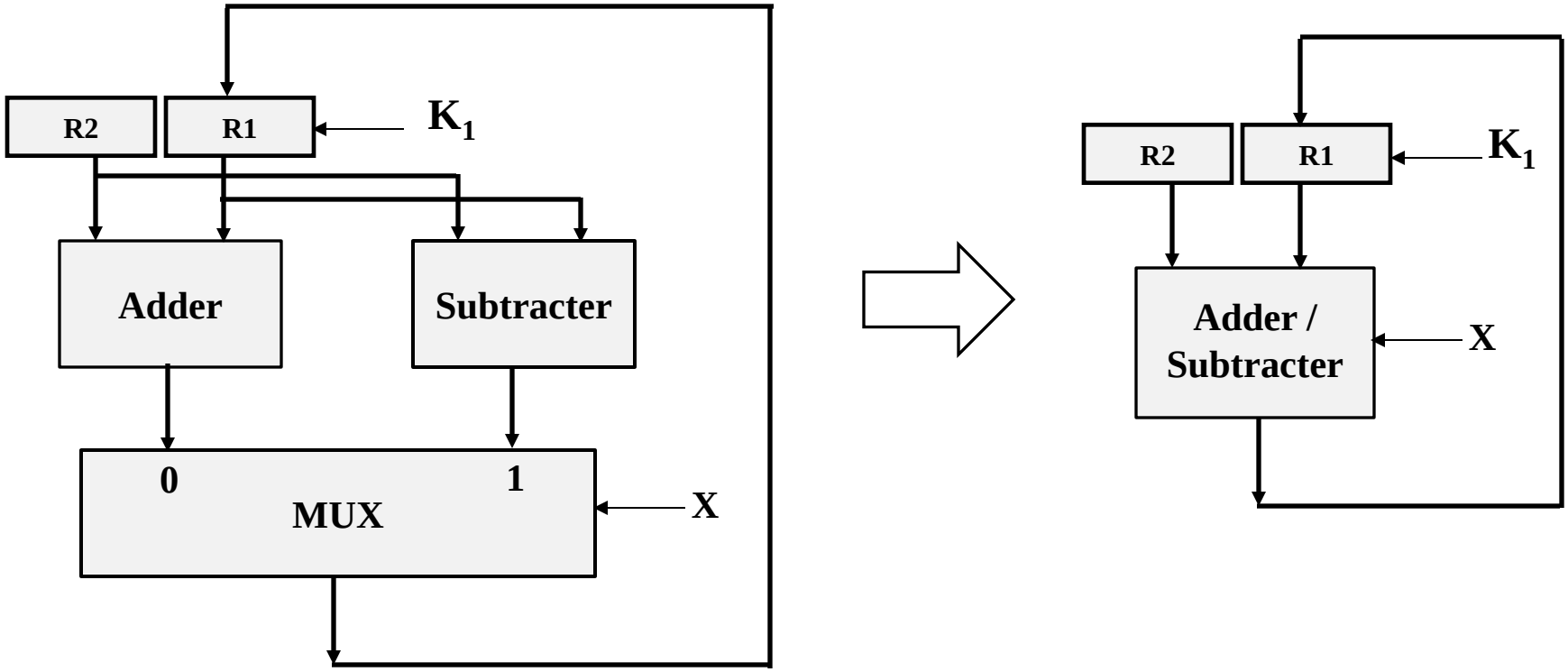
  $X\ K_1$: R1 ← R2 + $\overline{R1}$ + 1

- Variable K_1 enables the add or subtract operation.
- If X =0, then $\overline{X}$ =1 so $\overline{X}\ K_1$ = 1, activating the addition of R1 and R2.
- If X = 1, then $X\ K_1$ = 1, activating the addition of R2 and the two's complement of R1 (subtract).

Arithmetic Microoperations

$\overline{X}\ K_1$: $R1 \leftarrow R2 + R1$

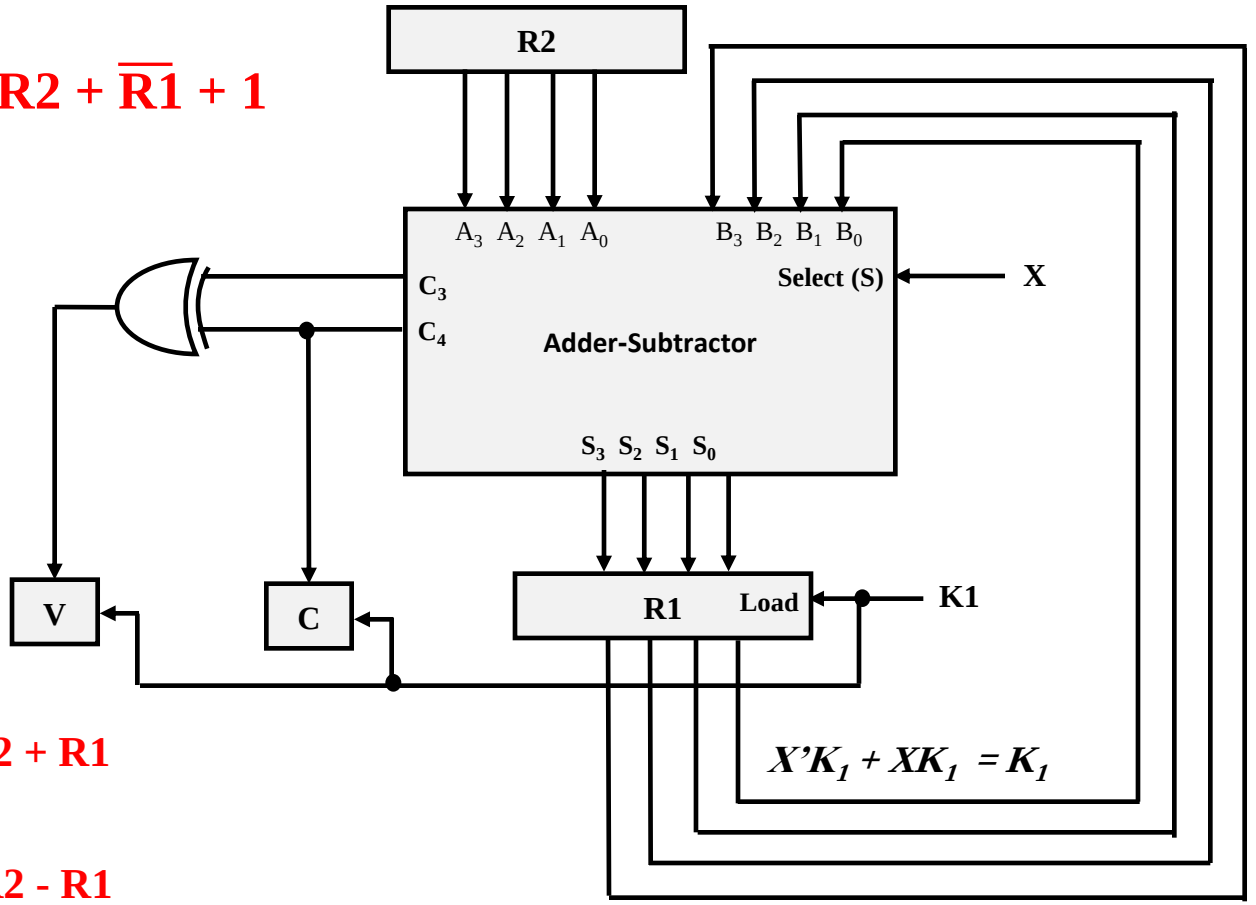
$X\ K_1$: $R1 \leftarrow R2 + \overline{R1} + 1$



Arithmetic Microoperations

$\overline{X}\,K_1$: R1 ← R2 + R1

$X\,K_1$: R1 ← R2 + $\overline{R1}$ + 1



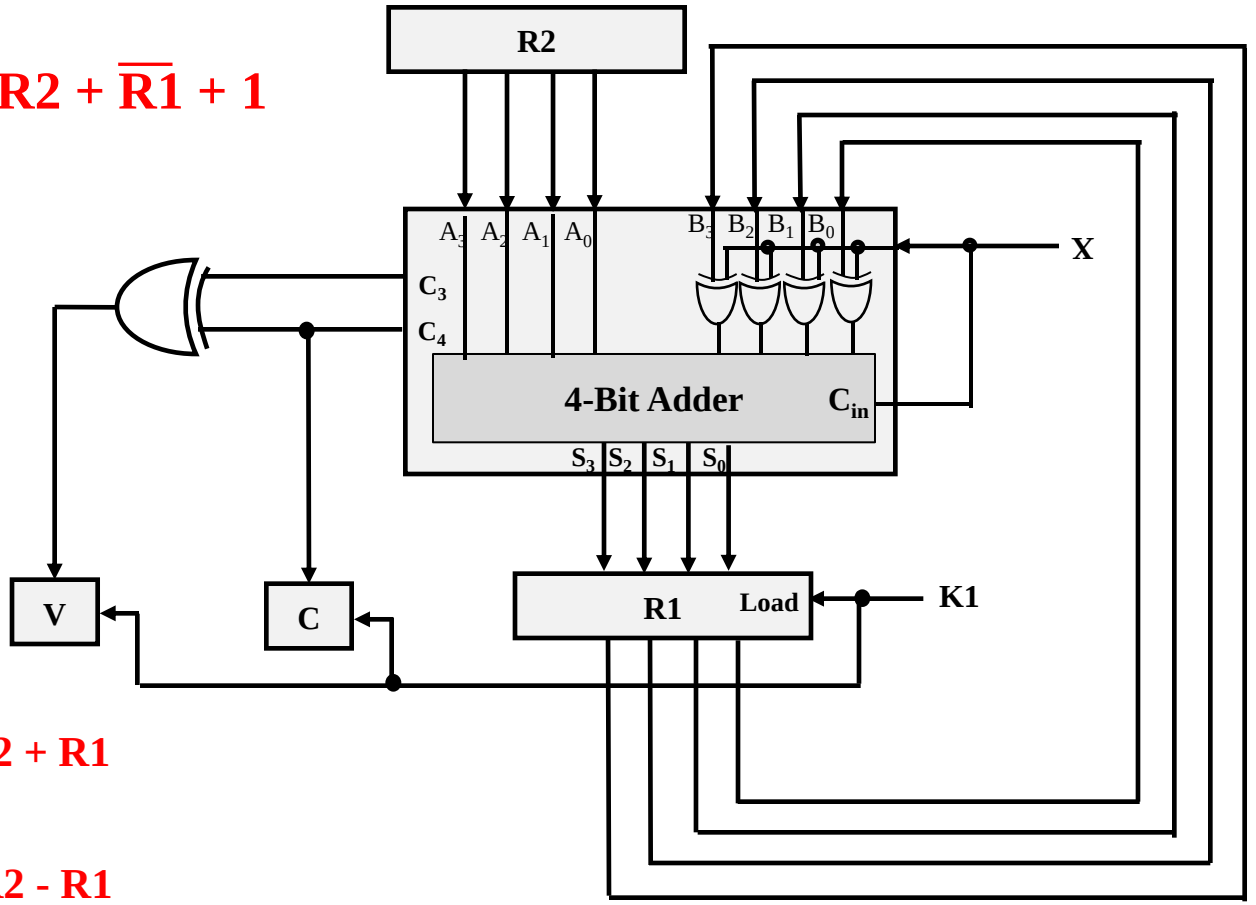
X= 0 ▶ R1 ← R2 + R1

1 ▶ R1 ← R2 - R1

Arithmetic Microoperations

$\overline{X}\,K_1$: R1 ← R2 + R1

$X\,K_1$: R1 ← R2 + $\overline{R1}$ + 1



X= 0 ► R1 ← R2 + R1

X= 1 ► R1 ← R2 - R1

Two's Complement Arithmetic

$(9)_{10} = (0\ 1001)_2 \rightarrow (-9)_{10} = (1\ 0111)_2$

$(4)_{10} = (0\ 0100)_2 \rightarrow (-4)_{10} = (1\ 1100)_2$

$9 - 4 = 9+(-4) = (0\ 1001)_2 + (1\ 1100)_2$

1 1 0 0 0

0 1001

\+

1 1100

---

0 0101 $\rightarrow (5)_{10}$

Two's Complement Arithmetic

$(9)_{10} = (0\ 1001)_2 \rightarrow (-9)_{10} = (1\ 0111)_2$

$(4)_{10} = (0\ 0100)_2 \rightarrow (-4)_{10} = (1\ 1100)_2$

$4 - 9 = 4 + (-9) = (0\ 0100)_2 + (1\ 0111)_2$

0 0 1 0 0

$0\ 0100$

$+$

$1\ 0111$

$1\ 1011 \rightarrow (-5)_{10}$

Two's Complement Arithmetic

$(9)_{10} = (0\ 1001)_2 \rightarrow (-9)_{10} = (1\ 0111)_2$

$(8)_{10} = (0\ 1000)_2 \rightarrow (-8)_{10} = (1\ 1000)_2$

$9 + 8 = (0\ 1001)_2 + (0\ 1000)_2$

0 1 0 0 0

0 1001

\+

0 1000

---

1 0001 → OVERFLOW

$-2^4 \leq \text{Number} \leq 2^4 - 1 \rightarrow -16 \leq \text{Number} \leq +15$

Two's Complement Arithmetic

$(9)_{10} = (0\ 1001)_2 \rightarrow (-9)_{10} = (1\ 0111)_2$

$(8)_{10} = (0\ 1000)_2 \rightarrow (-8)_{10} = (1\ 1000)_2$

$(-9) + (-8) = (1\ 0111)_2 + (1\ 1000)_2$

1 0 0 0 0

1 0111

\+

1 1000

---

0 1111 → OVERFLOW

$-2^4 \leq \text{Number} \leq 2^4 - 1 \rightarrow -16 \leq \text{Number} \leq +15$

Arithmetic Microoperations

- **From Table 7-3:**

Symbolic Designation	Description
$R0 \leftarrow R1 + R2$	Addition
$R0 \leftarrow \overline{R1}$	Ones Complement
$R0 \leftarrow \overline{R1} + 1$	Two's Complement
$R0 \leftarrow R2 + \overline{R1} + 1$	R2 minus R1 (2's Comp)
$R1 \leftarrow R1 + 1$	Increment (count up)
$R1 \leftarrow R1 - 1$	Decrement (count down)

- **Note that any register may be specified for source 1, source 2, or destination.**
- **These simple microoperations operate on the whole word**

Logical Microoperations

Symbolic Designation	Description
$R0 \leftarrow \overline{R1}$	Bitwise NOT
$R0 \leftarrow R1 \vee R2$	Bitwise OR (sets bits)
$R0 \leftarrow R1 \wedge R2$	Bitwise AND (clears bits)
$R0 \leftarrow R1 \oplus R2$	Bitwise EXOR (complements bits)

Logical Microoperations

Example:

- **Let R1 = 10101010,**
  and R2 = 11110000
- **Then after the operation, R0 becomes:**

R0	Operation
01010101	$R0 \leftarrow \overline{R1}$
11111010	$R0 \leftarrow R1 \vee R2$
10100000	$R0 \leftarrow R1 \wedge R2$
01011010	$R0 \leftarrow R1 \oplus R2$

Shift Microoperations

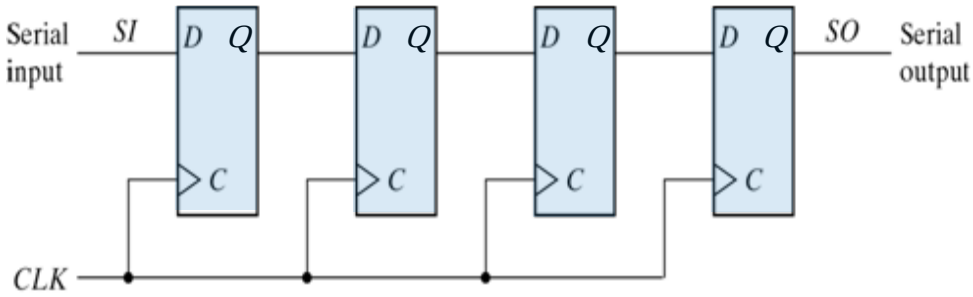
- From Table 7-5:
- Let R2 = 11001001
- Then after the operation, R1 becomes:

Symbolic Designation	Description
R1 ← sl R2	Shift Left
R1 ← sr R2	Shift Right

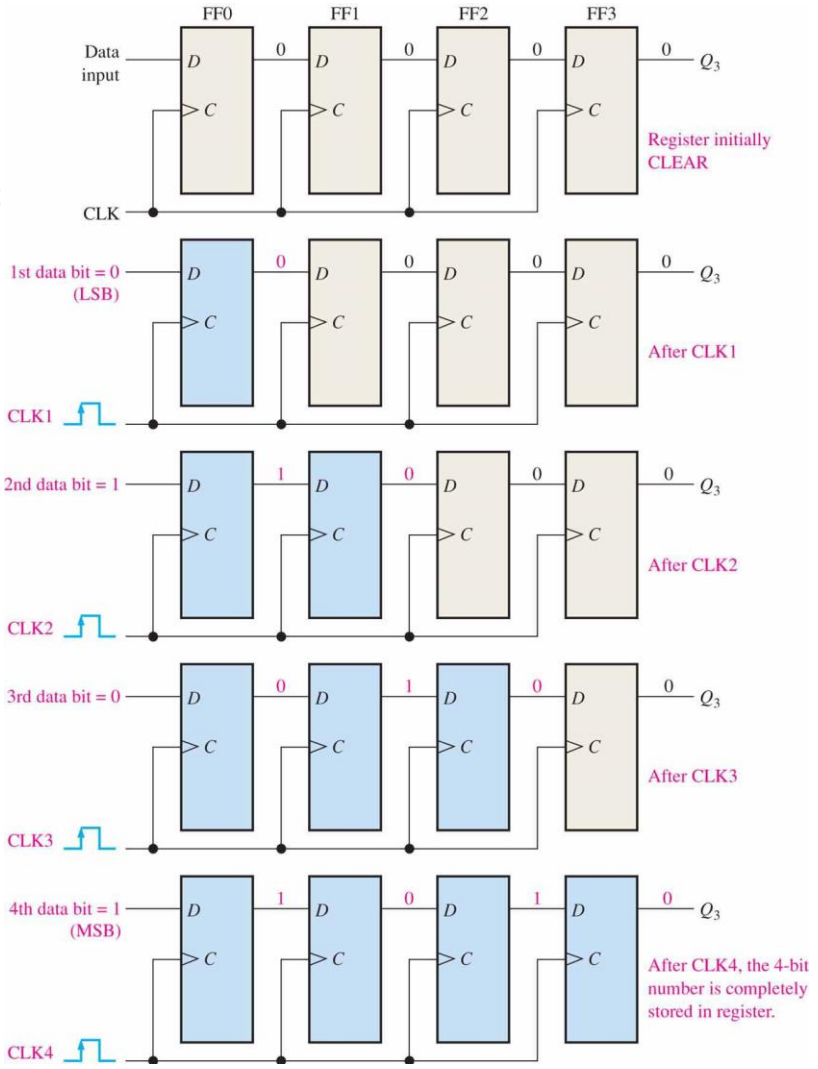
R1	Operation
10010010	R1 ← sl R2
01100100	R1 ← sr R2

- Note: These shifts "zero fill". Sometimes a separate flip-flop is used to provide the data shifted in, or to "catch" the data shifted out.
- Other shifts are possible (rotates, arithmetic) (see Chapter 10).

Shift Registers

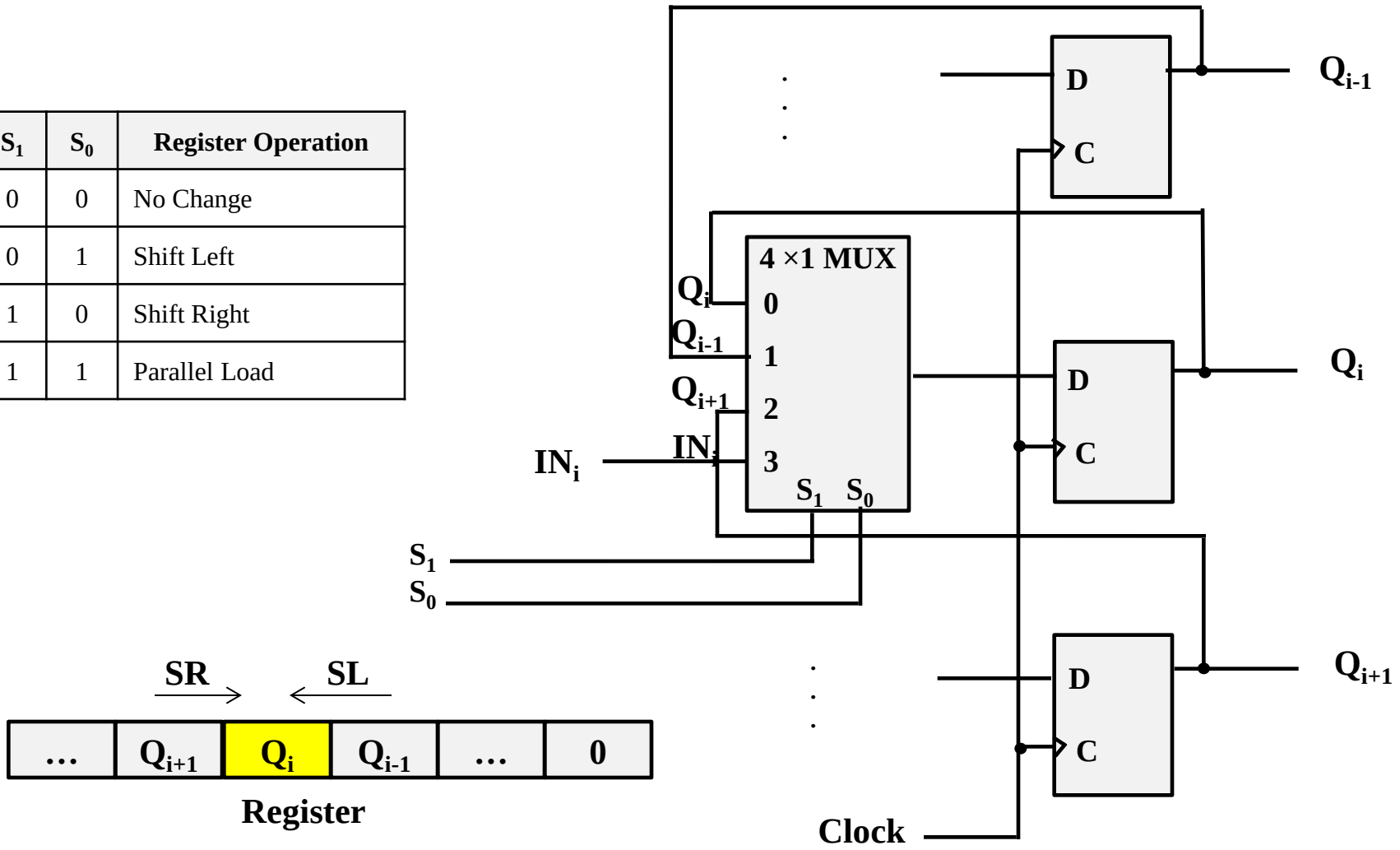


4-Bit Shift Register



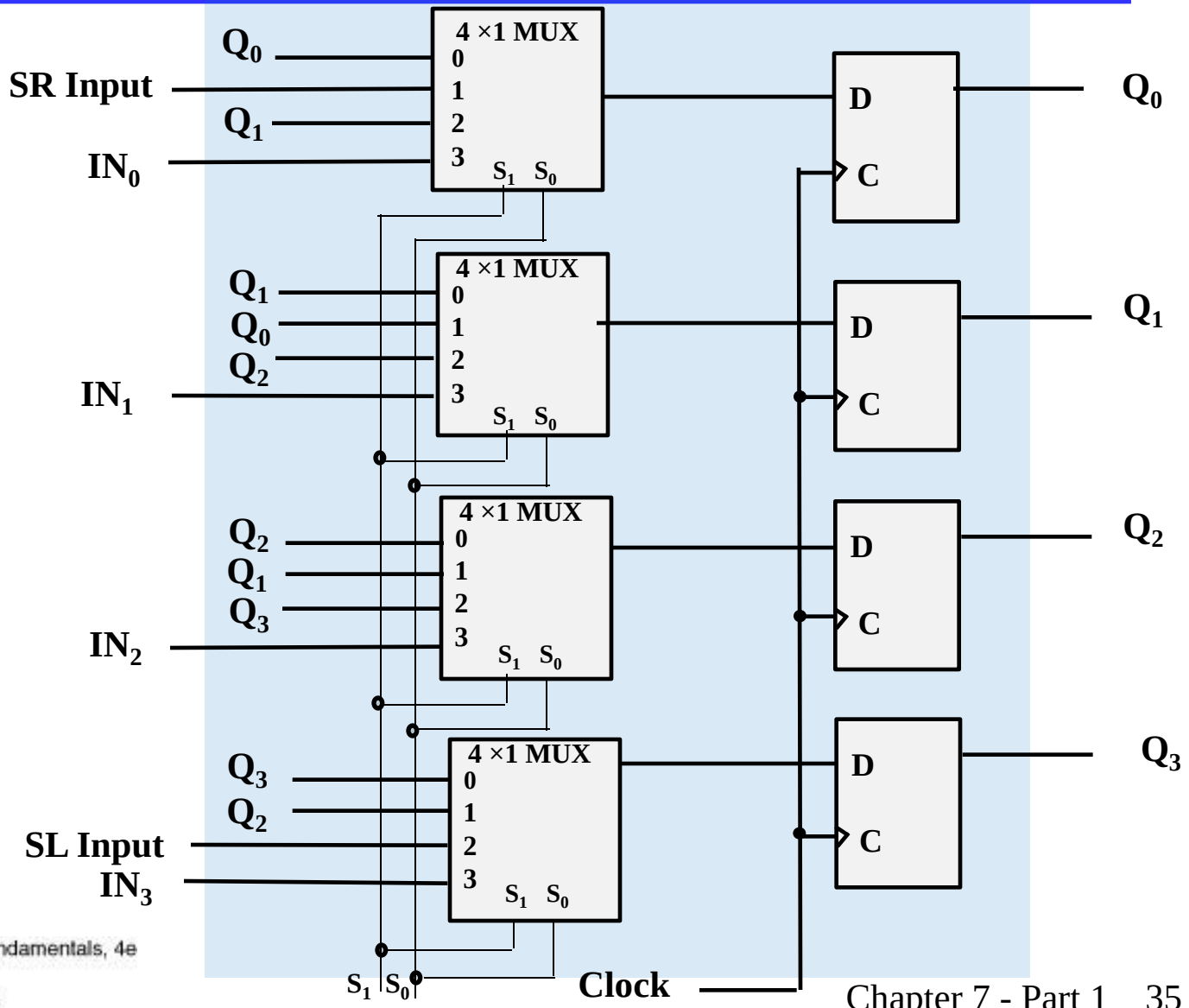
Bidirectional Shift Register with Parallel Load

S_1	S_0	Register Operation
0	0	No Change
0	1	Shift Left
1	0	Shift Right
1	1	Parallel Load



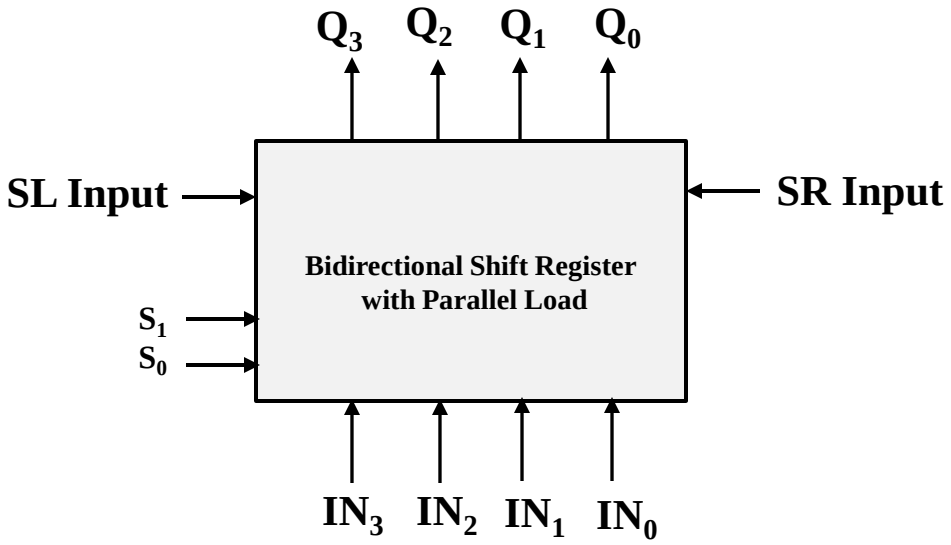
Bidirectional Shift Register with Parallel Load

S_1	S_0	Register Operation
0	0	No Change
0	1	Shift Left
1	0	Shift Right
1	1	Parallel Load



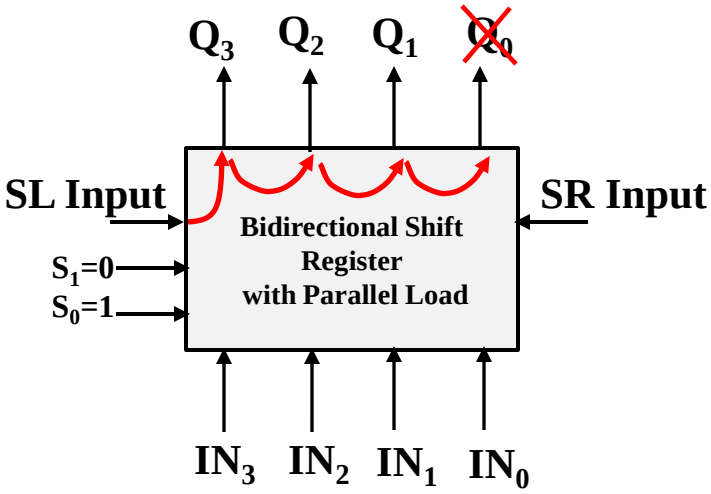
Bidirectional Shift Register with Parallel Load

S_1	S_0	Register Operation
0	0	No Change
0	1	Shift Left
1	0	Shift Right
1	1	Parallel Load

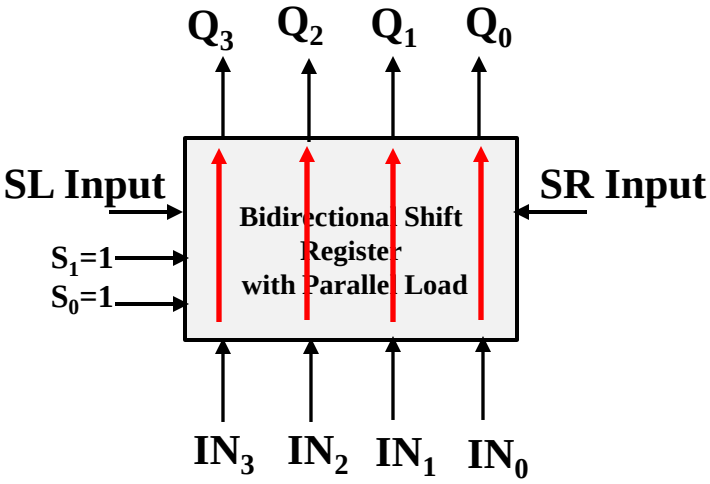


Bidirectional Shift Register with Parallel Load

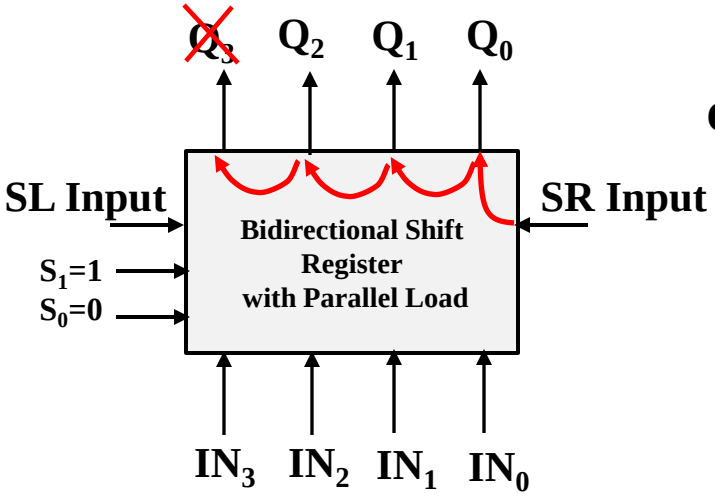
S_1	S_0	Register Operation
0	0	No Change
0	1	Shift Left
1	0	Shift Right
1	1	Parallel Load



Case 1: Shift Right



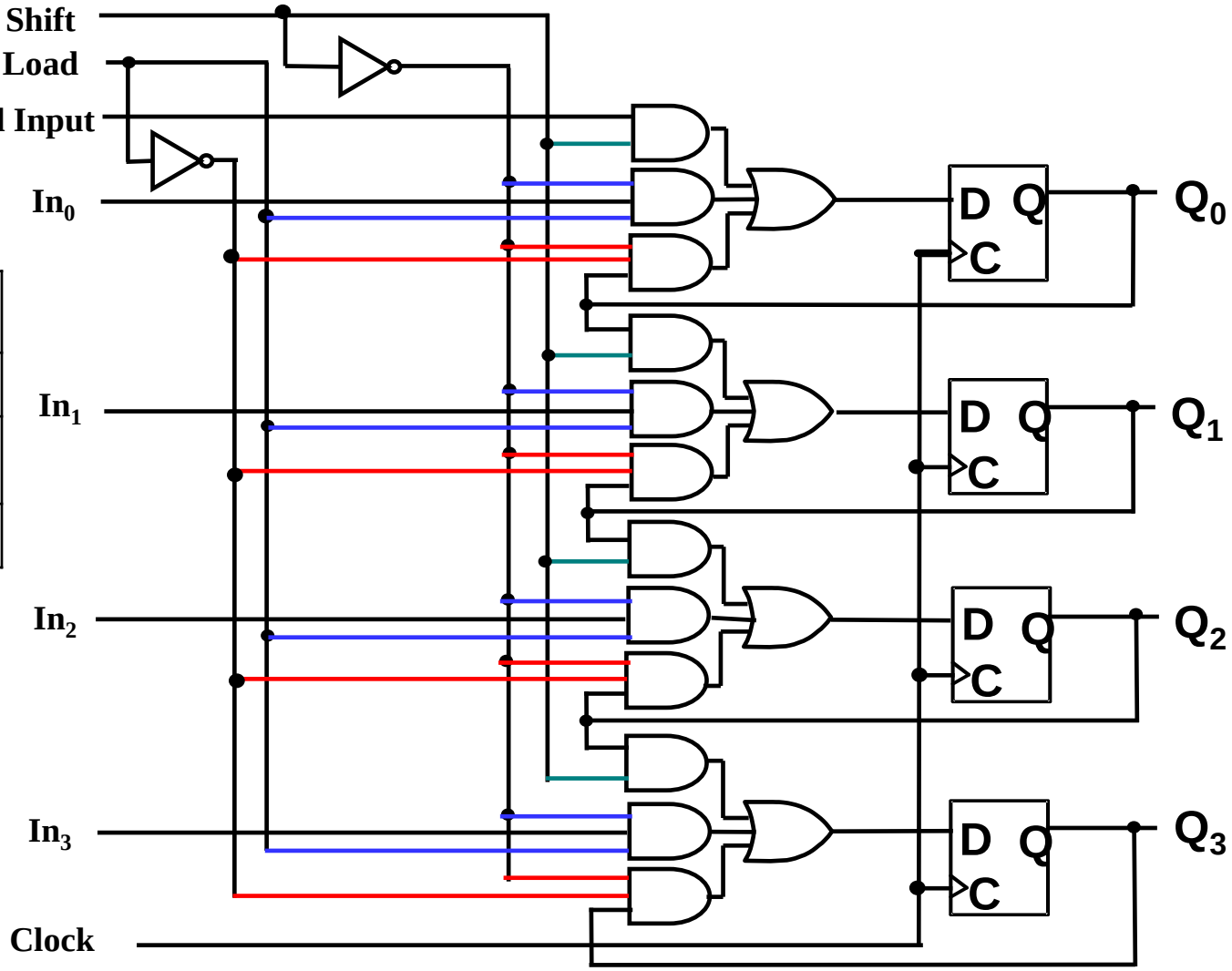
Case 3: Parallel Load



Case 2: Shift Left

Shift Register with Parallel Load

Shift	Load	Register Operation
0	0	No Change
0	1	Load Parallel Data
1	X	Shift Left



Register Transfer Structures

- **Multiplexer-Based Transfers - Multiple inputs are selected by a multiplexer dedicated to the register**
- **Bus-Based Transfers - Multiple inputs are selected by a shared multiplexer driving a bus that feeds inputs to multiple registers**
- **Three-State Bus - Multiple inputs are selected by 3-state drivers with outputs connected to a bus that feeds multiple registers**
- **Other Transfer Structures - Use multiple multiplexers, multiple buses, and combinations of all the above**

Multiplexer-Based Transfers

- **Multiplexers connected to register inputs produce flexible transfer structures (Note: Clocks are omitted for clarity)**

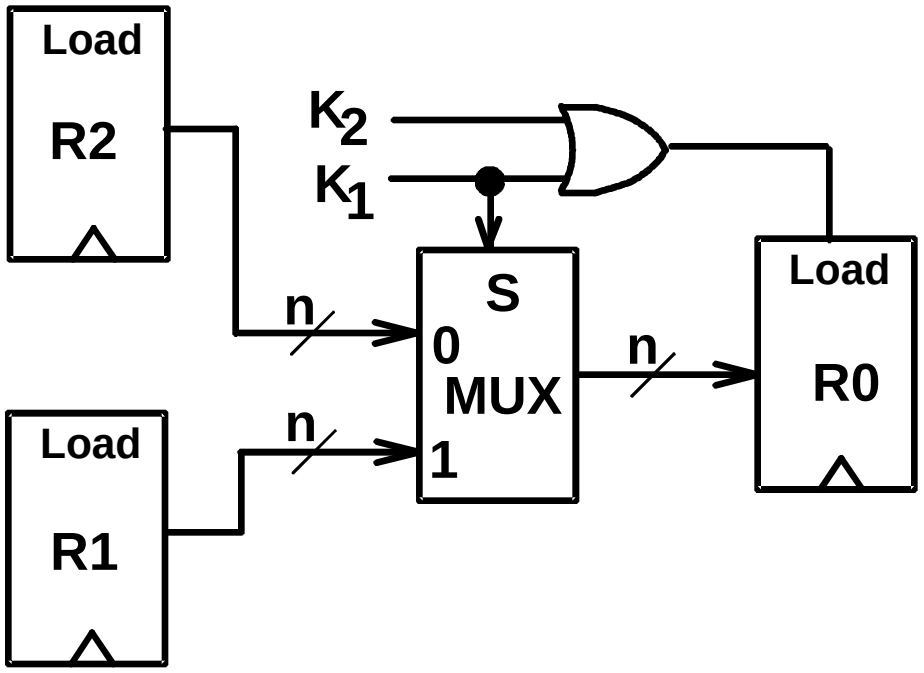
Example:

- **The transfers are:**

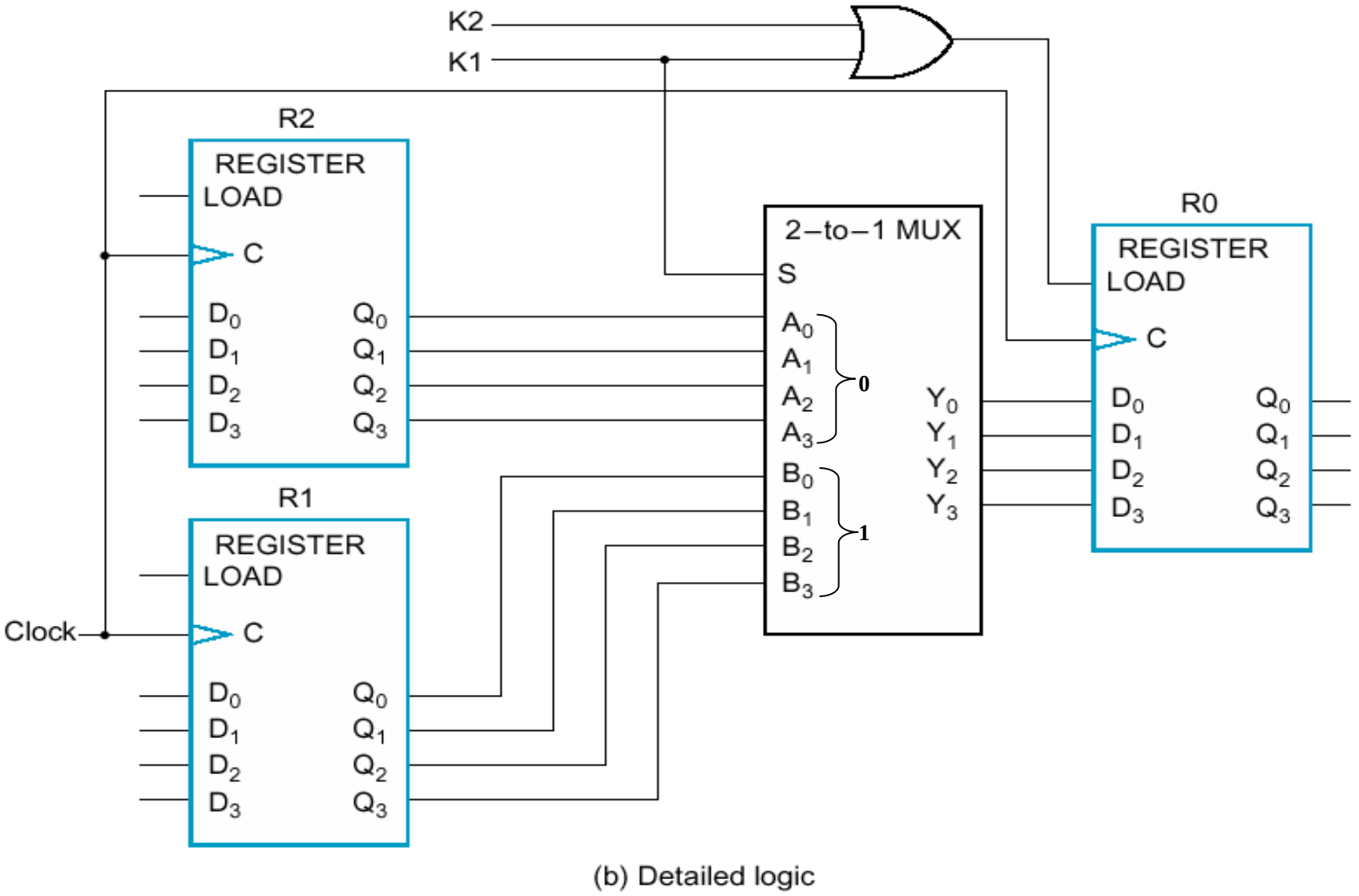
$\overline{K_1}$: R0 ← R1

$K_1 \cdot$ K2: R0 ← R2

$K_1 + \overline{K_1} K_2 = K_1 + K_2$

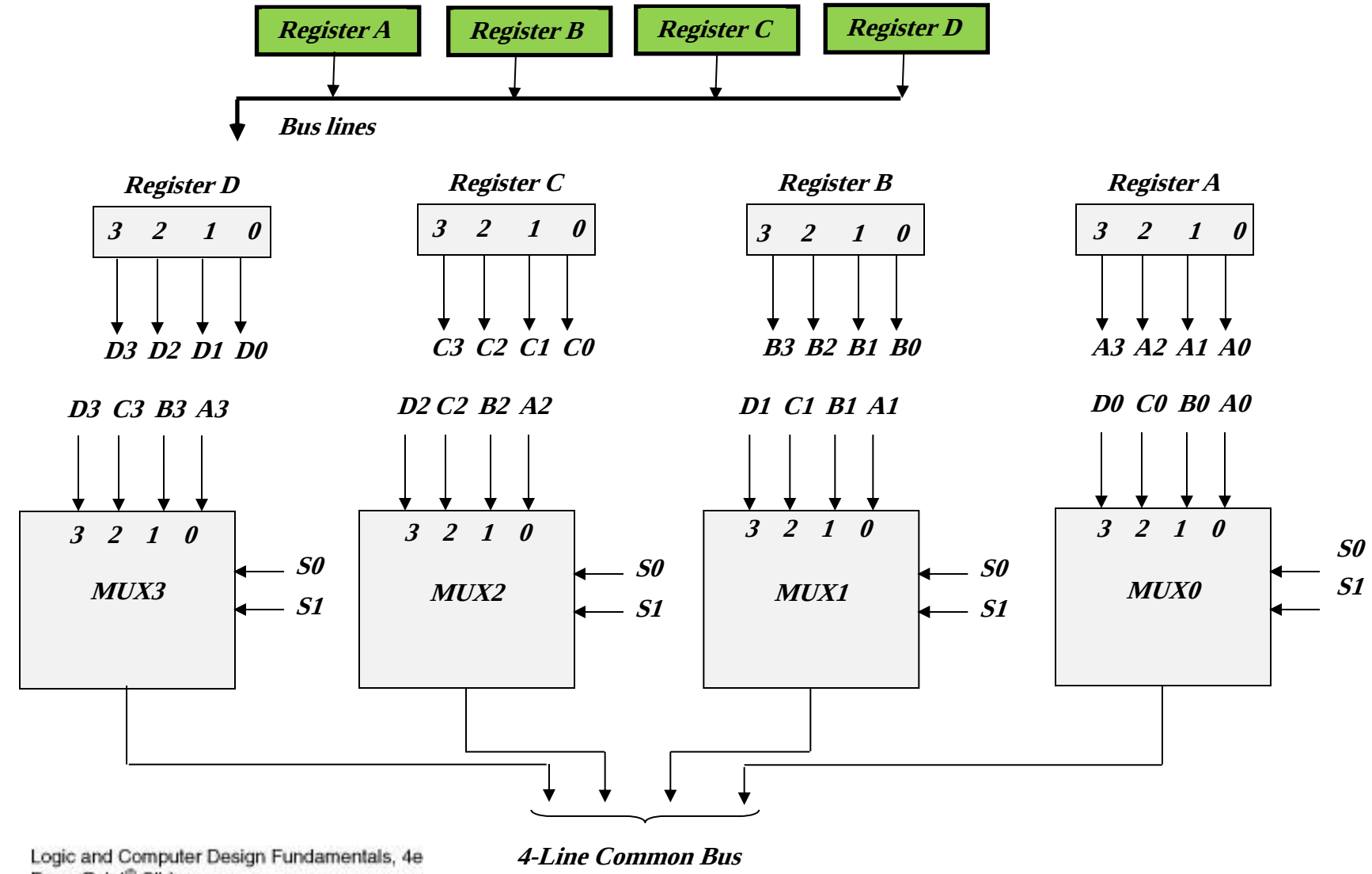


Multiplexer-Based Transfer Example: Two 4-bit registers



(b) Detailed logic

Bus Transfers

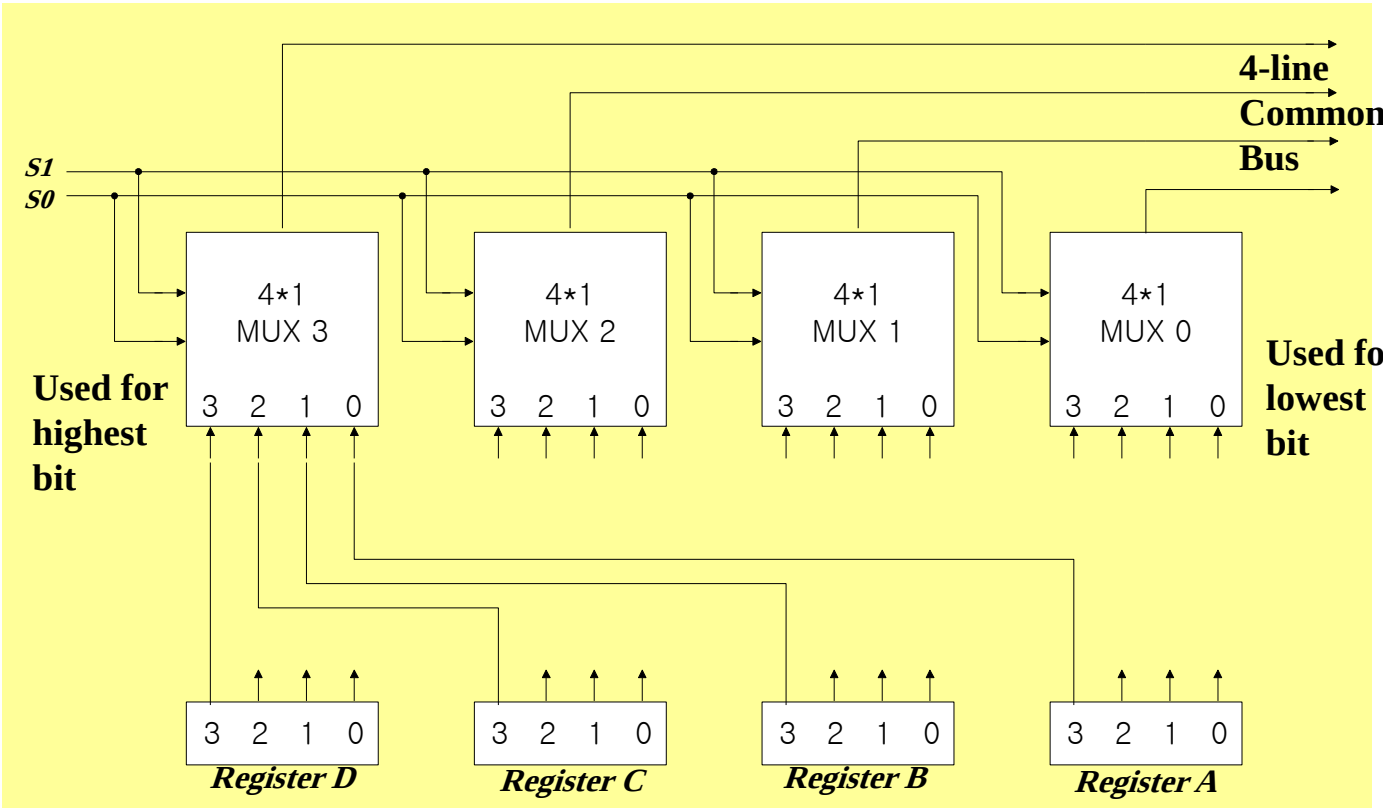


Bus Transfer

Example:

For register R0 to R3 in a 4 bit system

S1	S0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

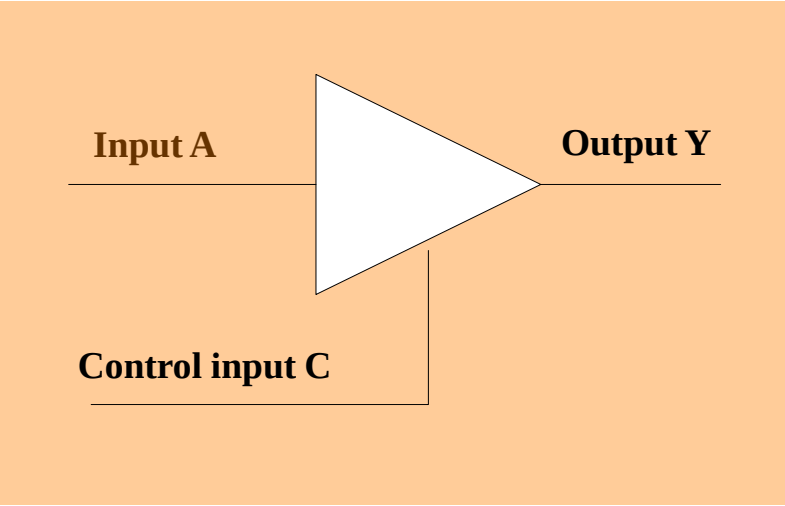


Bus Transfer

- **For register R0 to R63 in a 16 bit system:**
  - **What is the MUX size we use?** **64 × 1 MUX**
  - **How many MUX we need?** **16 MUXs**
  - **How many select bit?** **6 bits**

Tri-State Buffers

- **Tri-state buffer gate:**
  - **When control input =1 : The output is nabled (output Y = input A)**
  - **When control input =0 : The output is disabled (output Y = high-impedance)**

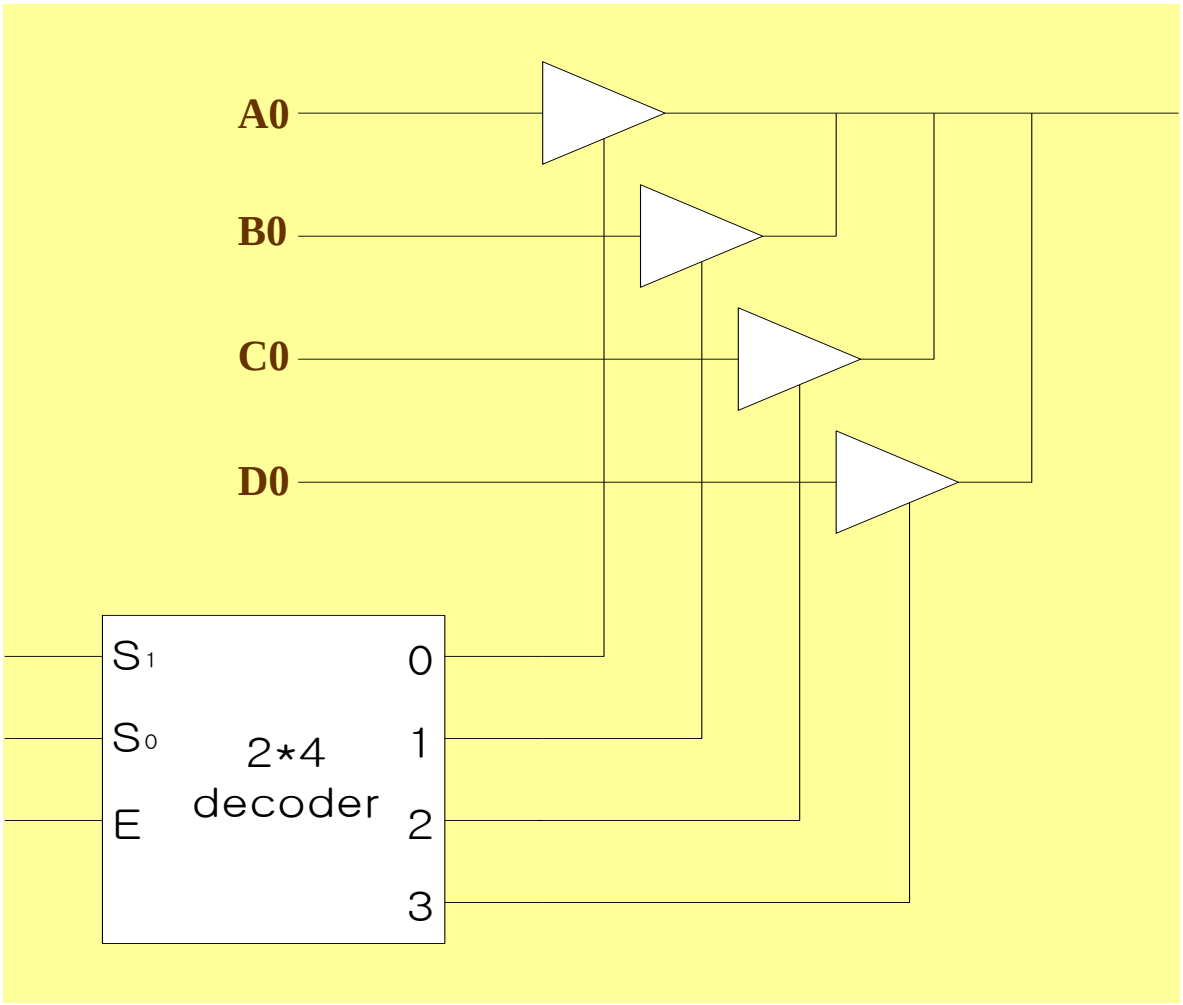


If C=1, Output Y = A

If C=0, Output = High-impedance

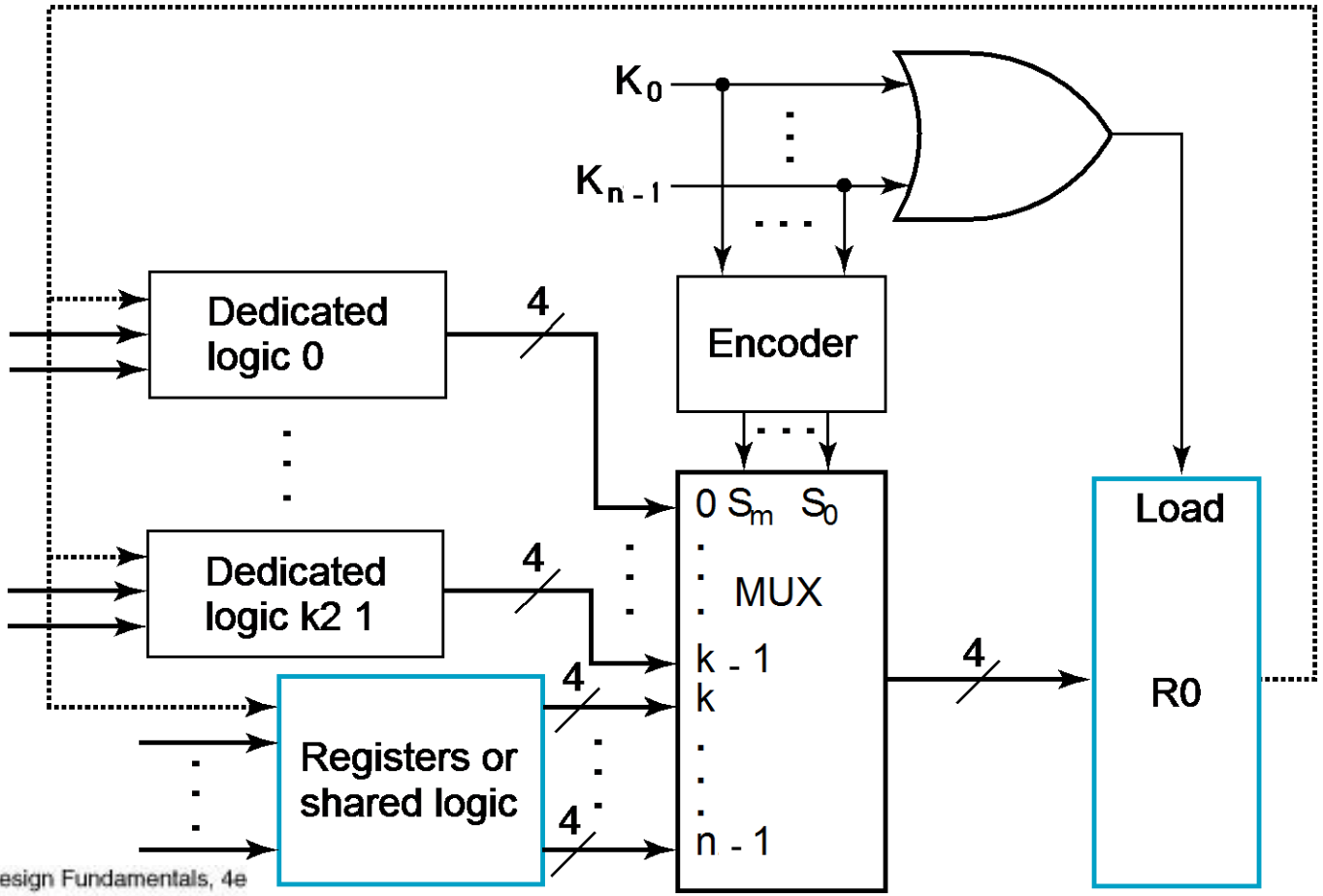
Bus system with tri-state buffer

S1	S0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D



Multiplexer Approach

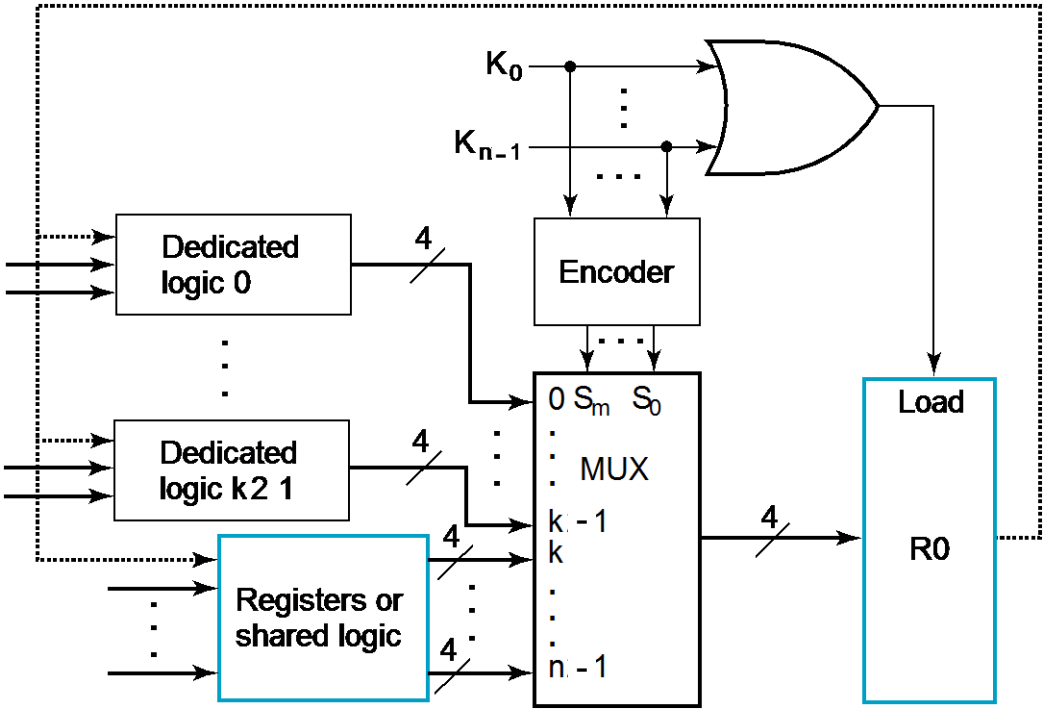
- **Uses an n-input multiplexer with a variety of transfer sources and functions**



Multiplexer Approach

- **Load enable by OR of control signals K_0, K_1, … K_{n-1}**
  - Assumes no load for 00…0
- **Use:**
  - **Encoder + Multiplexer (shown) or**
  - **n x 2 AND-OR**

to select sources and/or transfer functions

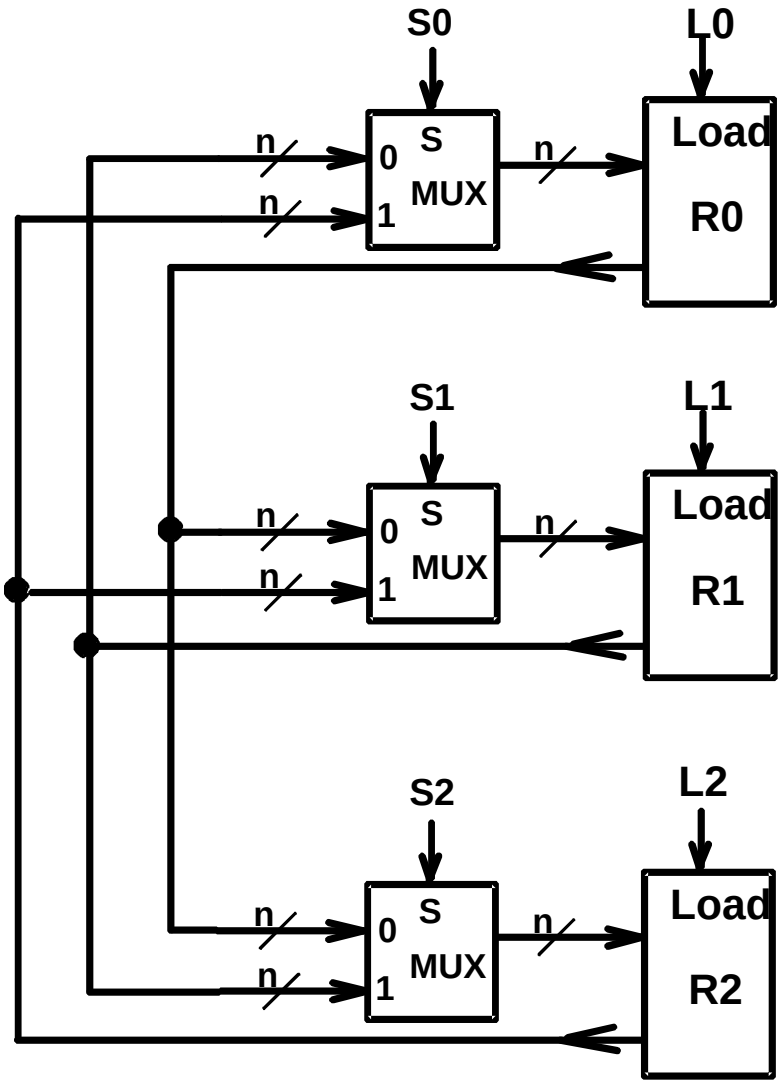


Multiplexer and Bus-Based Transfers for Multiple Registers

- **Multiplexer dedicated to each register**
- **Shared transfer paths for registers**
  - **A shared transfer object is a called a *bus* (Plural: *buses*)**
- **Bus implementation using:**
  - **multiplexers**
  - **three-state nodes and drivers**
- **In most cases, the number of bits is the length of the receiving register**

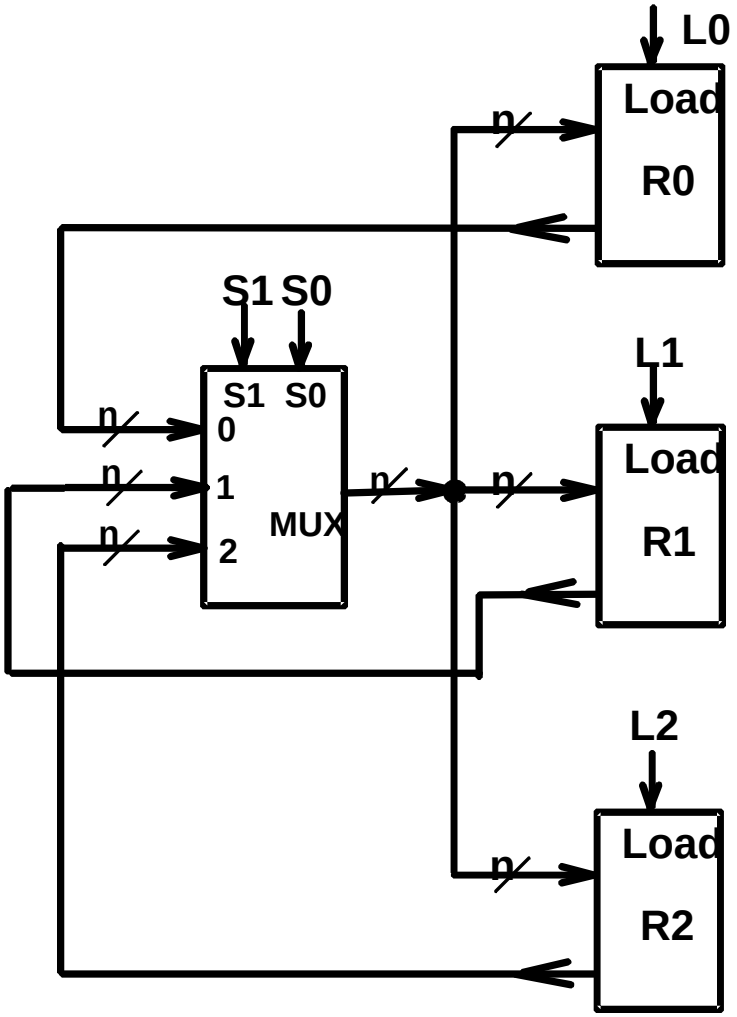
Dedicated MUX-Based Transfers

- Multiplexer connected to each register input produces a very flexible transfer structure =>
- Characterize the simultaneous transfers possible with this structure.



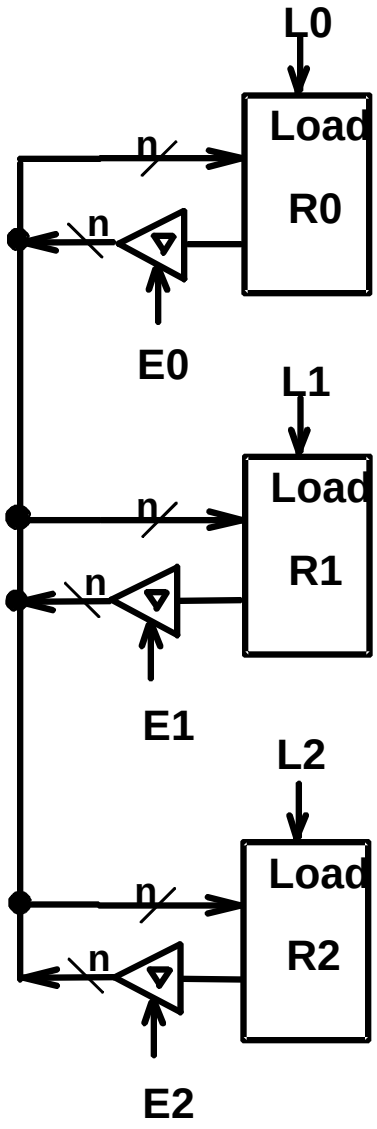
Multiplexer Bus

- **A single bus driven by a multiplexer lowers cost, but limits the available transfers.**
- **Characterize the simultaneous transfers possible with this structure.**
- **Characterize the cost savings compared to dedicated multiplexers**



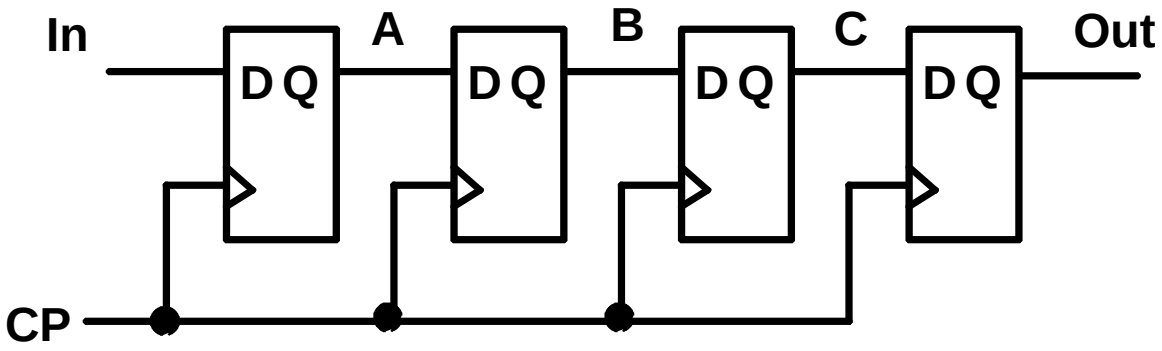
Three-State Bus

- **The 3-input MUX can be replaced by a 3-state node (bus) and 3-state buffers.**
- **Cost is further reduced, but transfers are limited**
- **Characterize the simultaneous transfers possible with this structure.**
- **Characterize the cost savings and compare**
- **Other advantages?**



Shift Registers

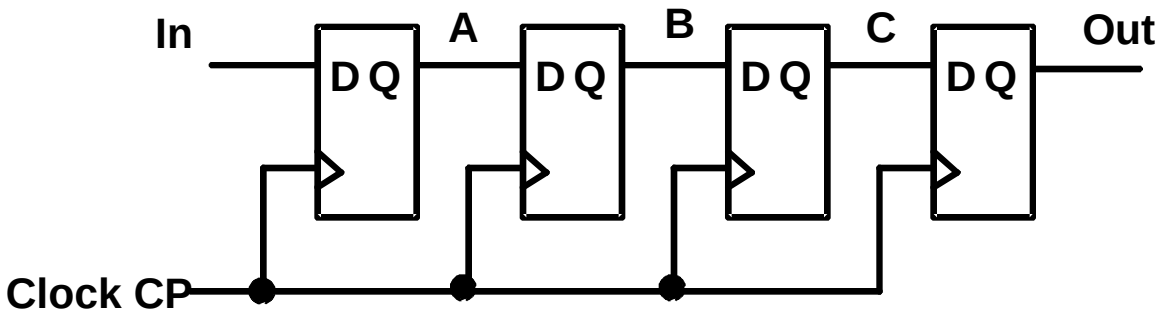
- **Shift Registers move data laterally within the register toward its MSB or LSB position**
- **In the simplest case, the shift register is simply a set of D flip-flops connected in a row like this:**



- **Data input, In, is called a *serial input* or the *shift right input.***
- **Data output, Out, is often called the *serial output.***
- **The vector (A, B, C, Out) is called the *parallel output.***

Shift Registers (continued)

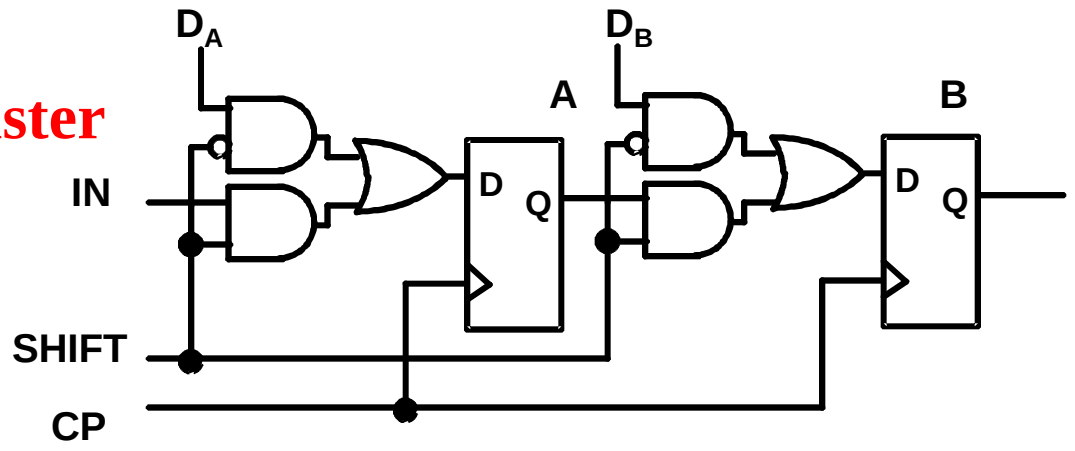
- The behavior of the serial shift register is given in the listing on the lower right
- T0 is the register state just before the first clock pulse occurs
- T1 is after the first pulse and before the second.
- Initially unknown states are denoted by "?"



CP	In	A	B	C	Out
T0	0	?	?	?	?
T1	1	0	?	?	?
T2	1	1	0	?	?
T3	0	1	1	0	?
T4	1	0	1	1	0
T5	1	1	0	1	1
T6	1	1	1	0	1

Parallel Load Shift Registers

- **By adding a mux between each shift register stage, data can be shifted or loaded**



- **If SHIFT is low, A and B are replaced by the data on D_A and D_B lines, else data shifts right on each clock.**
- **By adding more bits, we can make *n*-bit parallel load shift registers.**

Note:

- **A parallel load shift register with an added "hold" operation that stores data unchanged is given in Figure 7-10 of the text.**

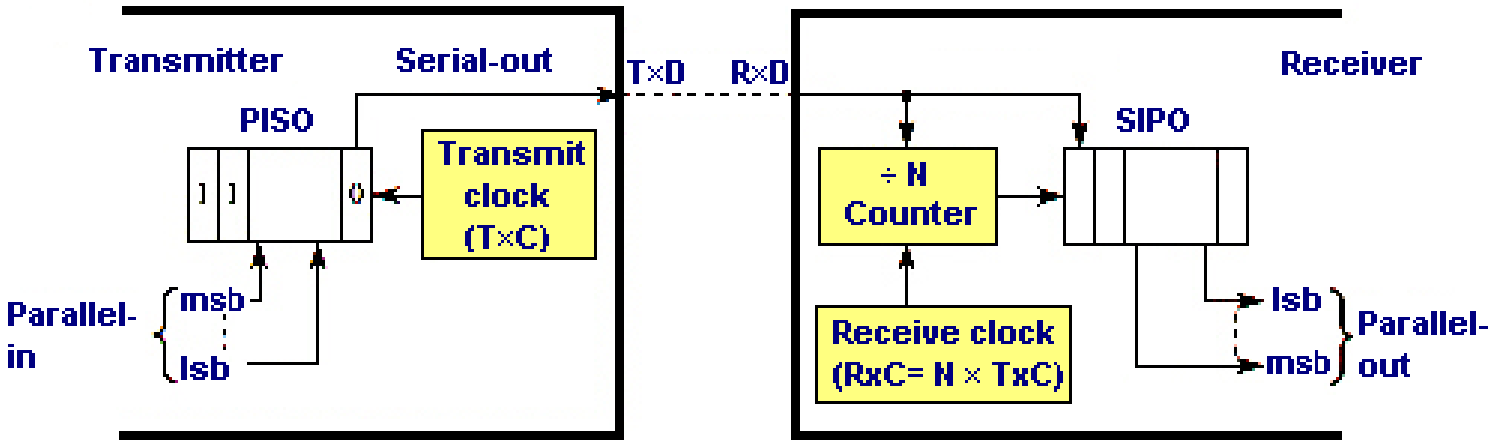
Shift Registers with Additional Functions

- **By placing a 4-input multiplexer in front of each D flip-flop in a shift register, we can implement a circuit with shifts right, shifts left, parallel load, hold.**
- **Shift registers can also be designed to shift more than a single bit position right or left**
- **Shift registers can be designed to shift a variable number of bit positions specified by a variable called a *shift amount*.**

Serial Transfers and Microoperations

- **Serial Transfers**
  - **Used for "narrow" transfer paths**
  - **Example 1: Telephone or cable line**
    - **Parallel-to-Serial conversion at source**
    - **Serial-to-Parallel conversion at destination**
  - **Example 2: Initialization and Capture of the contents of many flip-flops for test purposes**
    - **Add shift function to all flip-flops and form large shift register**
    - **Use shifting for simultaneous Initialization and Capture operations**
- **Serial microoperations**
  - **Example 1: Addition**
  - **Example 2: Error-Correction for CDs**

Parallel-to-Serial / Serial-to-Parallel

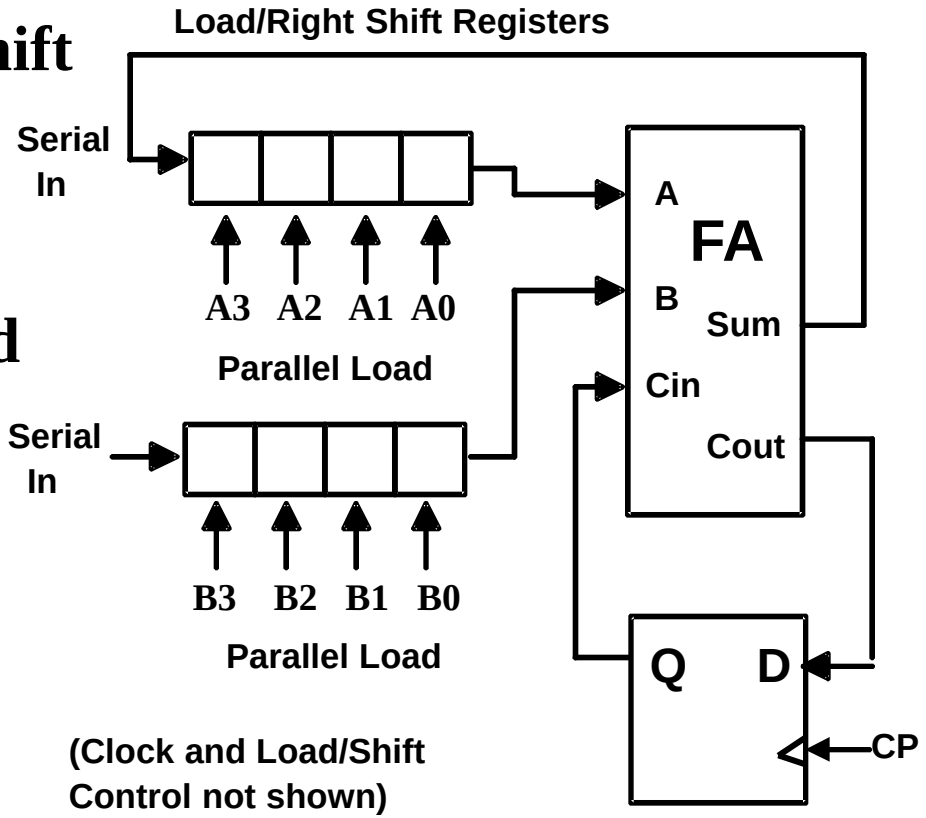


Serial Microoperations

- **By using two shift registers for operands, a full adder, and a flip flop (for the carry), we can add two numbers serially, starting at the least significant bit.**
- **Serial addition is a low cost way to add large numbers of operands, since a "tree" of full adder cells can be made to any depth, and each new level doubles the number of operands.**
- **Other operations can be performed serially as well, such as parity generation/checking or more complex error-check codes.**
- **Shifting a binary number left is equivalent to multiplying by 2.**
- **Shifting a binary number right is equivalent to dividing by 2.**

Serial Adder

- The circuit shown uses two shift registers for operands A(3:0) and B(3:0).
- A full adder, and one more flip flop (for the carry) is used to compute the sum.
- The result is stored in the A register and the final carry in the flip-flop



- With the operands and the result in shift registers, a tree of full adders can be used to add a large number of operands. Used as a common digital signal processing technique.

Terms of Use

- **All (or portions) of this material © 2008 by Pearson Education, Inc.**
- **Permission is given to incorporate this material or adaptations thereof into classroom presentations and handouts to instructors in courses adopting the latest edition of Logic and Computer Design Fundamentals as the course textbook.**
- **These materials or adaptations thereof are not to be sold or otherwise offered for consideration.**
- **This Terms of Use slide or page is to be included within the original materials or any adaptations thereof.**