İstanbul
Bilgi Üniversitesi

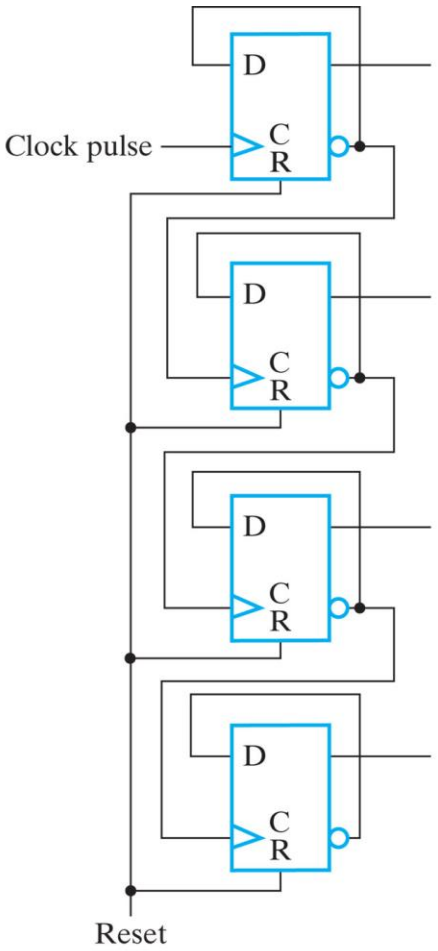
CMPE321 - Computer Architecture

Lecture 3
Counters and Bus-based Transfers for Multiple Registers

Hakan Ayral, PhD.

Ripple Counter

- A register that goes through a prescribed sequence of distinct states upon the application of a sequence of input pulses is called a **counter**.
  - The input pulses may be clock pulses or may originate from some other source.
  - The sequence of states may follow the binary number sequence or any other specific sequence of states.
  - A counter that follows the binary number sequence is called a **binary counter**. An n-bit binary counter consists of n FFs and can count in binary from 0 through 2^n - 1.
- Counters are available in two categories: **ripple counters** and **synchronous counters**.
  - In a ripple counter, the FF output transitions serve as the sources for triggering the changes in other FFs; In other words, the *C* inputs of some of the FFs are triggered by the transitions that occur on other FF outputs (instead of clock signal).
  - In a synchronous counter, the *C* inputs of FFs receive the common clock pulse, and the change of state is determined from the present state of the counter.
- The logic diagram of a **4-bit binary ripple counter** is shown on the Figure to the right;
  - it is constructed from *D*-FFs connected such that the application of a positive edge to the *C* input of each FF causes the FF to complement its state.
  - The complemented output of each FF is connected to the *C* input of the next most significant FF. The FF holding the least significant bit receives the incoming clock pulses.
  - Positive-edge triggering makes each FF complement its value when the signal on its *C* input goes through a positive transition.
  - Positive transition occurs when the complemented output of the previous FF, to which *C* is connected, goes from 0 to 1.
  - HI (1) signal on *Reset* driving the *R* inputs clears the register to all zeros asynchronously.



Ripple Counter

- To understand the operation of a binary ripple counter, let us examine the upward counting sequence given in the left half of Table.
  - The count starts at binary 0 and increments by one with each count pulse.
  - After the count of 15, the counter goes back to 0 to repeat the count.
- The least significant bit (Q0) is complemented by each count pulse.
  - Every time that Q0 goes from 1 to 0, $\overline{Q0}$ goes from 0 to 1, complementing Q1.
  - Every time that Q1 goes from 1 to 0, it complements Q2.
  - Every time that Q2 goes from 1 to 0, it complements Q3, and so on for any higher-order bits in the ripple counter.
- For example, consider the transition from count 0011 to 0100.
  - Q0 is complemented with the count pulse positive edge.
  - Since Q0 goes from 1 to 0, it triggers Q1 and complements it.
  - As a result, Q1, goes from 1 to 0, which complements Q2, changing it from 0 to 1.
  - Q2 does not trigger Q3, because 2 produces a negative transition, and the flip-flops respond only to positive transitions.
  - Thus, the count from 0011 to 0100 is achieved by changing the bits one at a time.
  - The counter goes from 0011 to 0010 (Q0 from 1 to 0), then to 0000 (Q1 from 1 to 0), and finally to 0100 (Q2 from 0 to 1).
- The flip-flops change one at a time in quick succession as the signal propagates through the counter in a ripple fashion from one stage to the next.

TABLE 6-8
Counting Sequence of Binary Counter

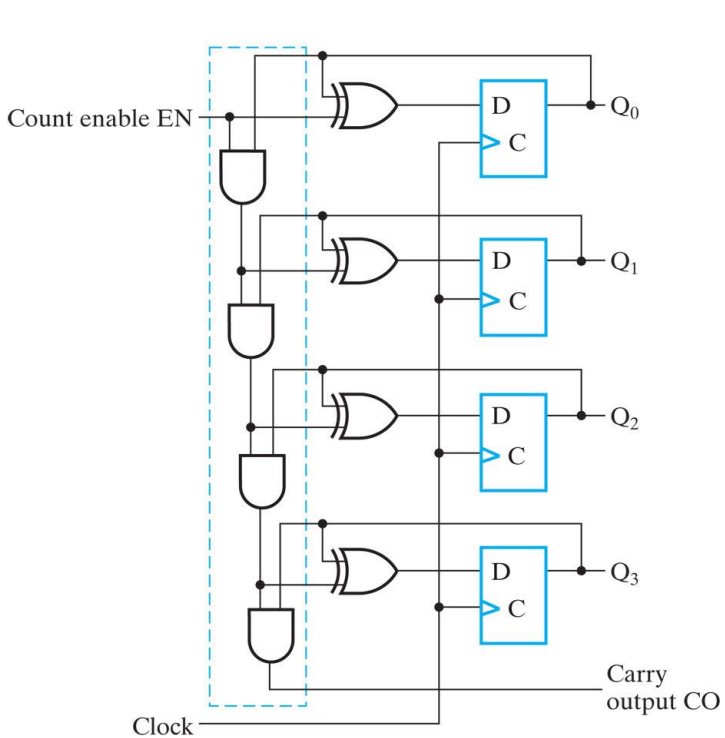
Upward Counting Sequence				Downward Counting Sequence			
Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	0
0	0	1	0	1	1	0	1
0	0	1	1	1	1	0	0
0	1	0	0	1	0	1	1
0	1	0	1	1	0	1	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	0	0
1	0	0	0	0	1	1	1
1	0	0	1	0	1	1	0
1	0	1	0	0	1	0	1
1	0	1	1	0	1	0	0
1	1	0	0	0	0	1	1
1	1	0	1	0	0	1	0
1	1	1	0	0	0	0	1
1	1	1	1	0	0	0	0

Ripple Counter

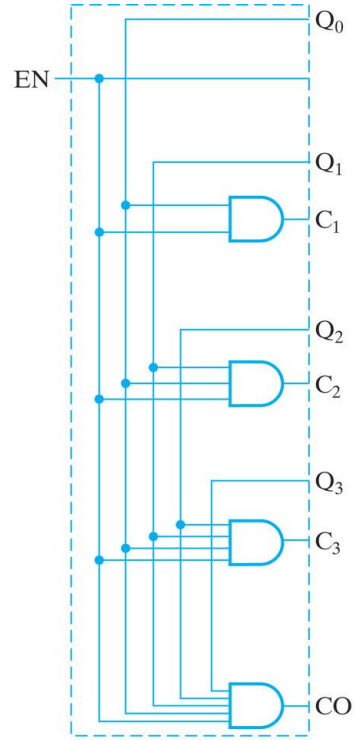
- The advantage of ripple counters is their simple hardware.
- Unfortunately, they are asynchronous circuits and, with added logic, can become circuits with delay dependence and unreliable operation.
- This is particularly true for logic that provides feedback paths from counter outputs back to counter inputs.
- Also, due to the length of time required for the ripple to finish, large ripple counters can be slow circuits.
- As a consequence, synchronous binary counters are favored in all but low-power designs, where ripple counters have an advantage.

Synchronous Binary Counter

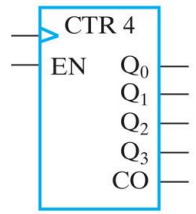
- **Synchronous counters**, in contrast to ripple counters, have the clock applied to the C inputs of all FFs.
- Thus, the common clock pulse triggers all flip-flops simultaneously rather than one at a time, as in a ripple counter.
- A synchronous binary counter can be constructed from an incrementer and D-FFs, as shown in (a).
- The carry output CO is added by not placing an X value on the C4 output before the contraction of an adder to the incrementer.
- Output CO is used to extend the counter to more stages.
- Note that the FFs trigger on the positive-edge transition of the clock.
- The polarity of the clock is not essential here, as it was for the ripple counter.
- The synchronous counter can be designed to trigger with either the positive or the negative clock transition.



(a) Logic diagram—serial gating



(b) Logic diagram—parallel gating



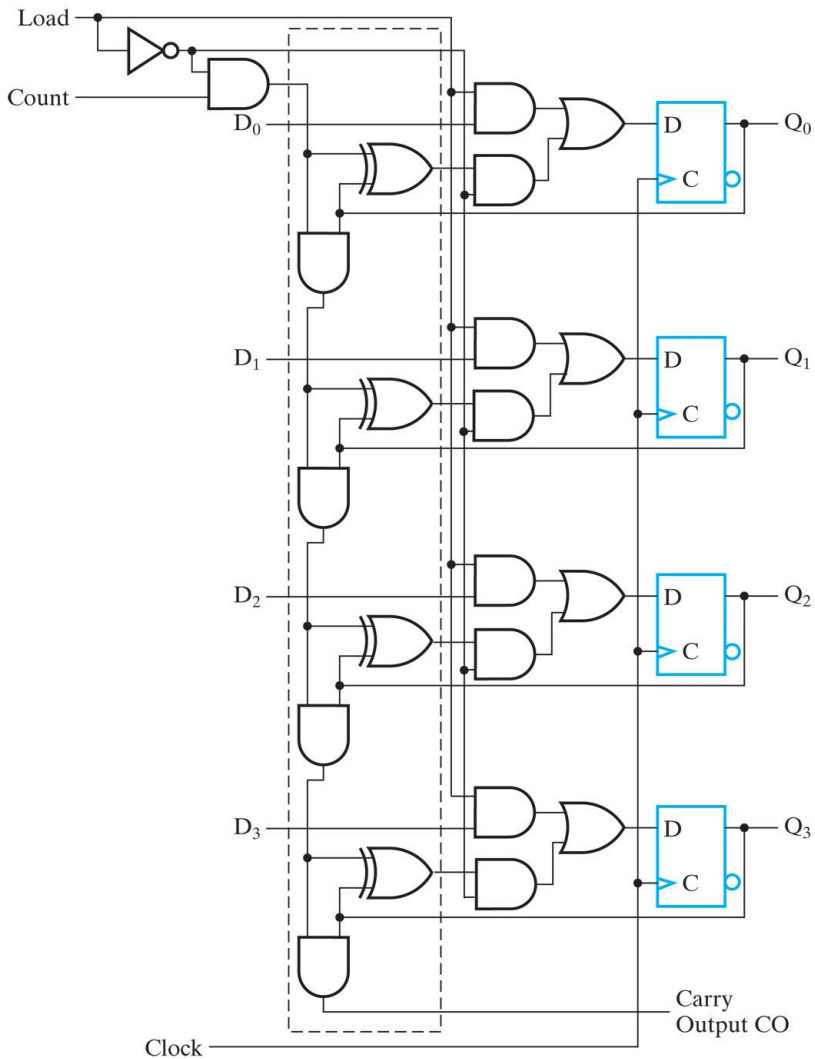
(c) Symbol

Serial and Parallel Counters

- We will use the synchronous counter to demonstrate two alternative designs for binary counters.
- In previous figure (a), a chain of 2-input AND gates is used to provide information to each stage about the state of the prior stages in the counter.
  - This is similar to the carry logic in the ripple carry adder.
  - A counter that uses such logic is said to have **serial gating** and is referred to as a **serial counter**.
  - The analogy to the ripple carry adder suggests that there might be counter logic analogous to the carry lookahead adder.
  - Such logic can be derived by contracting a carry lookahead adder, with the result shown in previous figure (b).
  - This logic can simply replace that in the blue box in (a) to produce a counter with **parallel gating**, called a **parallel counter**.
- The advantage of parallel gating logic is that, in going from state 1111 to state 0000, only one AND-gate delay occurs instead of the four AND gate delays that occur for the serial counter.
  - This reduction in delay allows the counter to operate much faster.
- If we connect two 4-bit parallel counters together by connecting the *CO* output of one to the *EN* input of the other, the result is an 8-bit serial-parallel counter.
  - This counter has two 4-bit parallel parts connected in series with each other.
  - The idea can be extended to counters of any length.
- Again, employing the analogy to carry lookahead adders, additional levels of gating logic can be introduced to replace the serial connections between the 4-bit segments.
- The added reduction in delay that results is useful for constructing large, fast counters.

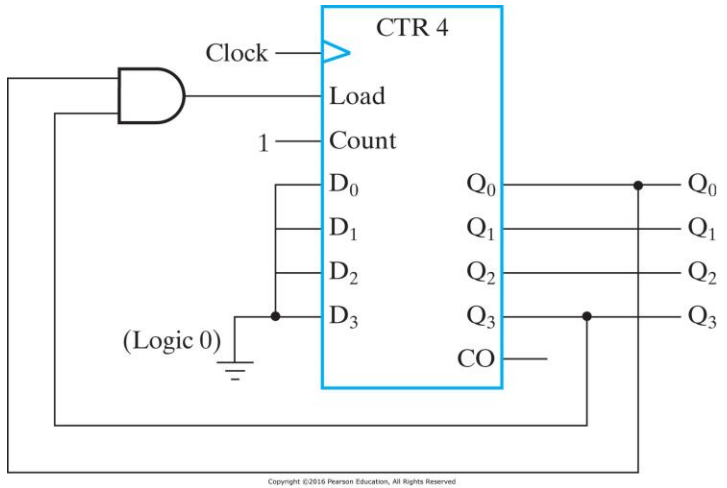
Binary Counter with Parallel Load

- Counters used in digital systems often require a parallel-load capability for transferring an initial number into the counter before counting.
- Two inputs control the operation, **Load** and **Count**.
- These inputs can take on four combinations, but only three operations are provided: ***Load*** (10), ***Count*** (01), and ***Hold*** (00).
  - The effect of the remaining input combination (11) will be considered shortly.
- The implementation uses an **incrementer,** 2n+1 **ENABLEs**, a **NOT** gate, and n **OR** gates as shown in Figure on right.
- The **1st n ENABLEs** with enable input *Load* are used to enable and disable the parallel load of input data, D.
- The **2nd n ENABLEs** with enable input $\overline{Load}$ on the incrementer outputs are used to disable both the count and hold operations when Load = 1.
- When Load = 0, both count and hold are enabled.
- Without the additional ENABLE, *Count* = 1, causes counting, and with *Count* = 0 the hold operation occurs.
- What about the (11) combination? Counting is disabled by the $\overline{Load}$ signal and loading is enabled by *Load*.
- What about the output CO? With *Count* = 1, the carry chain for the incrementer is active and can produce CO equal to 1. But CO should not be active outside of the counting operation.
- To deal with this problem, Count is enabled using $\overline{Load}$. With *Load* = 1, $\overline{Load}$ = 0, disabling Count into the carry chain. forcing CO to 0.
- Thus, for (11), a load occurs. This is sometimes described as Load overriding Count.
- When 4-bit counters are concatenated to form 4n-bit counters, for the first state, a count control input is attached to Count in the least significant stage.
- For all other stages, CO from the prior state is attached to Count.



Binary Counter with Parallel Load

- The binary counter with parallel load can be converted into a synchronous **BCD counter** (without load input) by connecting an external AND gate to it, as shown in Figure.
- The counter starts with an all-zero output, and the count input is always active.
- As long as the output of the AND gate is 0, each positive clock edge increments the counter by 1.
- When the output reaches the count of 1001, both *Q*0 and *Q*3 become 1, making the output of the AND gate equal to 1.
- This condition makes *Load* active; so on the next clock transition, the counter does not count, but is loaded from its four inputs.
- Since all four inputs are connected to logic 0, 0000 is loaded into the counter following the count of 1001.
- Thus, the circuit counts from 0000 through 1001, followed by 0000, as required for a BCD counter.



Other Counters

- Counters can be designed to generate any desired number of states in sequence.
- A ***divide-by-N counter*** (aka *modulo-N counter*) is a counter that goes through a repeated sequence of *N* states.
- The sequence may follow the binary count or may be an arbitrary sequence.
- In either case, the design of the counter follows the procedure for the design of synchronous sequential circuits.
- To demonstrate this procedure, we present the design of two counters: a **BCD counter** and **a counter with an arbitrary sequence** of states.
- We have seen a BCD counter obtained from a binary counter with parallel load. It is also possible to design a BCD counter directly using individual flip-flops and gates.
- Assuming *D*-FFs for the counter, we list the present states and corresponding next states in Table to the right.
- An output *Y* is included in the table.
  - This output is equal to 1 when the present state is 1001.
  - In this way, *CO* can enable the count of the next decade while its own decade switches from 1001 to 0000.

TABLE 6-9
State Table and Flip-Flop Inputs for BCD Counter

Present State				Next State				Output
Q_8	Q_4	Q_2	Q_1	$D_8 = Q_8(t+1)$	$D_4 = Q_4(t+1)$	$D_2 = Q_2(t+1)$	$D_1 = Q_1(t+1)$	Y
0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	1	0	0
0	0	1	0	0	0	1	1	0
0	0	1	1	0	1	0	0	0
0	1	0	0	0	1	0	1	0
0	1	0	1	0	1	1	0	0
0	1	1	0	0	1	1	1	0
0	1	1	1	1	0	0	0	0
1	0	0	0	1	0	0	1	0
1	0	0	1	0	0	0	0	1

Other Counters

- The flip-flop input equations for *D* are obtained from the next-state values listed in the table and can be simplified by means of K-maps.
- The unused states for minterms 1010 through 1111 are used as don't-care conditions.
- The simplified input equations for the BCD counter are

$$D_1 = \overline{Q}_1$$
$$D_2 = Q_2 \oplus Q_1\overline{Q}_8$$
$$D_4 = Q_4 \oplus Q_1Q_2$$
$$D_8 = Q_8 \oplus (Q_1Q_8 + Q_1Q_2Q_4)$$
$$CO = Q_1Q_8.$$

- Synchronous BCD counters can be cascaded to form counters for decimal numbers of any length.
- The cascading is done by replacing *D*1 with *D*1 = *Q*1 *CI*, where *CI* is an input driven by *CO* from the next lower BCD counter.
- Also, *CI* needs to be ANDed with the product terms to the right of each of the XOR symbols in each of the equations for *D*2 through *D*8.

Arbitrary Count Sequence

- Suppose we wish to design a counter that has a repeated sequence of six states, as listed in Table to the right.
- The count sequence for the counter is not straight binary, and two states, 011 and 111, are not present in the count.
- The *D* flip-flop input equations can b simplified using minterms 3 and 7 as don't-car conditions.
- The simplified functions are $D_A = A \oplus B$

$$D_B = C$$

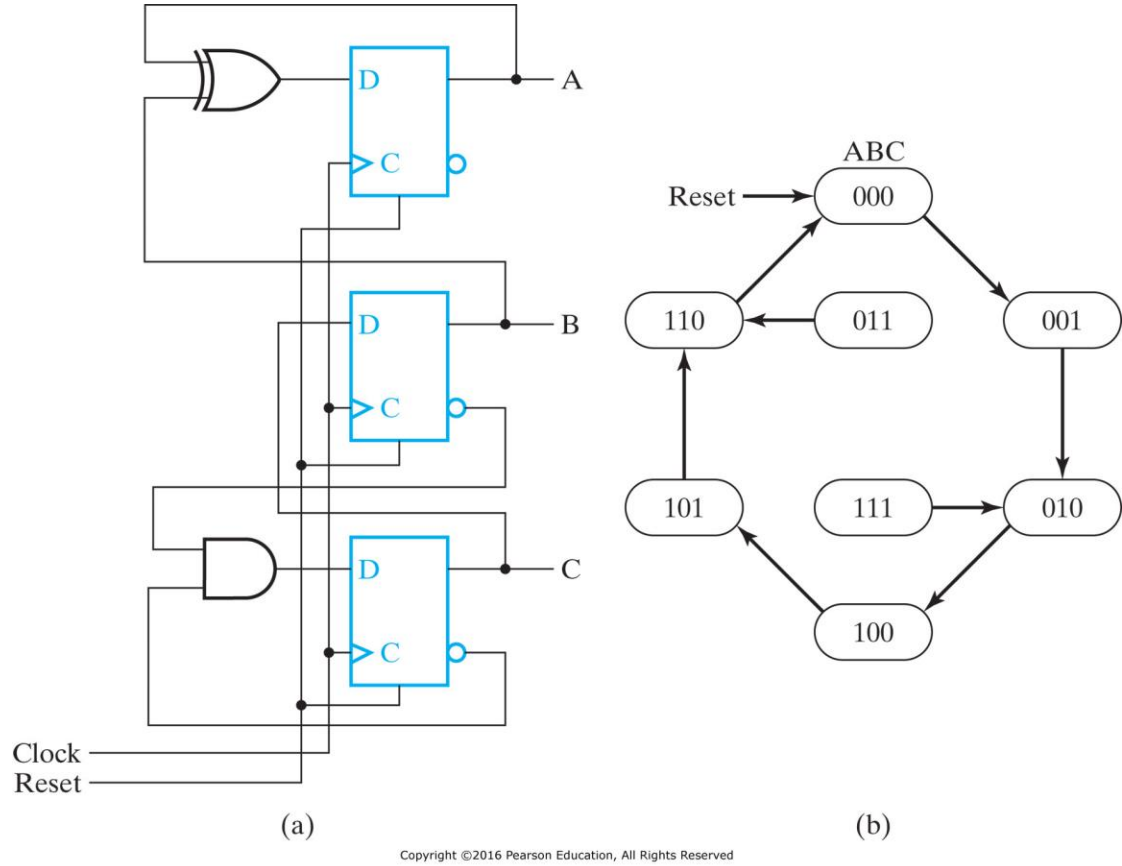
$$D_C = \overline{B}\overline{C}$$

TABLE 6-10
State Table and Flip-Flop Inputs for Counter

Present State			Next State		
A	B	C	DA = A(t + 1)	DB = B(t + 1)	DC = C(t + 1)
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	0	0	0

Arbitrary Count Sequence

- The logic diagram of the counter is shown in (a).
- Since there are two unused states, we analyze the circuit to determine their effect.
- The state diagram obtained is drawn in (b).
- This diagram indicates that if the circuit ever goes to one of the unused states, the next count pulse transfers it to one of the valid states, and the circuit then continues to count correctly.

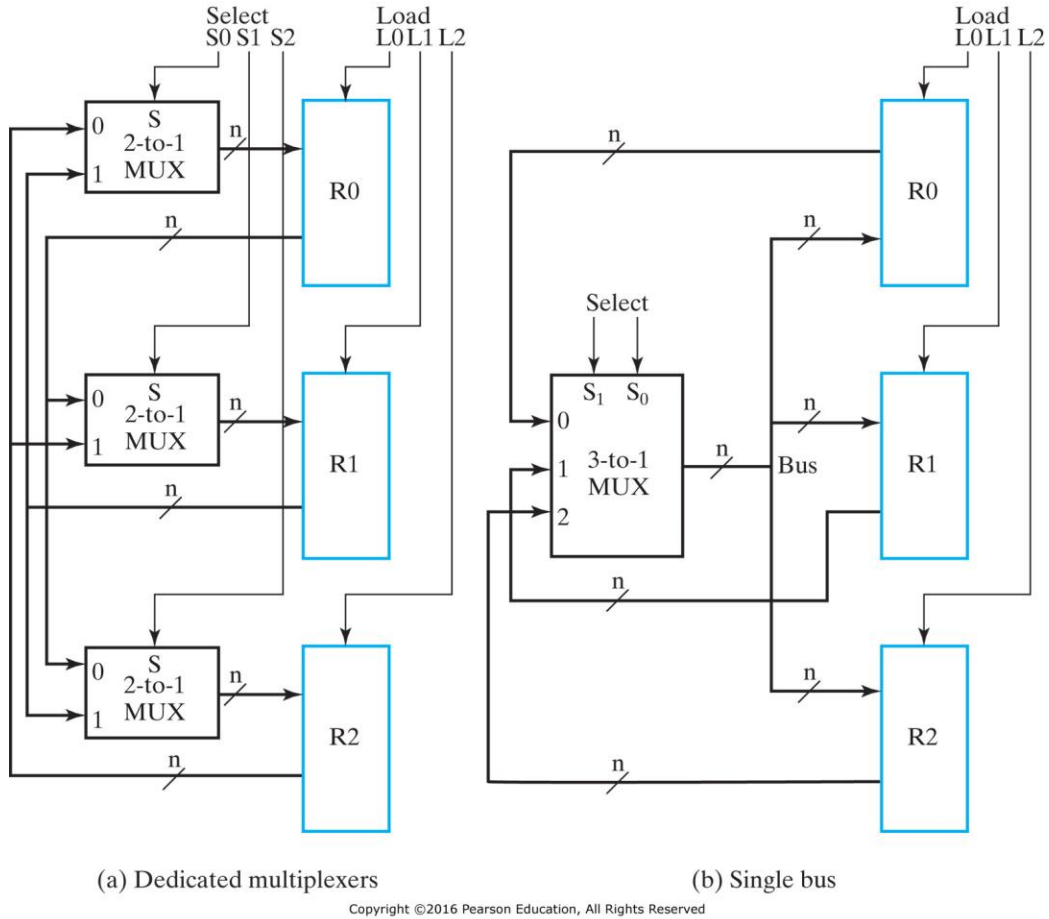


Multiplexer and Bus-based Transfers for Multiple Registers

- A typical digital system has many registers.
- Paths must be provided to transfer data from one register to another.
- The amount of logic and the number of interconnections may be excessive if each register has its own dedicated set of multiplexers.
- A more efficient scheme for transferring data between registers is a system that uses a shared transfer path called a **bus**.
- A bus is characterized by a set of common lines, with each line driven by selection logic.
- **Control signals for the logic select a single source and one or more destinations on any clock cycle for which a transfer occurs**.

Multiplexer and Bus-based Transfers for Multiple Registers

- We saw that multiplexers and parallel load registers can be used to implement dedicated transfers from multiple sources.
- A block diagram for such transfers between three registers is shown in (a).
- There are three n-bit 2-to-1 multiplexers, each with its own select signal.
- Also, each register has its own load signal.
- The same system can be implemented by using a single n-bit 3:1 multiplexer and parallel load registers based on a **bus** architecture.
- If a set of multiplexer outputs is shared as a common path, these output lines are a bus.
- Such a system with a single bus for transfers between three registers is shown in (b).
- The control input pair, **Select**, determines the contents of the single source register that will appear on the multiplexer outputs (i.e., on the bus).
- The **Load** inputs determine the destination register or registers to be loaded with the bus data.



Multiplexer and Bus-based Transfers for Multiple Registers

- In Table on the right, transfers using the single-bus implementation of (b) from previous figure are illustrated.
- The first transfer is from R2 to R0.
  - Select equals 10, selecting input R2 to the multiplexer.
  - Load signal L0 for register R0 is 1, with all other loads at 0, causing the contents of R2 on the bus to be loaded into R0 on the next positive clock transition.
- The second transfer in the table illustrates the loading of the contents of R1 into both R0 and R2.
  - The source R1 is selected because Select is equal to 01.
  - In this case, L2 and L0 are both 1, causing the contents of R1 on the bus to be loaded into registers R0 and R2.
- The third transfer, an exchange between R0 and R1, is impossible in a single clock cycle, since it requires two simultaneous sources, R0 and R1, on the single bus.
  - Thus, this transfer requires at least two buses or a bus combined with a dedicated path from one of the registers to the other.
  - Note that such a transfer can be executed on the dedicated multiplexers in (a) from previous figure.
- So, for a single-bus system, simultaneous transfers with different sources in a single clock cycle are impossible, whereas for the dedicated multiplexers, any combination of transfers is possible.
  - Hence, the reduction in hardware that occurs for a single bus in place of dedicated multiplexers results in limitations on simultaneous transfers.

TABLE 6-13
Examples of Register Transfers Using the Single Bus in Figure 6-19(b)

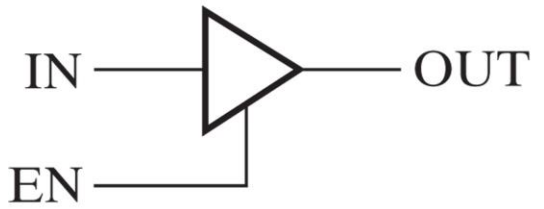
	Select		Load		
Register Transfer	**S1**	**S0**	**L2**	**L1**	**L0**
$R0 \leftarrow R2$	1	0	0	0	1
$R0 \leftarrow R1, R2 \leftarrow R1$	0	1	1	0	1
$R0 \leftarrow R1, R1 \leftarrow R0$	Impossible				

Multiplexer and Bus-based Transfers for Multiple Registers

- If we assume that only single-source transfers are needed, then we can use previous Figure to compare the complexity of the hardware in dedicated versus bus-based systems.
- First of all, assume a multiplexer design.
- In (a), there are 2n AND gates and n OR gates per multiplexer (not counting inverters), for a total of 9n gates.
- In contrast, in (b), the bus multiplexer requires only 3n AND gates and n OR gates, for a total of 4n gates.
- Also, the data input connections to the multiplexers are reduced from 6n to 3n.
- Thus, the cost of the selection hardware is reduced by about half.

Three-State Buffer and Three-State Bus

- A bus can be constructed with three-state buffers instead of multiplexers.
- This has the potential for additional reductions in the number of connections.
- But why use three-state buffers instead of a multiplexer, particularly for implementing buses?
- The reason is that many three-state buffer outputs can be connected together to form a bit line of a bus, and this bus is implemented using only one level of logic gates.



(a) Logic symbol

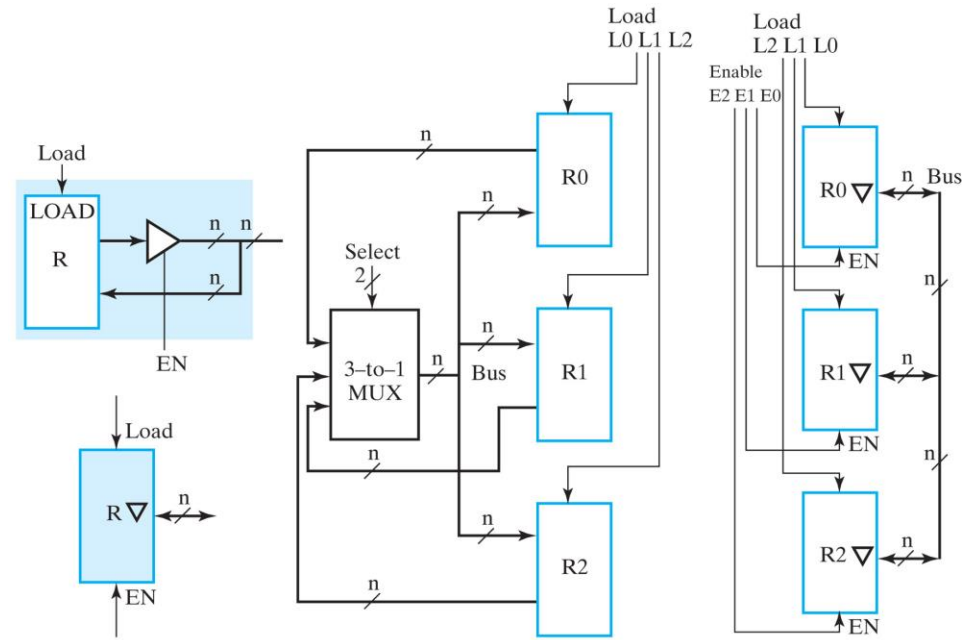
EN	IN	OUT
0	X	Hi-Z
1	0	0
1	1	1

(b) Truth table

- On the other hand, in a multiplexer, such a large number of sources means a high fan-in OR, which requires multiple levels of OR gates, introducing more logic and increasing delay.
- In contrast, three-state buffers provide a practical way to construct fast buses with many sources, so they are often preferred in such cases.

Three-State Buffer and Three-State Bus

- More important, however, is the fact that signals can travel in two directions on a three-state bus.
- Thus, the three-state bus can use the same interconnection to carry signals into and out of a logic circuit.
- This feature, which is most important when crossing chip boundaries, is illustrated in (a).
- The figure shows a register with n lines that serve as both inputs and outputs lying across the boundary of the shaded area.
- If the three-state buffers are enabled, then the lines are outputs; if the three-state buffers are disabled, then the lines can be inputs.
- The symbol for this structure is also given in the figure.
- Note that the bidirectional bus lines are represented by a two-headed arrow. Also, a small inverted triangle denotes the three-state outputs of the register.
- (b) and (c) show a multiplexer-implemented bus and a three-state bus, respectively, for comparison.
- The symbol from (a) for a register with bidirectional input–output lines is used in (c).
- In contrast to the situation in previous figure, where dedicated multiplexers were replaced by a bus, these two implementations are identical in terms of their register–transfer capability.



(a) Register with bidirectional input–output lines and symbol

(b) Multiplexer bus

(c) Three-state bus using registers with bidirectional lines

Three-State Buffer and Three-State Bus

- Note that, in the three-state bus, there are only three data connections to the set of register blocks for each bit of the bus.
- The multiplexer-implemented bus has six data connections per bit to the set of register blocks.
- This reduction in the number of data connections by half, along with the ability to easily construct a bus with many sources, makes the three-state bus an attractive alternative.
- The use of such bidirectional input–output lines is particularly effective between logic circuits in different physical packages.