İstanbul
Bilgi Üniversitesi

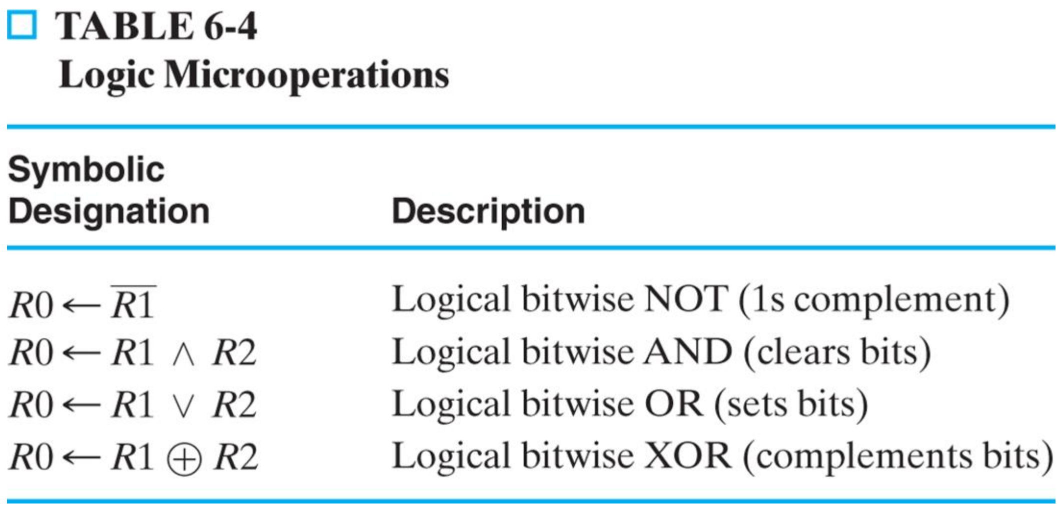
CMPE321 - Computer Architecture

Lecture 2
Microoperations

Hakan Ayral, PhD.

Logic Microoperations

- Logic microoperations are used to manipulating the bits stored in a register.
- These operations consider each bit in the register separately and treat it as a binary variable.
- The **NOT** microoperation, represented by a bar over the source register name, complements all bits and thus is the same as the **1's complement**.
- The symbol **∧** is used to denote the AND microoperation and the symbol **∨** to denote the OR microoperation.



□ **TABLE 6-4**
Logic Microoperations

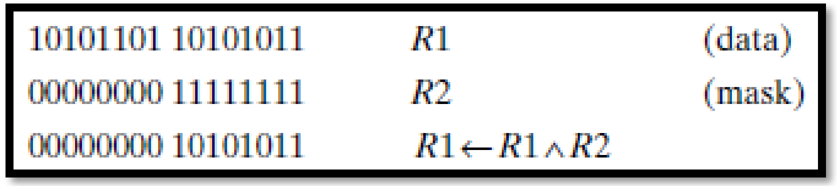
Symbolic Designation	Description
$R0 \leftarrow \overline{R1}$	Logical bitwise NOT (1s complement)
$R0 \leftarrow R1 \wedge R2$	Logical bitwise AND (clears bits)
$R0 \leftarrow R1 \vee R2$	Logical bitwise OR (sets bits)
$R0 \leftarrow R1 \oplus R2$	Logical bitwise XOR (complements bits)

- By using these special symbols, we can distinguish between the **add** microoperation represented by **+** and the **OR** microoperation.
  - The **+** symbol has two meanings, but we can distinguish between them by noting where the symbol occurs.
  - If **+** occurs in a microoperation, it denotes addition; if it occurs in a control or Boolean function, it denotes OR.
  - OR microoperation always use the **∨** symbol.
- For example, in the statement $(K_1 + K_2)$: $R1 \leftarrow R2 + R3,\ R4 \leftarrow R5 \vee R6$
  - the **+** between *K*1 and *K*2 is an OR operation between two variables in a control condition.
  - the **+** between *R*2 and *R*3 specifies an add microoperation.
  - the OR microoperation is designated by the **∨** symbol between registers *R*5 and *R*6.

Logic Microoperations

- Logic microoperations can be easily implemented with a group of gates, one for each bit position.
  - The NOT of a register of *n* bits is obtained with *n* NOT gates in parallel.
  - The AND microoperation is obtained using a group of *n* AND gates, each receiving a pair of corresponding inputs from the two source registers.
  - The outputs of the AND gates are applied to the corresponding inputs of the destination register.
- The logic microoperations can change bit values, clear a group of bits, or insert new bit values into a register.
- Example below show how the bits stored in the 16-bit register *R*1 can be selectively changed by using a logic microoperation and a logic operand stored in the 16-bit register *R*2.

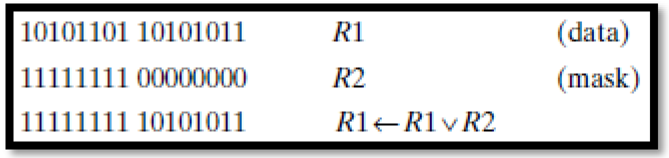
Logic Microoperations



10101101 10101011	$R1$	(data)
00000000 11111111	$R2$	(mask)
00000000 10101011	$R1 \leftarrow R1 \wedge R2$	

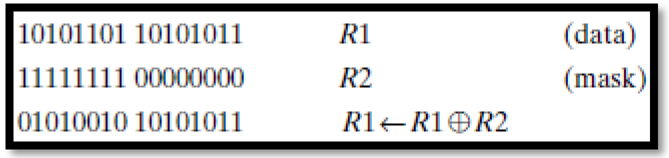
- The AND microoperation can be used for clearing one or more bits in a register to 0.
- The Boolean equations $X \cdot 0 = 0$ and $X \cdot 1 = X$ dictate that, when ANDed with 0, a binary variable *X* produces a 0, but when ANDed with 1, the variable remains unchanged.
- A given bit or group of bits in a register can be cleared to 0 if ANDed with 0.
- The 16-bit logic operand in *R*2 has 0s in the high-order byte and 1s in the low-order byte.
- By ANDing the contents of *R*2 with the contents of *R*1, it is possible to clear the high-order byte of *R*1 and leave the bits in the low-order byte unchanged.
- Thus, the AND operation can be used to selectively clear bits of a register.
- This operation is sometimes called *masking out* the bits, because it masks or deletes all 1s in the *data* in *R*1, based on bit positions that are 0 in the *mask* provided in *R*2.

Logic Microoperations



10101101 10101011	$R1$	(data)
11111111 00000000	$R2$	(mask)
11111111 10101011	$R1 \leftarrow R1 \vee R2$	

- The OR microoperation is used to set one or more bits in a register.
  - X+1 = 1 and X+0 = X so, when ORed with 1, the binary variable X produces a 1, but when ORed with 0, the variable remains unchanged.
  - A given bit or group of bits in a register can be set to 1 if ORed with 1.
  - Example above has high-order byte of $R1$ set to all 1s by ORing it with all 1s in the $R2$ operand.
  - The low-order byte remains unchanged because it is ORed with 0s.
- The XOR (exclusive-OR) microoperation can be used to complement one or more bits in a register.
  - $X \oplus 1 = X$ and $X \oplus 0 = X$, so when a binary variable X is XORed with 1, it is complemented, but when XORed with 0, the variable remains unchanged.
  - By XORing a bit or group of bits in register $R1$ with 1s in selected positions in $R2$, it is possible to complement the bits in the selected positions in $R1$.
  - Example below has the high-order byte in $R1$ is complemented after the XOR operation with $R2$, and the low-order byte is unchanged.



10101101 10101011	$R1$	(data)
11111111 00000000	$R2$	(mask)
01010010 10101011	$R1 \leftarrow R1 \oplus R2$	

Shift Microoperations

- Shift microoperations are used for lateral movement of data.
- The contents of a source register can be shifted either right or left.
- A *left shift* is toward the most significant bit, and a *right shift* is toward the least significant bit.
- Shift microoperations are used in the serial transfer of data. They are also used for manipulating the contents of registers in arithmetic, logical, and control operations.
- The destination register for a shift microoperation may be the same as or different from the source register.
- The **incoming bit**:
  - For a left-shift microoperation, we call the rightmost bit of the destination register the *incoming bit*.
  - For a right-shift microoperation, we define the leftmost bit of the destination register as the incoming bit.
  - The incoming bit may have different values, depending upon the type of shift microoperation.
- Here we assume that, for sr and sl, the incoming bit is 0, as shown in the examples in Table 5.
- The ***outgoing bit*** is the leftmost bit of the source register for the left-shift operation and the rightmost bit of the source register for the right-shift operation.
- For the left and right shifts shown, the **outgoing bit value is simply discarded**.

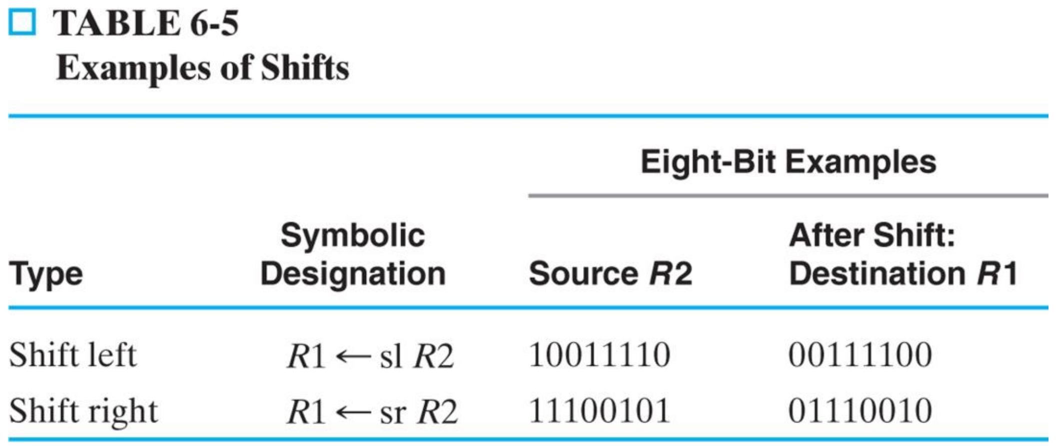


TABLE 6-5
Examples of Shifts

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

Microoperations on a Single Register

- Now we will look at the implementation of one or more microoperations with a single register as the destination of all primary results.
- Single register may also serve as a source of an operand for binary and unary operations.
- Because of its relation to a single set of storage elements and the microoperations, the combinational logic implementing the microoperations is assumed to be a part of the register and is called <u>**dedicated logic** of the register</u>.
- This is in contrast to <u>logic which is shared by multiple destination registers</u>.
  - In this case, the combinational logic implementing the microoperations is called **shared logic** for the set of destination registers.
- The combinational logic implementing the microoperations can use one or more common functional blocks or it can be designed specifically for the register.

Register Transfer Mechanisms

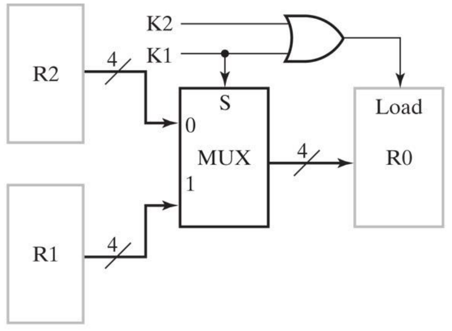
- **Multiplexer-Based Transfers**: Multiple inputs are selected by a multiplexer dedicated to the register
- **Bus-Based Transfers**: Multiple inputs are selected by a shared multiplexer driving a bus that feeds inputs to multiple registers
- **Three-State Bus**: Multiple inputs are selected by 3-state 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

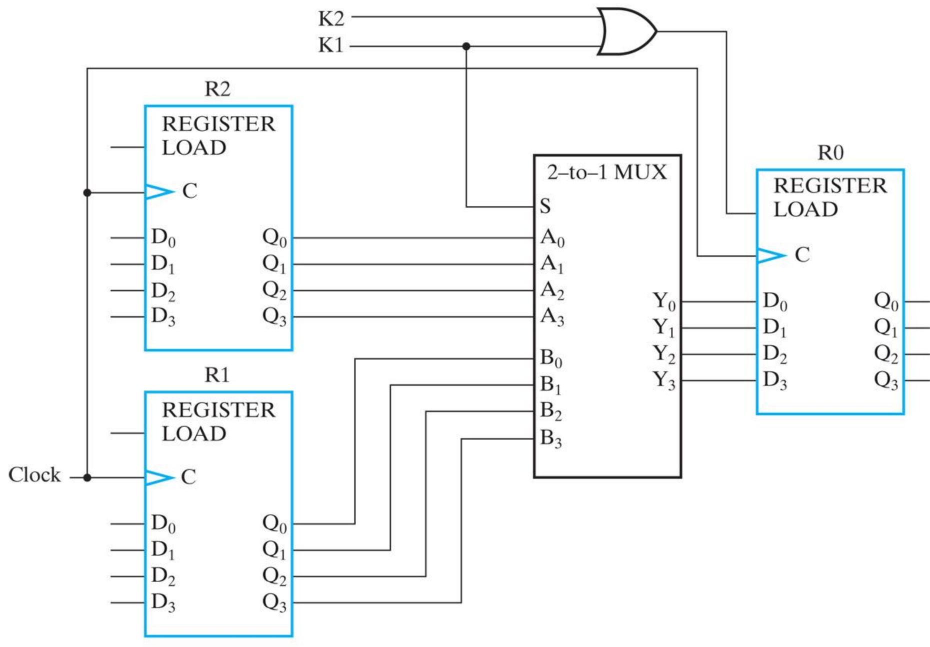
- Sometimes a register receives data from different sources at different times.
- Consider the statement: if $(K_1 = 1)$ then $(R0 \leftarrow R1)$ else if $(K_2 = 1)$ then $(R0 \leftarrow R2)$
- Contents of register *R*1 is transferred to register *R*0 when control signal *K*1 is 1.
- When *K*1 is 0,
  - the value in register *R*2 is transferred to *R*0 when *K*2 equals 1.
  - Otherwise, the contents of *R*0 remains unchanged.
- The conditional statement can be broken into two parts as:

$$K_1\colon R0 \leftarrow R1, \quad \overline{K}_1 K_2\colon R0 \leftarrow R2$$

- This specifies hardware connections from *R*1 and *R*2, to a common destination register *R*0.
- Making the selection between the two source registers is based on the control variables *K*1 and *K*2.



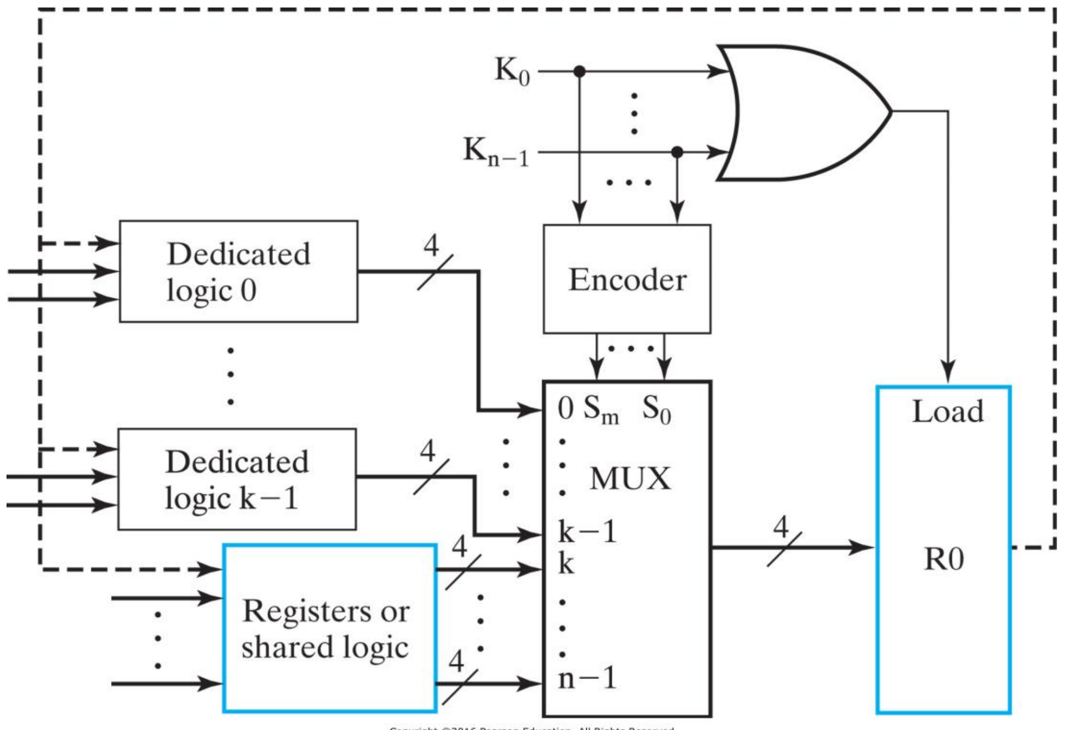
(a) Block diagram



(b) Detailed logic

Generalization of Multiplexer Selection for Multiple Sources

- Previous example can be generalized by allowing the multiplexer to have multiple sources which are either register outputs or combinational logic implementing microoperations. (the diagram to the right assumes this)
- When microoperations are dedicated to the register, the dedicated logic is included as a part of the register.
- In the figure, the first k sources are dedicated logic and the last n − k sources are either registers or shared logic.
- The control signals that select a given source are either a single control variable or the OR of all control signals corresponding to the microoperations associated with the source.
- To force R0 to load for a microoperation, these control signals are ORed together to form the Load signal.

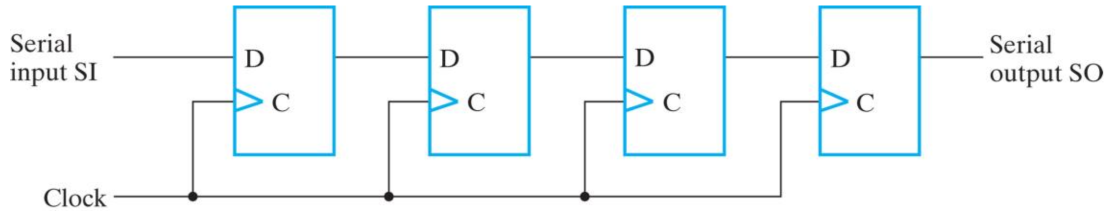


Generalization of Multiplexer Selection for Multiple Sources

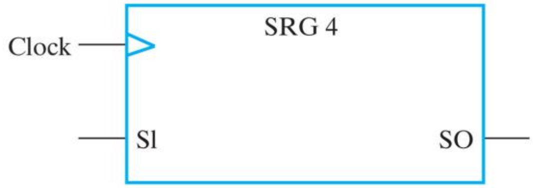
- On the previous example it is assumed that only one of the control signals is 1 at any time.
- Therefore these signals must be encoded to provide the selection codes for the multiplexer.
- Two modifications are possible:
  - The control signals could be applied directly to a 2 × n AND-OR circuit (i.e., a multiplexer with the decoder deleted).
  - The control signals could already be encoded, omitting the use of the all-zero code, so that the OR gate still forms the Load signal correctly.

Shift Registers

- The logical configuration of a shift register consists of a chain of flip-flops, with the output of one flip-flop connected to the input of the next flip-flop.
- All flip-flops have a common clock-pulse input that activates the shift.
- The simplest possible shift register uses only flip-flops (see figure).
- Output of every flip-flop is connected to the *D* input of the flip-flop at its right.
- The clock is common to all flip-flops.
- *Serial input SI* is the input to the leftmost flip-flop.
- Serial output SO is taken from the output of the rightmost flip-flop.
- A symbol for the shift register is given in (b).
- Sometimes it is necessary to control the register so that it shifts only when needed.
- Shift register on figure can be controlled by modulating (gating) the clock, with Shift replacing Load. (due to clock skew, this is not the best approach)
- Thus, we'll see that the shift operation can be controlled through the D inputs of the flip-flops rather than through the clock inputs C.



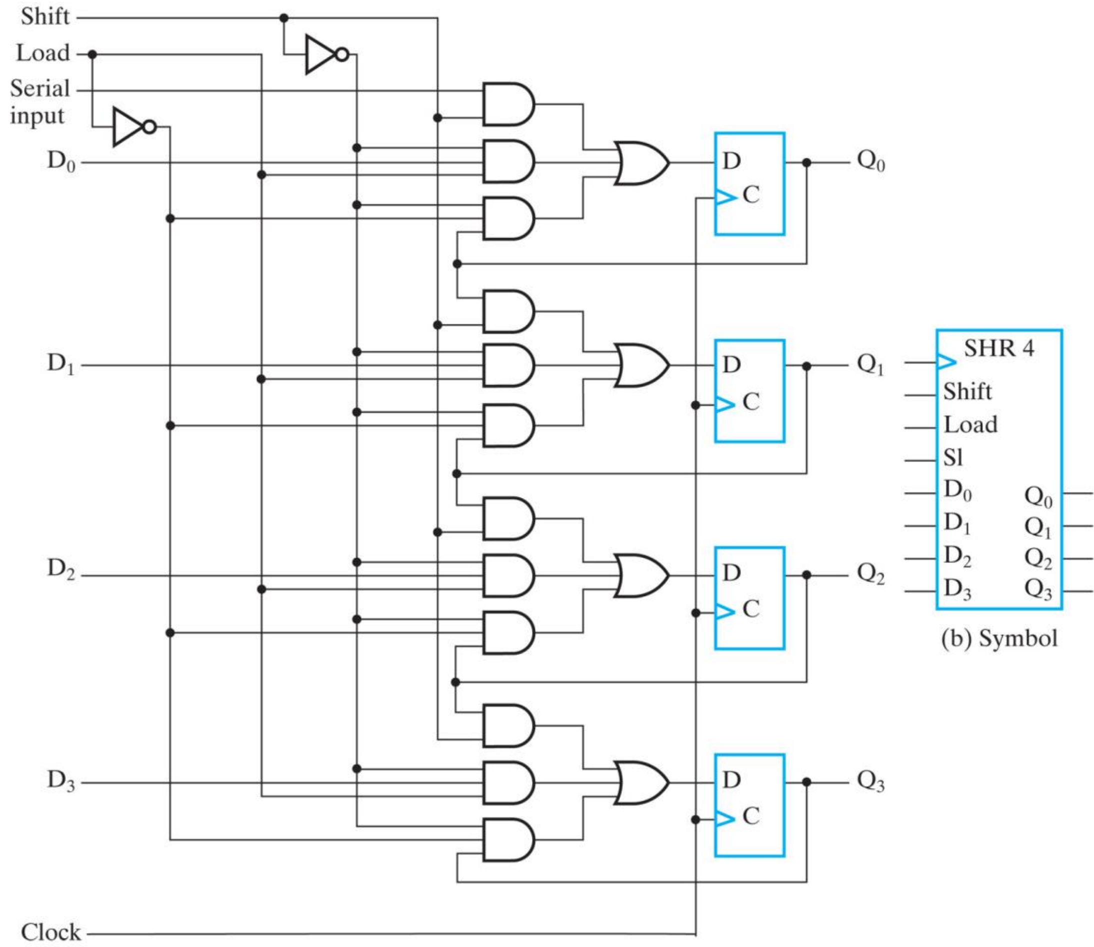
(a) Logic diagram



(b) Symbol

Shift Register with Parallel Load

- If all flip-flops of a shift register are readable, then information entered serially by shifting can be taken out in parallel from the flip-flop outputs.
- If a parallel load capability is also added to a shift register, then data entered in parallel can be shifted out serially.
- Thus, a shift register with readable flip-flops and parallel load can be used for converting incoming parallel data to outgoing serial data and vice versa.
- The logic diagram for a 4-bit shift register with parallel load and the symbol for this register are shown in Figure.
- There are two control inputs, one for the shift and the other for the load.
- Each stage of the register consists of a *D* flip-flop, an OR gate, and three AND gates.
    - The first AND enables the shift operation.
    - The second AND enables the input data.
    - The third AND restores the contents of the register when no operation is required.



(b) Symbol

Shift Register with Parallel Load

- The operation of this register is specified in Table on right and is also given by the register transfers $Shift: Q \leftarrow \mathrm{sl}\,Q$ and $\overline{Shift} \cdot Load: Q \leftarrow D$
- The "No change" operation, also called "Hold", is implicit if neither of the conditions are satisfied.
- When both the shift and load control inputs are 0, the third AND gate in each stage is enabled, and the output of each flip-flop is applied to its own *D* input.

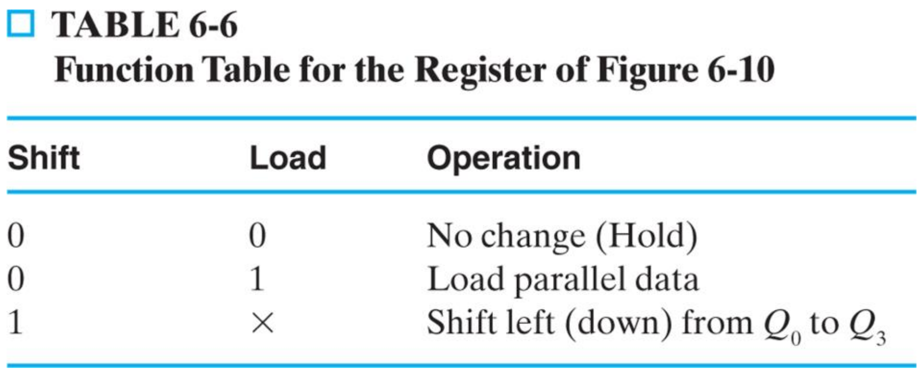


TABLE 6-6
Function Table for the Register of Figure 6-10

Shift	Load	Operation
0	0	No change (Hold)
0	1	Load parallel data
1	×	Shift left (down) from Q_0 to Q_3

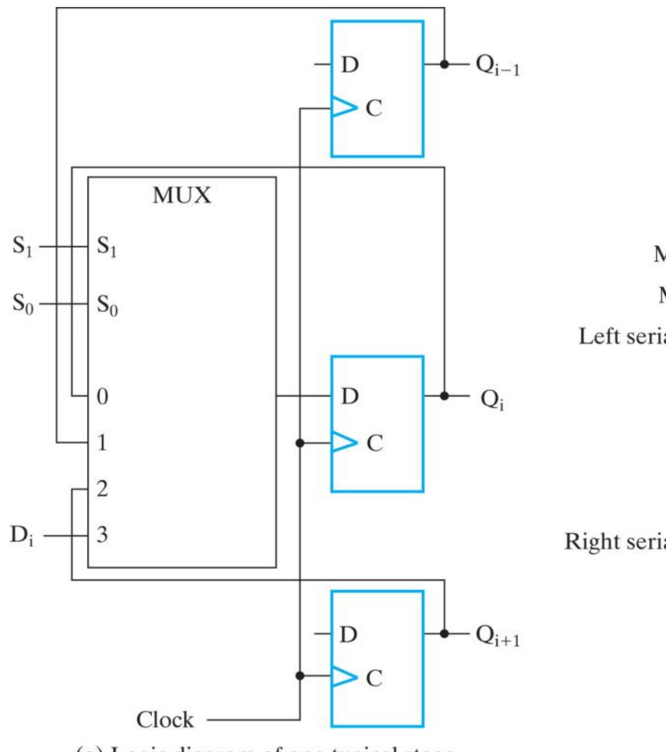
- A positive transition of the clock restores the contents to the register, and the output is unchanged.
- When the shift input is 0 and the load input is 1, the second AND gate in each stage is enabled, and the input *Di* is applied to the *D* input of the corresponding flip-flop.
- The next positive clock transition transfers the parallel input data into the register.
- When the shift input is equal to 1, the first AND gate in each stage is enabled and the other two are disabled.
- Since the *Load* input is disabled by the input on the second AND gate, we mark it with a don't-care condition in the *Shift* row of the table.
- When a positive edge occurs on the clock, the shift operation causes the data from the serial input *SI* to be transferred to flip-flop *Q*0, the output of *Q*0 to be transferred to flip-flop *Q*1, and so on down the line.

Shift Register with Parallel Load

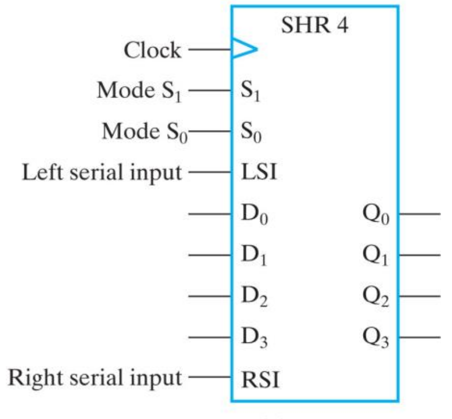
- Shift registers are often used to interface digital systems that are distant from each other.
- Suppose it is necessary to transmit an *n*-bit quantity between two points; if the distance is large, it is expensive to use *n* lines to transmit the *n* bits in parallel.
- It is more economical to use a single line and transmit the information serially, one bit at a time.
- The **transmitter** loads the *n*-bit data in parallel into a shift register and then transmits the data serially along the common line.
- The **receiver** accepts the data serially into a shift register.
- When all *n* bits are accumulated, they can be taken in parallel from the outputs of the register.
- Thus, the transmitter performs a parallel-to-serial conversion of data, and the receiver does a serial-to-parallel conversion.

Bidirectional Shift Register

- A register shifting in only one direction is called a **unidirectional shift register**.
- A register that can shift in both directions is a **bidirectional shift register**.
- It is possible to modify the circuit on previous Figure, by adding a fourth AND gate in each stage, for shifting the data in the upward direction.
- The four AND gates, together with the OR gate in each stage, constitute a multiplexer with the selection inputs controlling the operation of the register.



(a) Logic diagram of one typical stage



(b) Symbol

- One stage of a bidirectional shift register with parallel load is shown in (a).
- Each stage consists of a D flip-flop and a 4-to-1-line multiplexer.
- The two selection inputs S1 and S0 select one of the multiplexer inputs to apply to the D flip-flop.

Bidirectional Shift Register

- The selection lines control the mode of operation of the register according to Table on right and the register transfers below:

$$\overline{S}_1 \cdot S_0: \quad Q \leftarrow \mathrm{sl}\, Q$$
$$S_1 \cdot \overline{S}_0: \quad Q \leftarrow \mathrm{sr}\, Q$$
$$S_1 \cdot S_0: \quad Q \leftarrow D$$

- The “No Change” operation is implicit if none of the conditions for transfers is satisfied.
- When the mode control *S*1*S*0 = 00, input 0 of the multiplexer is selected.
    - This forms a path from the output of each flip-flop into its own input.
    - The next clock transition transfers the current stored value back into each flip-flop, and no change of state occurs.
- When *S*1*S*0 = 01,
    - the terminal marked 1 on the multiplexer has a path to the *D* input of each flip-flop.
    - These paths cause a shift-down operation.
    - The serial input is transferred into the first stage, and the content of each stage, *Qi* – 1, is transferred into stage *Qi* .
- When *S*1*S*0 = 10,
    - a shift-up operation results in a second serial input that enters the last stage.
    - In addition, the value in each stage Qi+1 is transferred into stage Qi .
- When S1S0 = 11,
    - the binary information on each parallel input line is transferred into the corresponding flip-flop, resulting in a parallel load.

TABLE 6-7
Function Table for the Register of Figure 6-11

Mode Control		Register Operation
S_1	S_0	
0	0	No change (Hold)
0	1	Shift left
1	0	Shift right
1	1	Parallel load