CSC 220: Computer Organization

Unit 8
Registers and RTL

Prepared by:
Md Saiful Islam, PhD

Department of Computer Science
College of Computer and Information Sciences

Overview

- Registers Construction
  - Basic Registers
  - Shift Registers
- Bus Construction
- Register Transfer Language
  - Micro operations
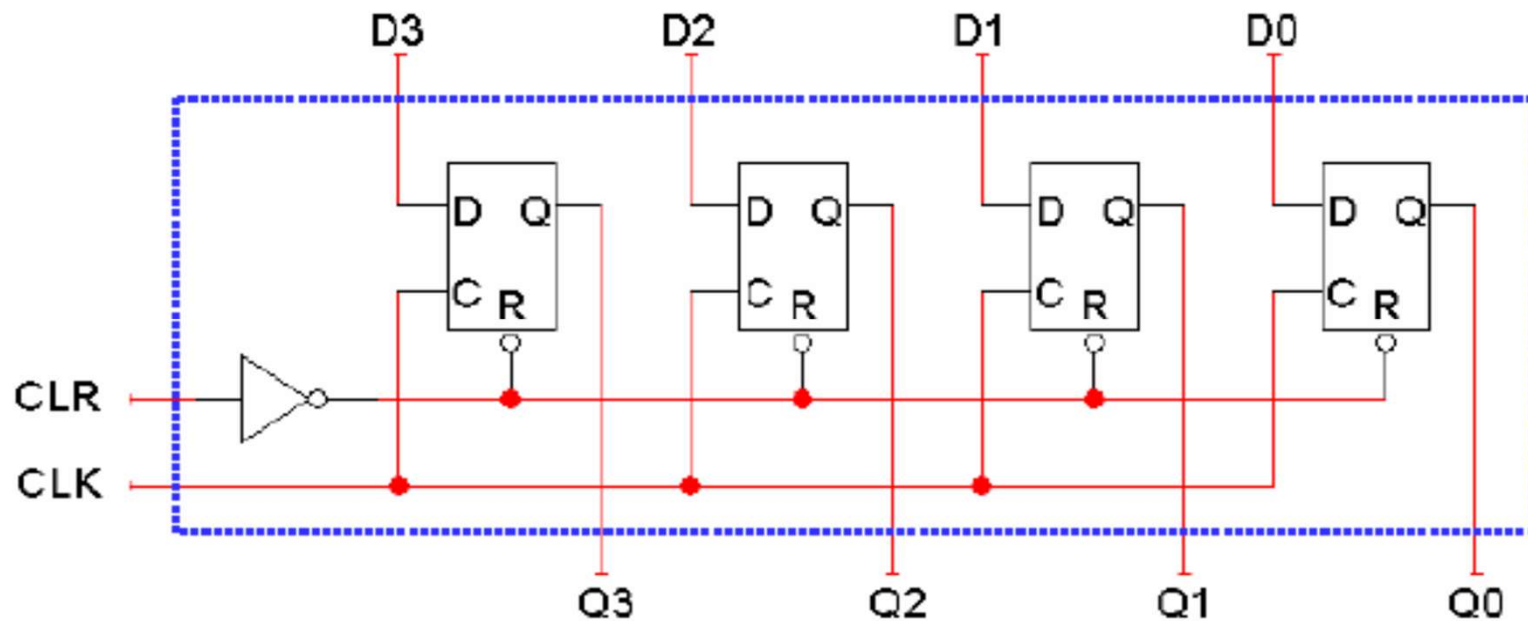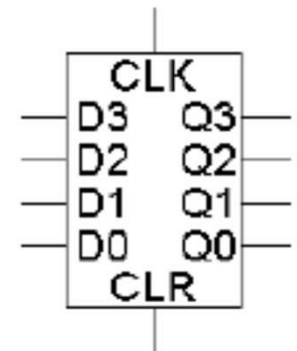
Registers

- Flip-flops are limited because they can store only one bit.
  - We had to use two flip-flops for most of our examples so far.
  - Most computers work with integers and single-precision floating-point numbers that are 32-bits long.
- A register is an extension of a flip-flop that can store multiple bits.
- Registers are commonly used as temporary storage in a processor.
  - They are faster and more convenient than main memory.
  - More registers can help speed up complex calculations.
- Later we'll learn more about how registers are used in processors, and some of the differences between registers and random-access memories or RAM.

A basic register

- Basic registers are easy to build. We can store multiple bits just by putting a bunch of flip-flops together!
- A 4-bit register from LogicWorks, Reg-4, is on the right, and its internal implementation is below.
  - This register uses D flip-flops, so it's easy to store data without worrying about flip-flop input equations.
 - All the flip-flops share a common CLK and CLR signal.

A Register with parallel load

Adding another operation

- The input D3-D0 is copied to the output Q3-Q0 on *every* clock cycle.
- How can we store the current value for more than one cycle?
- Let's try to add a load input signal LD to the register.
  - If LD = 0, the register keeps its current contents.
  - If LD = 1, the register stores a new value, taken from inputs D3-D0.

| LD	Q(t+1)
0	Q(t)
1	$D_3\text{-}D_0$

D3   D2   D1   D0

CLR
LD
CLK

Q3   Q2   Q1   Q0
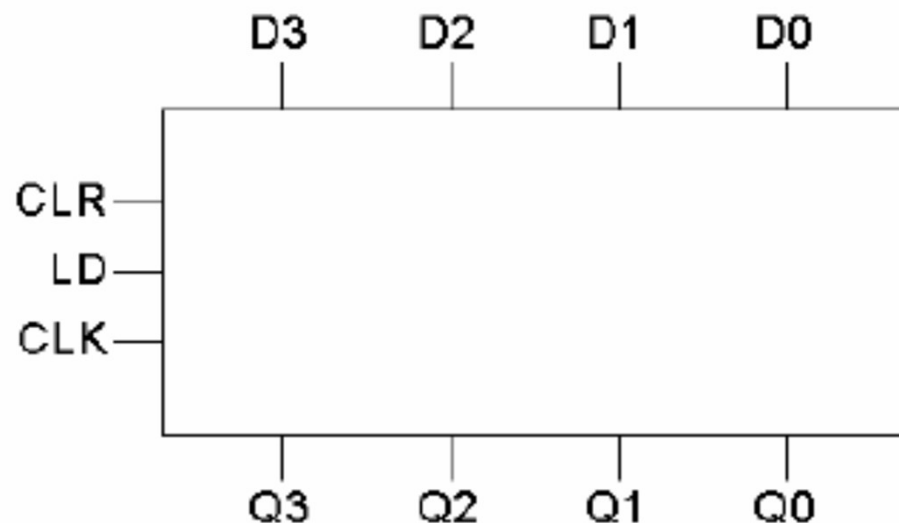
A better parallel load

- Another idea is to modify the flip-flop D inputs and not the clock signal.
  - When LD = 0 the flip-flop inputs are Q3-Q0, so each flip-flop keeps its current value.
  - When LD = 1 the flip-flop inputs are D3-D0, so this new value is loaded into the register.
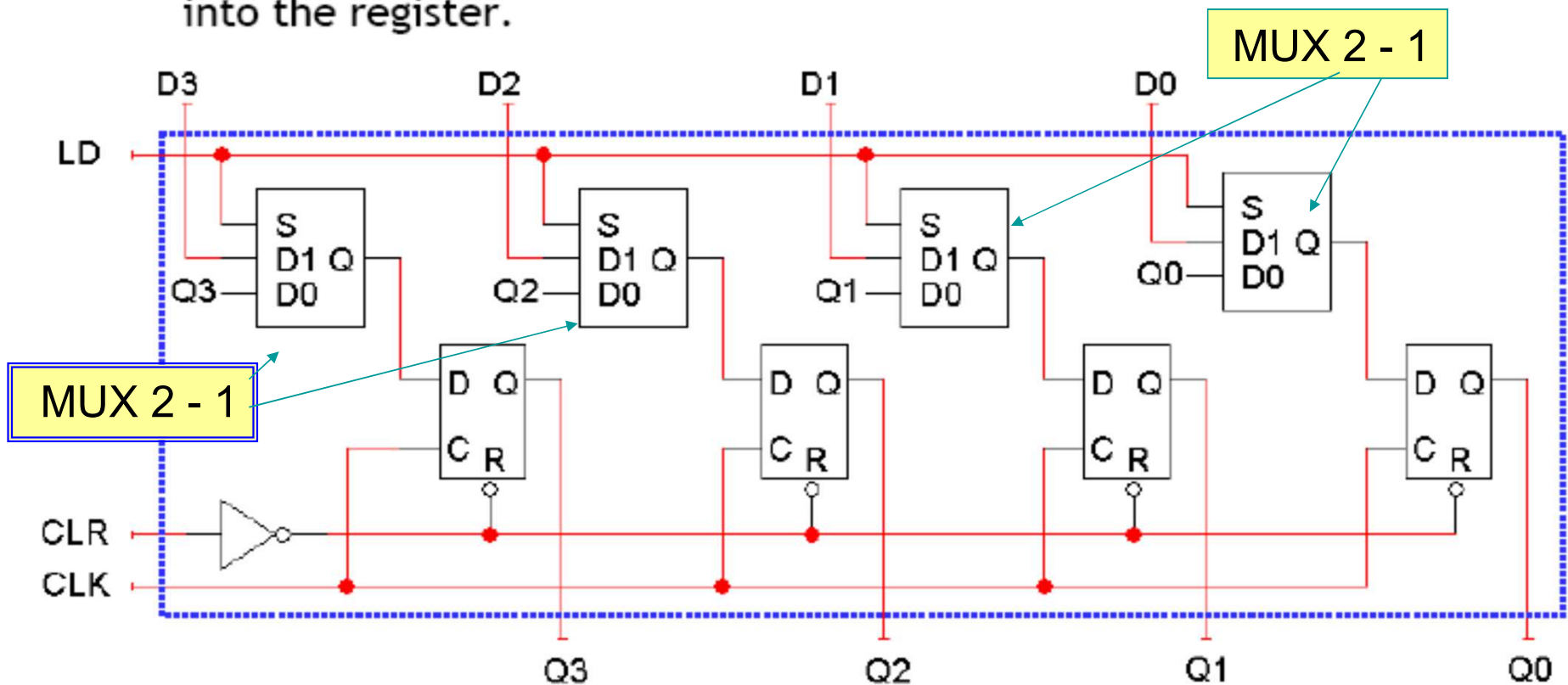
Shift registers (Left Shift)

- A shift register "shifts" its output once every clock cycle. SI is an input that supplies a new bit to shift "into" the register.
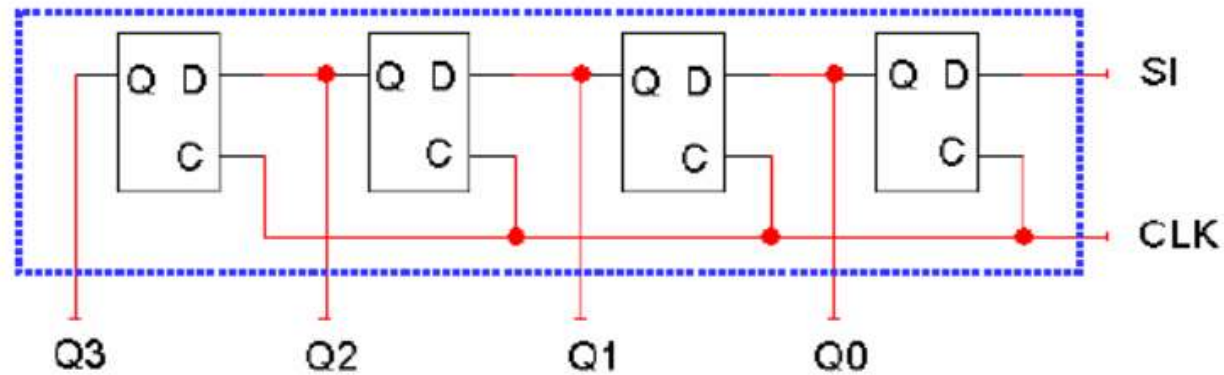
$$Q0(t+1) = SI$$
$$Q1(t+1) = Q0(t)$$
$$Q2(t+1) = Q1(t)$$
$$Q3(t+1) = Q2(t)$$

- Here is one example transition.

Present State Q3-Q0	Input SI	Next State Q3-Q0
0110	1	1101

- The current Q3 (0 in this example) will be lost on the next cycle.

7

Shift registers (Right Shift)

- A shift register "shifts" its output once every clock cycle. SI is an input that supplies a new bit to shift "into" the register.
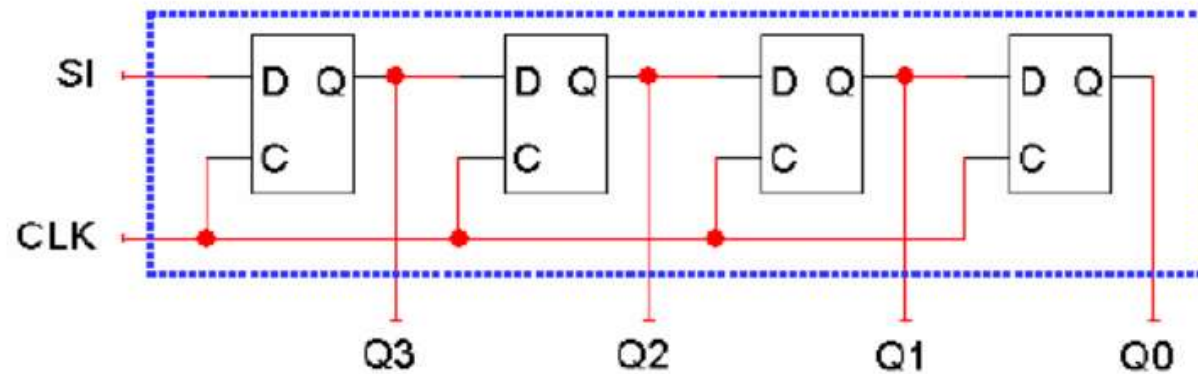
- Here is one example transition.

Present State Q3-Q0	Input SI	Next State Q3-Q0
0110	1	1011

- The current Q0 (0 in this example) will be lost on the next cycle.

8

Shift registers with parallel load

- We can add a parallel load operation, just as we did for regular registers.
  - When LD = 0 the flip-flop inputs will be SIQ3Q2Q1, so the register will shift on the next positive clock edge.
  - When LD = 1, the flip-flop inputs are D0-D3, and a new value is loaded into the register on the next positive clock edge.

Serial

Parallel

MUX 2 - 1

MUX 2 - 1

9
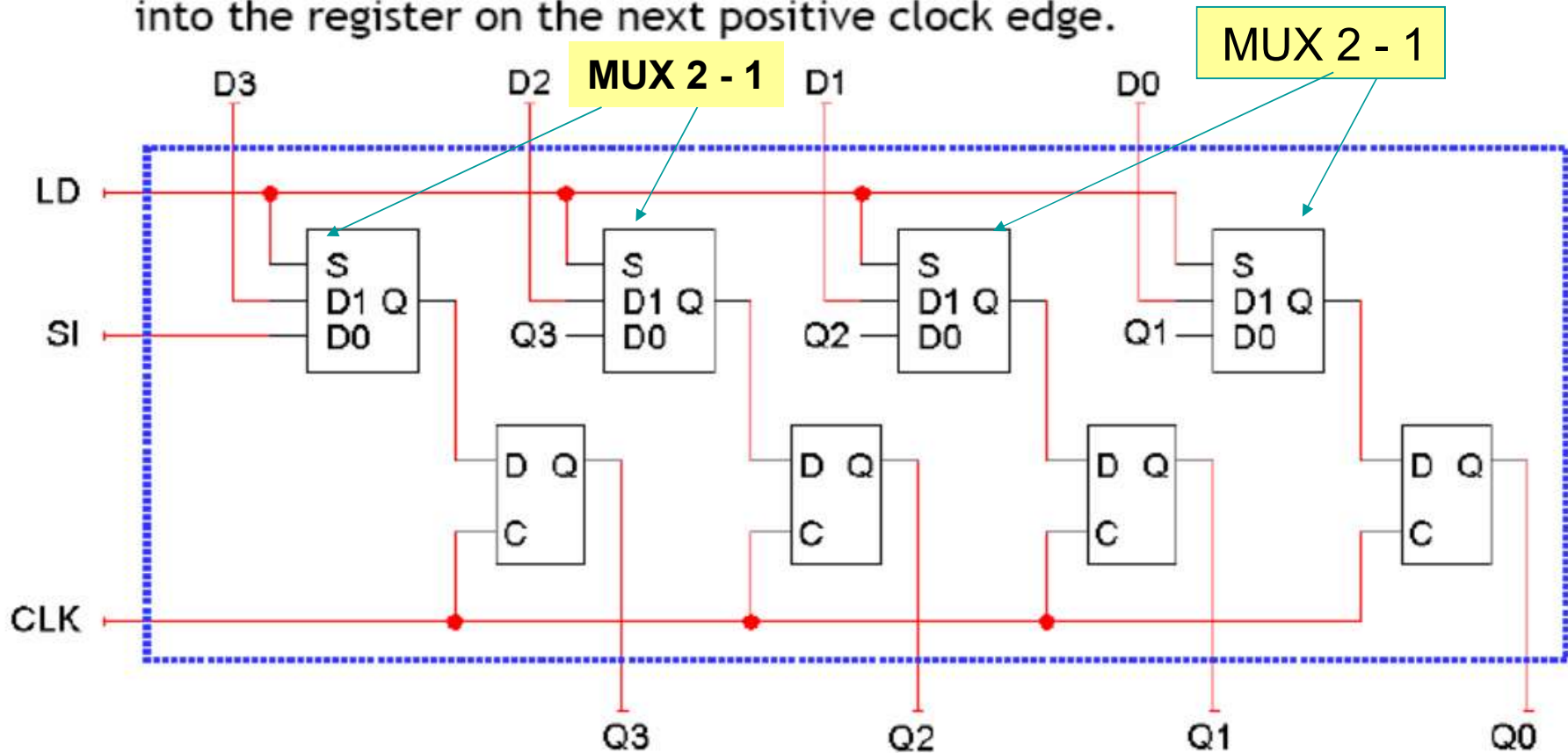
A Bidirectional Shift Register with parallel load

S1	S0	Operation
0	0	No change
0	1	Shift left
1	0	Shift right
1	1	Parallel load

10
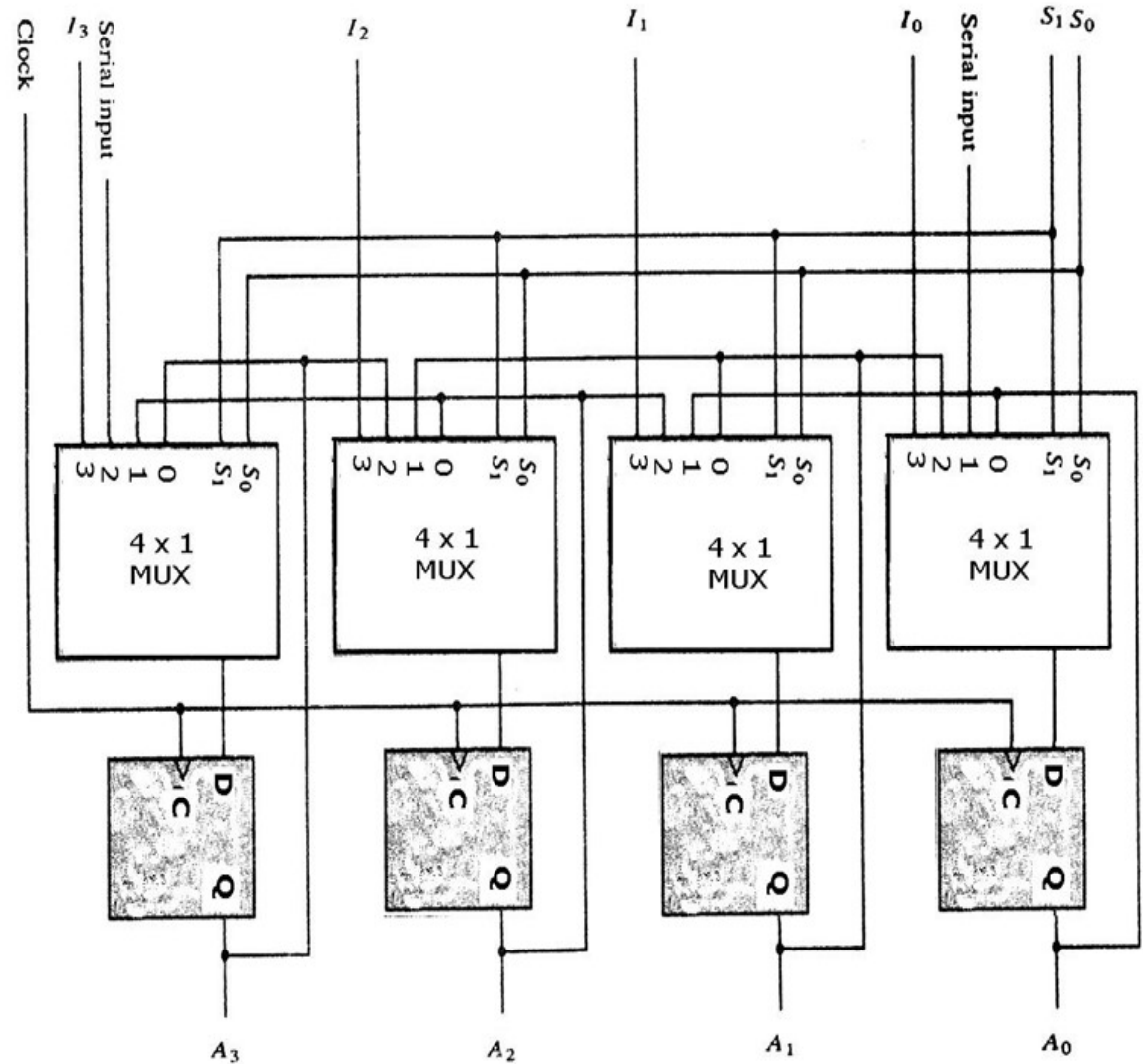
Other types of shift registers

- Logical shifts – Standard shifts like we just saw. In the absence of a SI input, 0 occupies the vacant position.
    - Left: 0110 -> 1100
    - Right: 0110 -> 0011
- Circular shifts (also called ring counters or rotates) – The shifted out bit wraps around to the vacant position.
    - Left: 1001 -> 0011
    - Right: 1001 -> 1100
- Switch-tail ring counter (aka Johnson counter) – Similar to the ring counter, but the serial input is the complement of the serial output.
    - Left: 1001 -> 0010
    - Right: 1001 -> 0100
- Arithmetical shifts – Left shifting is the same as a logical shift. Right shifting however maintains the MSB.
    - Left: 0110 -> 1100
    - Right: 0110 -> 0011;   1011 -> 1101

Bus-based Data transfers

- The computer need several registers
  - Bus is the Path for data transfer among registers
  - A bus consists of a set of parallel data lines
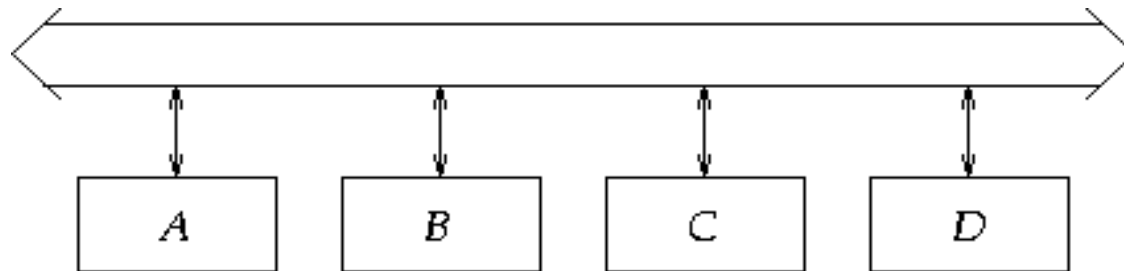
Multiplexer-based bus construction

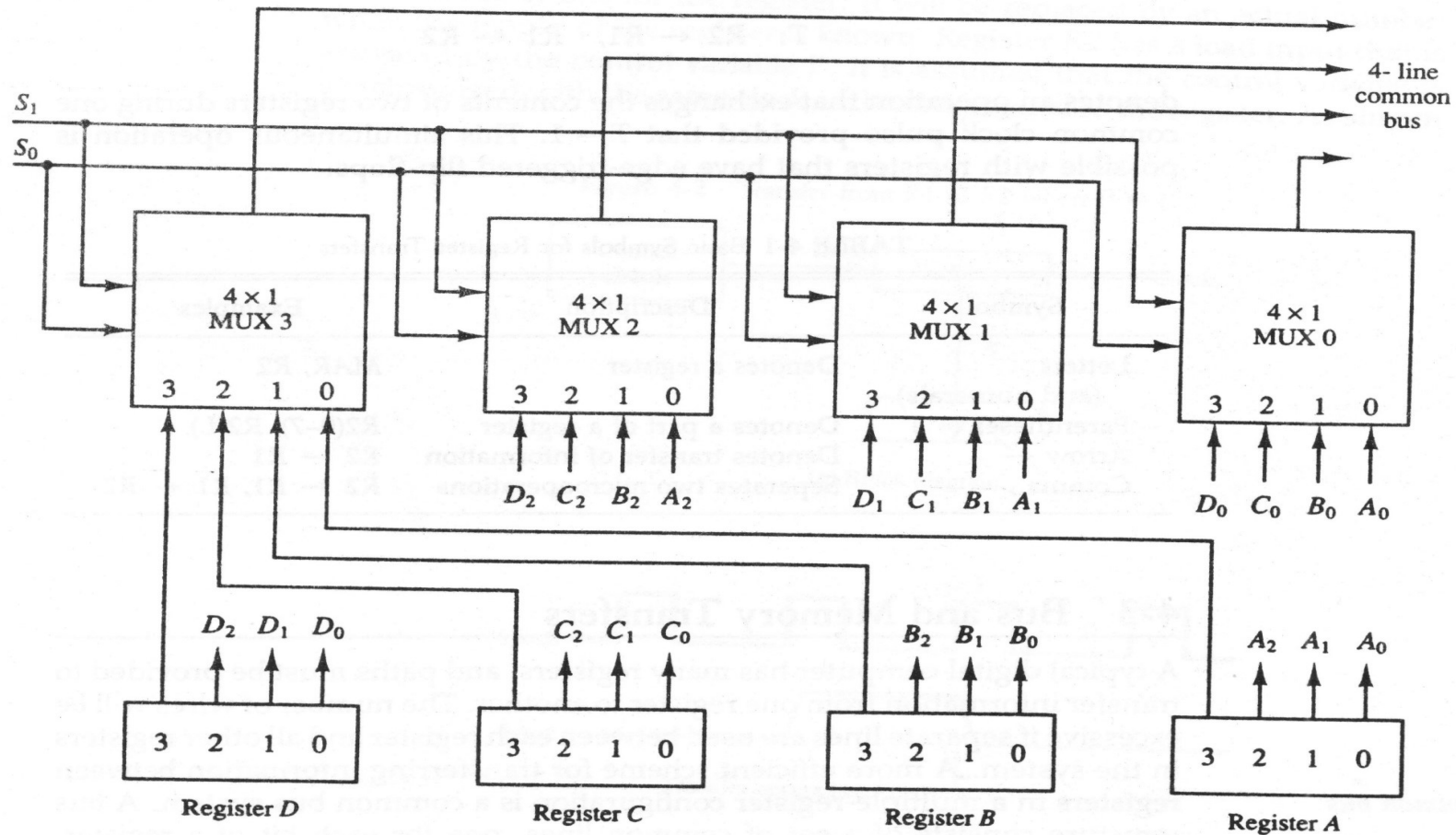Figure 4-3   Bus system for four registers.

TABLE 4-2   Function Table for Bus of Fig. 4-3

S_1	S_0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

13

Bus-based transfers for multiple registers
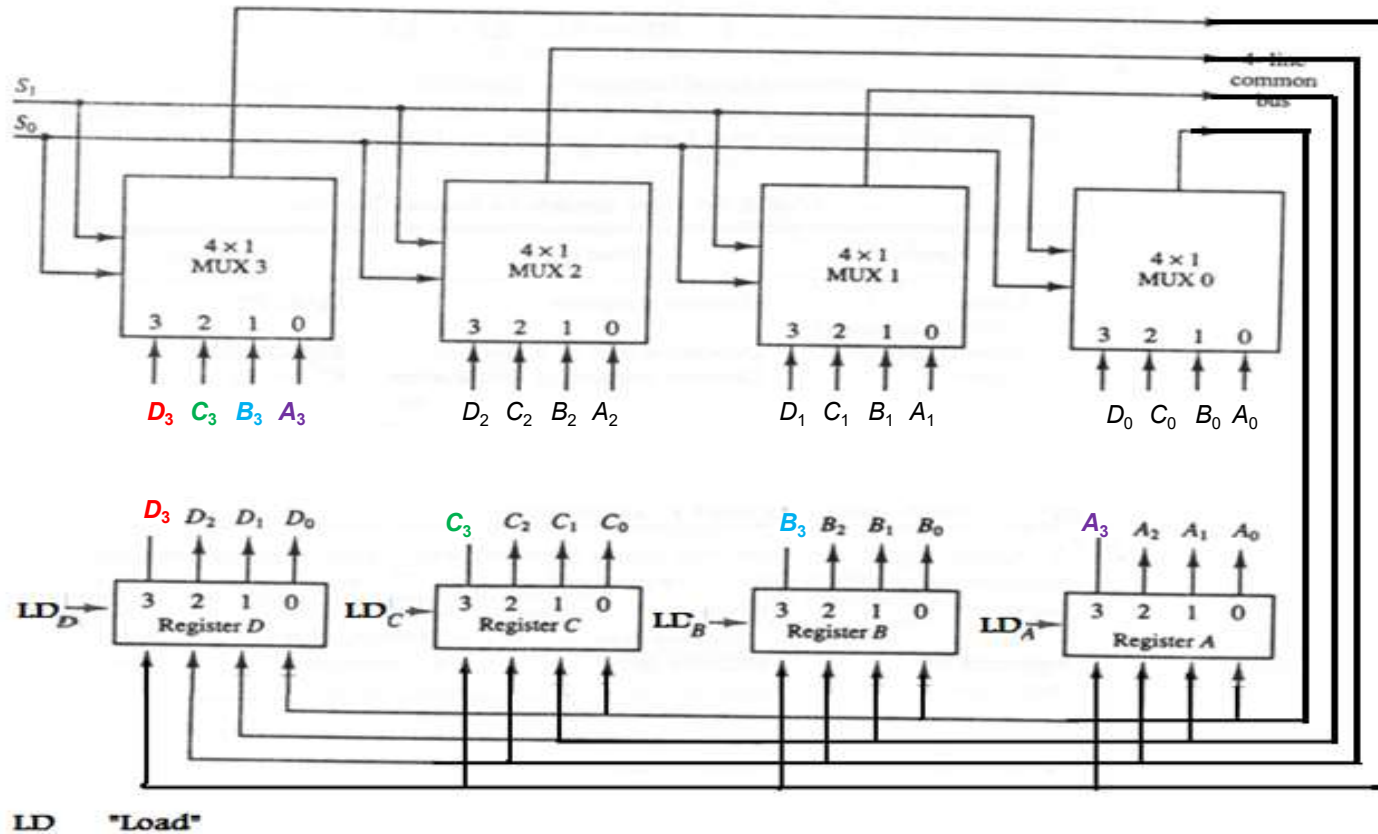
To transfer data using a bus:

- ◆connect the output of the source register to the bus;
- ◆connect the input of the target register to the bus;
- ◆when the clock pulse arrives, the transfer occurs

Examples: Bus-based transfers

S1	S0	LD$_A$	LD$_B$	LD$_C$	LD$_D$	Operation
0	0	0	1	0	0	Register B ← Register A
1	1	1	0	0	0	Register A ← Register D
0	0	0	1	1	0	Register B ← Register A; Register C ← Register A

14

Register Transfer Language

- **Micro operations** (micro-ops) are operations on data stored in registers

- **Register transfer language (RTL)** is a concise and precise means of describing those operations

- **RTL expressions** are made up of elements which describe the registers being manipulated, and the micro-ops being performed on them

- Registers are denoted by uppercase letters (sometimes followed by numbers) that indicate the function of the register
  - e.g. R0, R1, AR, PC, MAR, et al.
  - The individual bits can be denoted using parenthesis and bit numbers or labels
    - e.g. R0(0), R0(7:0), PC(L), PC(H)

R

(a) Register R

7 6 5 4 3 2 1 0

(b) Individual bits of 8-bit register

15   0

R2

(c) Numbering of 16-bit register

15   8 7   0

PC (H)	PC (L)
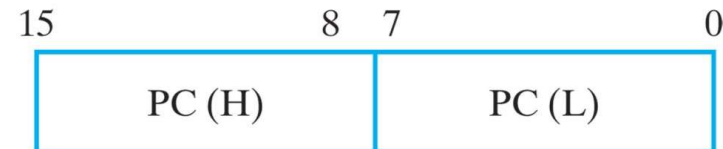
(d) Two-part 16-bit register

Register Transfer Language

TABLE 6-1
Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	$AR, R2, DR, IR$
Parentheses	Denotes a part of a register	$R2(1), R2(7\!:\!0), AR(L)$
Arrow	Denotes transfer of data	$R1 \leftarrow R2$
Comma	Separates simultaneous transfers	$R1 \leftarrow R2, R2 \leftarrow R1$
Square brackets	Specifies an address for memory	$DR \leftarrow M[AR]$

Destination register ← Source register:

- Data in source register does not change
- A datapath are available from the outputs of the source register to the inputs of the destination register
- The destination register has a parallel load capability
- All RTL statements occur in response to a **clock tick**
- Normally we want a given transfer to occur not for every clock pulse, but only for specific values of the control signals

16

Register Transfer Language

- RTL conditional statements:
  - e.g. If (K1 = 1) Then (R2 $\leftarrow$ R1)

 Control function notation (Colon, :)
  - e.g. K1: R2 $\leftarrow$ R1
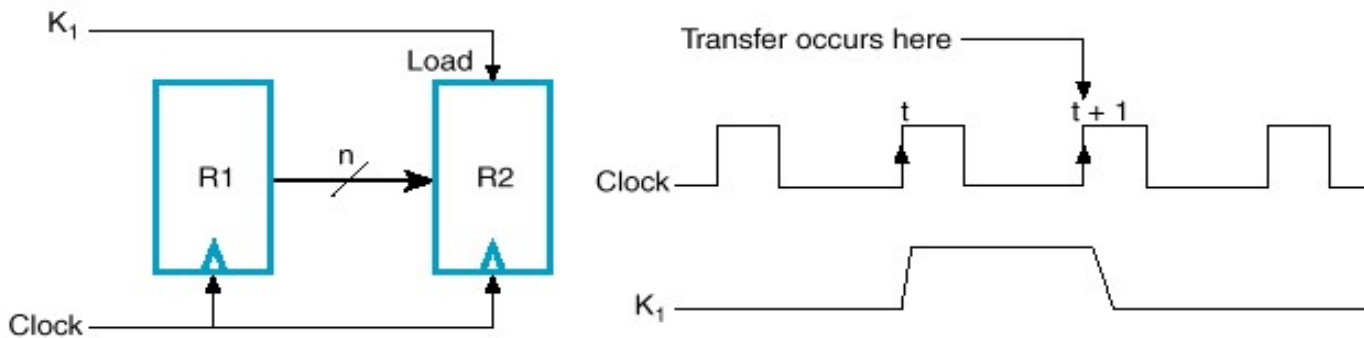
Register transfer operations

- We can apply arithmetic operations to registers.

$$R1 \leftarrow R2 + R3$$
$$R3 \leftarrow R1 - 1$$

- **Every RTL statement** written in register-transfer notation presupposes a hardware construct for implementing the transfer.

☐ **TABLE 6-3**
Arithmetic Microoperations

Symbolic Designation	Description
$R0 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R0$
$R2 \leftarrow \overline{R2}$	Complement of the contents of $R2$ (1s complement)
$R2 \leftarrow \overline{R2} + 1$	2s complement of the contents of $R2$
$R0 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus 2s complement of $R2$ transferred to $R0$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ (count up)
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ (count down)

Register Transfer Language

Bitwise operations:

- Most computers also support logical operations like AND, OR and NOT, but extended to multi-bit words instead of just single bits.

- To apply a logical operation to two words X and Y, apply the operation on each pair of bits X_i and Y_i:

```
        1 0 1 1              1 0 1 1              1 0 1 1
   AND  1 1 1 0        OR    1 1 1 0        XOR  1 1 1 0
   ─────────────      ──────────────       ─────────────
        1 0 1 0              1 1 1 1              0 1 0 1
```
```
                                                 NOT 1 0 1 1
                                                 ─────────────
                                                     0 1 0 0
```

- Single operand logical operation: "complementing" all the bits in a number.

□ **TABLE 6-4**
Logic Microoperations

Symbolic Designation	Description
$R0 \leftarrow \overline{R1}$	Logical bitwise NOT (1s complement)
$R0 \leftarrow R1 \wedge R2$	Logical bitwise AND (clears bits)
$R0 \leftarrow R1 \vee R2$	Logical bitwise OR (sets bits)
$R0 \leftarrow R1 \oplus R2$	Logical bitwise XOR (complements bits)

19

Register Transfer Language

Bitwise operations in programming

- Languages like C, C++ and Java provide bitwise logical operations:

    & (AND) | (OR)   ^ (XOR)   ~ (NOT)

- These operations treat each integer as a bunch of individual bits:

    13 & 25 = 9   because   01101 & 11001 = 01001

- They are *not* the same as the operators &&, || and !, which treat each integer as a single logical value (0 is false, everything else is true):

    13 && 25 = 1   because   true && true = true

- Bitwise operators are often used in programs to set a bunch of Boolean options, or flags, with one argument.
- Easy to represent sets of fixed universe size with bits:
    – 1: is member, 0 not a member. Unions: OR, Intersections: AND

Register transfer operations

- Finally, we can shift values left or right by one bit. The source register is not modified, and we assume that the shift input is always 0.

☐ **TABLE 6-5**
Examples of Shifts

Type	Symbolic Designation	Eight-Bit Examples	
		Source $R2$	After Shift: Destination $R1$
Shift left	$R1 \leftarrow sl\ R2$	10011110	00111100
Shift right	$R1 \leftarrow sr\ R2$	11100101	01110010