Digital Logic Design

Chapter 7 – Register & Register Transfers(II)

Counters, Register Cells, Buses & Serial Operations

浙江大学计算机学院　王总辉
zhwang@zju. edu. cn
http://course.zju.edu.cn

2021年10月

Overview

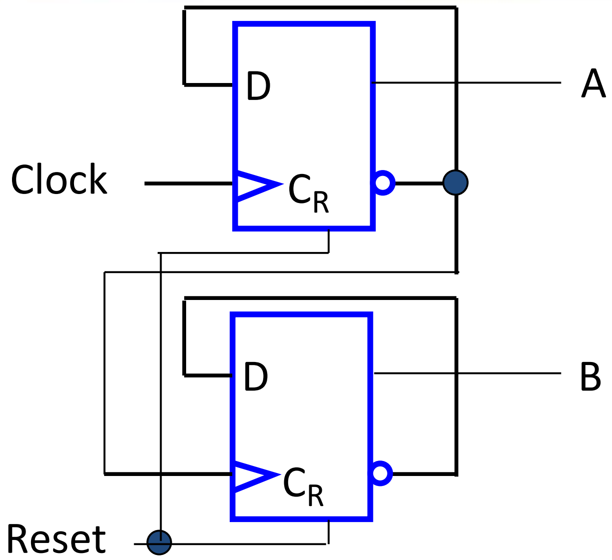
- **Part 1 – Registers, Microoperations and Implementations**
- **Part 2 – Counters, register cells, buses, & serial operations**
  - Microoperations on single register (continued)
    - Counters
  - Register cell design
  - Multiplexer and bus-based transfers for multiple registers
  - Serial transfers and microoperations
- **Part 3 – Control of Register Transfers**

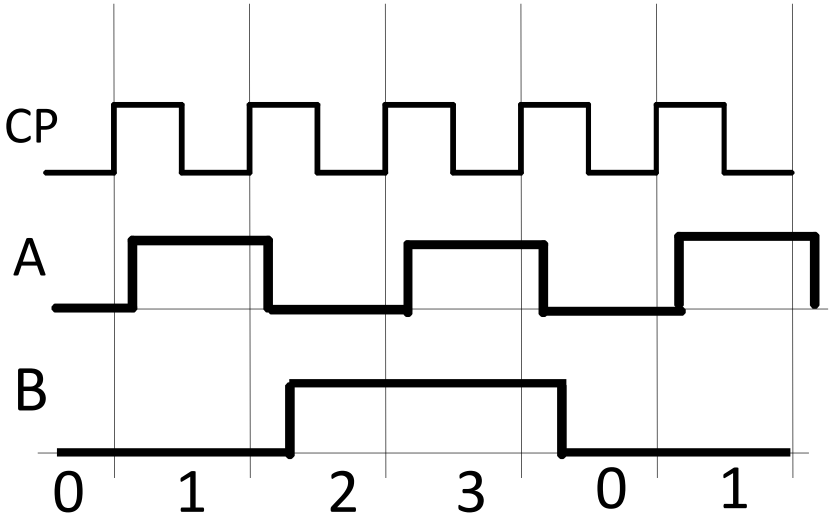
Counters

- **Counters are sequential circuits which "count" through a specific state sequence. They can count up, count down, or count through other fixed_sequences. Two distinct types are in common usage:**
- **Ripple Counters**
  - □ Clock connected to the flip-flop clock input on the LSB bit flip-flop
  - □ For all other bits, a flip-flop output is connected to the clock input, thus circuit is not truly synchronous!
  - □ Output change is delayed more for each bit toward the MSB.
  - □ Resurgent because of low power consumption
- **Synchronous Counters**
  - □ Clock is directly connected to the flip-flop clock inputs
  - □ Logic is used to implement the desired state sequencing

Ripple Counter

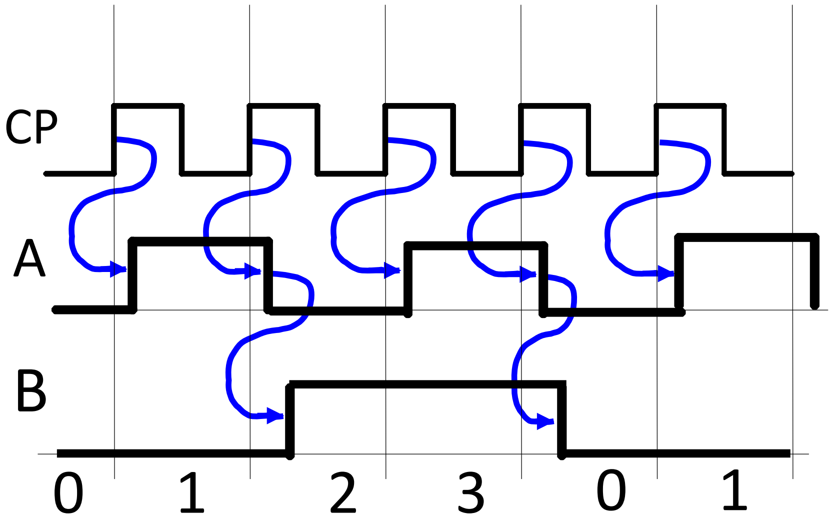
- **How does it work?**
  - When there is a positive edge on the clock input of A, A complements
  - The clock input for flip-flop B is the complemented output of flip-flop A
  - When flip A changes from 1 to 0, there is a positive edge on the clock input of B causing B to complement





Ripple Counter (continued)

- **The arrows show the cause-effect relationship from the prior slide =>**
- **The corresponding sequence of states =>**



(B,A) = (0,0), (0,1), (1,0), (1,1), (0,0), (0,1), …

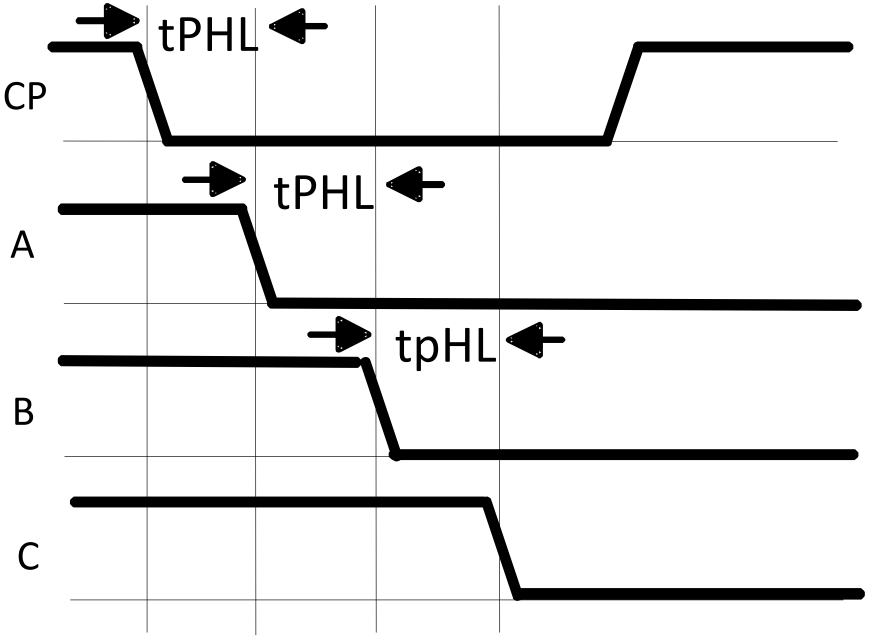
- **Each additional bit, C, D, …behaves like bit B, changing half as frequently as the bit before it.**
- **For 3 bits: (C,B,A) = (0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1), (0,0,0), …**

Ripple Counter (continued)

- These circuits are called *ripple counters* because each edge sensitive transition (positive in the example) causes a change in the next flip-flop’s state.
- The changes “ripple” upward through the chain of flip-flops, i. e., each transition occurs after a clock-to-output delay from the stage before.
- To see this effect in detail look at the waveforms on the next slide.

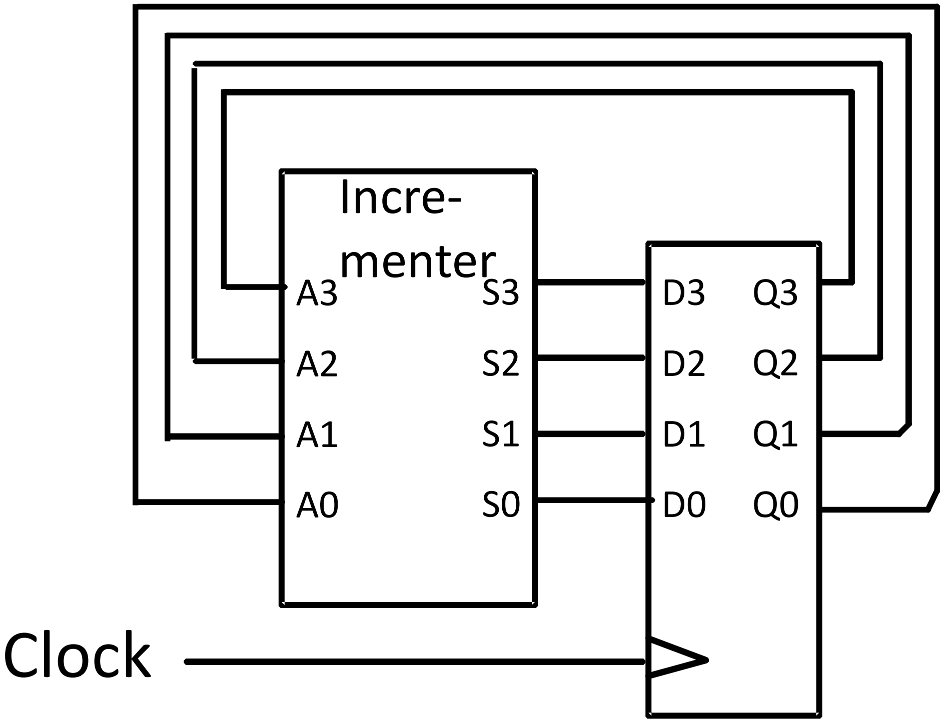
Ripple Counter (continued)

- **Starting with C = B = A = 1, equivalent to (C,B,A) = 7 base 10, the next clock increments the count to (C,B,A) = 0 base 10. In fine timing detail:**
  - The clock to output delay t_{PHL} causes an increasing delay from clock edge for each stage transition.
  - Thus, the count “ripples” from least to most significant bit.
  - For *n* bits, total worst case delay is *n* t_{PHL}.



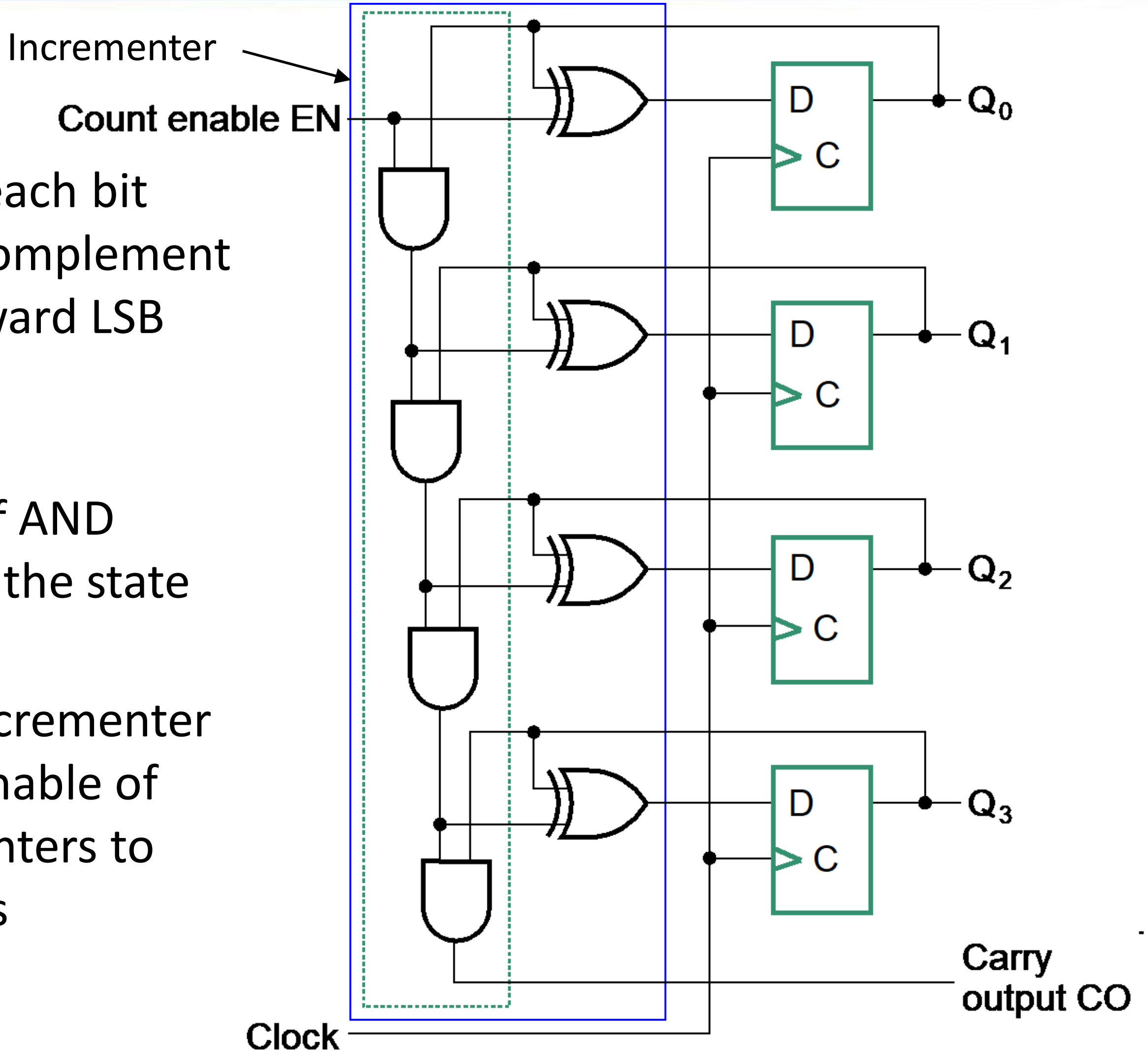
Synchronous Counters

- To eliminate the "ripple" effects, use a common clock for each flip-flop and a combinational circuit to generate the next state.
- For an up-counter, use an incrementer =>



Synchronous Counters (continued)

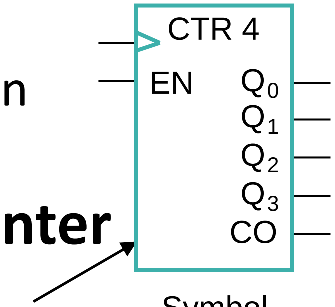
- **Internal details =>**
- **Internal Logic**
  - XOR complements each bit
  - AND chain causes complement of a bit if all bits toward LSB from it equal 1
- **Count Enable**
  - Forces all outputs of AND chain to 0 to "hold" the state
- **Carry Out**
  - Added as part of incrementer
  - Connect to Count Enable of additional 4-bit counters to form larger counters



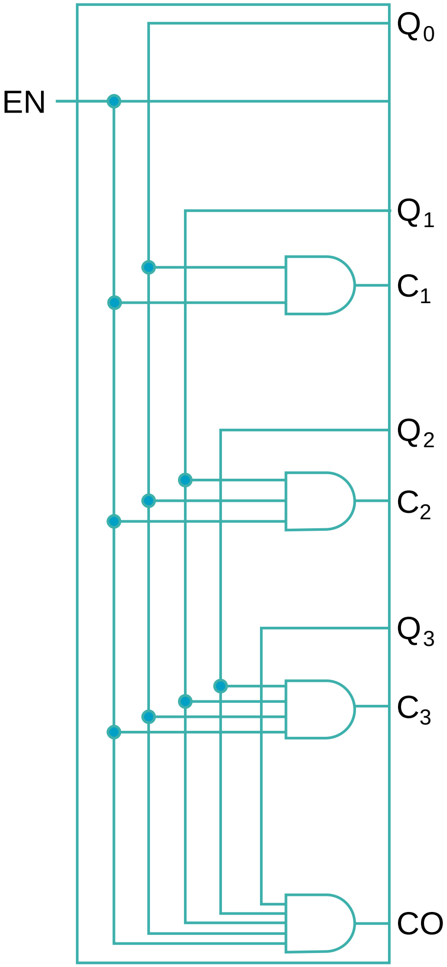
(a) Logic Diagram-Serial Gating

Synchronous Counters (continued)

- **Carry chain**
  - series of AND gates through which the carry “ripples”
  - Yields long path delays
  - Called *serial gating*
- **Replace AND carry chain with ANDs => in parallel**
  - Reduces path delays
  - Called *parallel gating*
  - Like carry lookahead
  - Lookahead can be used on COs and ENs to prevent long paths in large counters
- **Symbol for Synchronous Counter**



Symbol



Logic Diagram-Parallel Gating

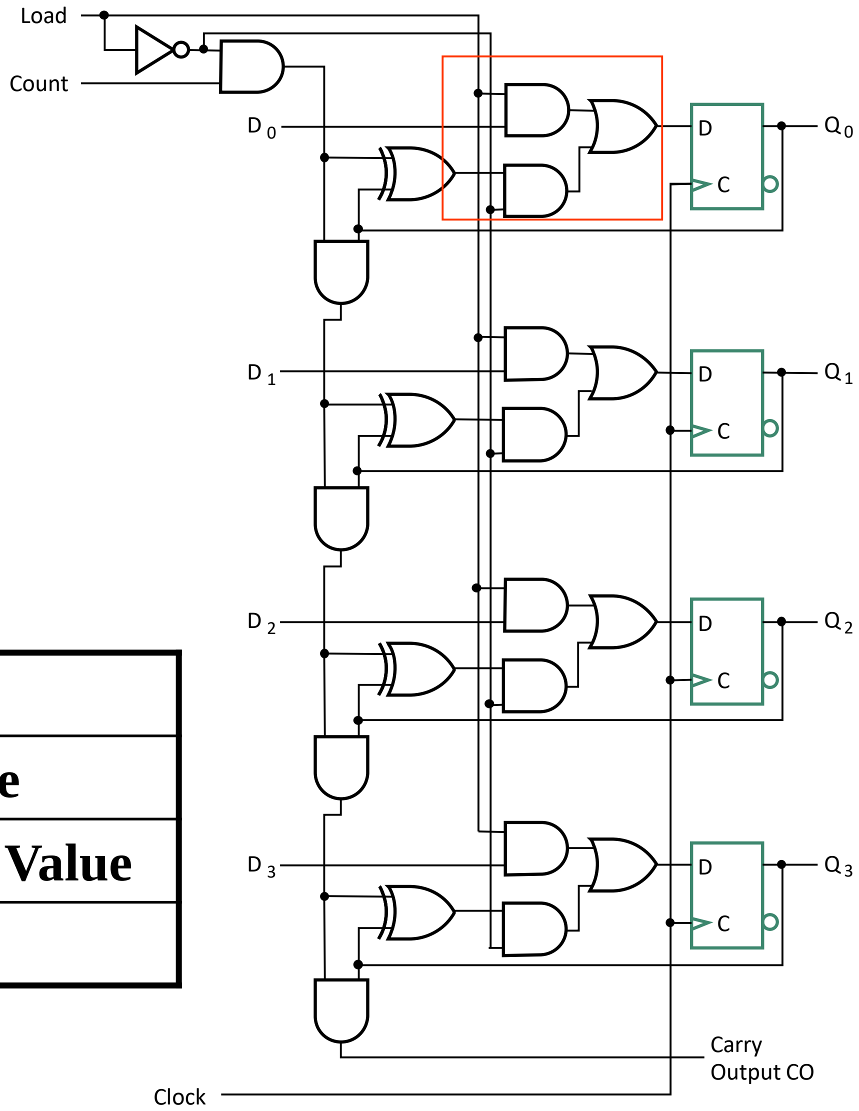
Other Counters

- **See text for:**
  - □ *Down Counter* - counts downward instead of upward
  - □ *Up-Down Counter* - counts up or down depending on value a control input such as Up/$\overline{\text{Down}}$
  - □ *Parallel Load Counter* - Has parallel load of values available depending on control input such as Load
- ***Divide-by-n (Modulo n) Counter***
  - □ Count is remainder of division by *n; n* may not be a power of 2 or
  - □ Count is arbitrary sequence of *n* states specifically designed state-by-state
  - □ Includes modulo 10 which is the *BCD counter*

Counter with Parallel Load

- **Add path for input data**
  - □ enabled for Load = 1
- **Add logic to:**
  - □ disable count logic for Load = 1
  - □ disable feedback from outputs for Load = 1
  - □ enable count logic for Load = 0 and Count = 1
- **The resulting function table:**

Load	Count	Action
0	0	Hold Stored Value
0	1	Count Up Stored Value
1	X	Load D



Design Example: Synchronous BCD

- **Use the sequential logic model to design a synchronous BCD counter with D flip-flops**
- **Input combinations 1010 through 1111 are don't cares**

State Table

Current State Q8 Q4 Q2 Q1	Next State Q8 Q4 Q2 Q1
0 0 0 0	0 0 0 1
0 0 0 1	0 0 1 0
0 0 1 0	0 0 1 1
0 0 1 1	0 1 0 0
0 1 0 0	0 1 0 1
0 1 0 1	0 1 1 0
0 1 1 0	0 1 1 1
0 1 1 1	1 0 0 0
1 0 0 0	1 0 0 1
1 0 0 1	0 0 0 0

Q2n+1

Q8 Q4 \ Q2 Q1	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	x	x	x	x
10	0	0	x	x

Q4n+1

Q8 Q4 \ Q2 Q1	00	01	11	10
00	0	0	1	0
01	1	1	0	1
11	x	x	x	x
10	0	0	x	x

$$Q1(n+1) = \overline{Q1}$$

$$Q2(n+1) = Q2 \oplus Q1\overline{Q8}$$

$$Q4(n+1) = Q4 \oplus Q1Q2$$

$$Q8(n+1) = Q8 \oplus (Q1Q8 + Q1Q2Q4)$$

Synchronous BCD (continued)

- **Use K-Maps to two-level optimize the next state equations and manipulate into forms containing XOR gates:**

  $$D_1 = \overline{Q_1}$$

  $$D_2 = Q_2 \oplus Q_1\overline{Q_8}$$

  $$D_4 = Q_4 \oplus Q_1Q_2$$

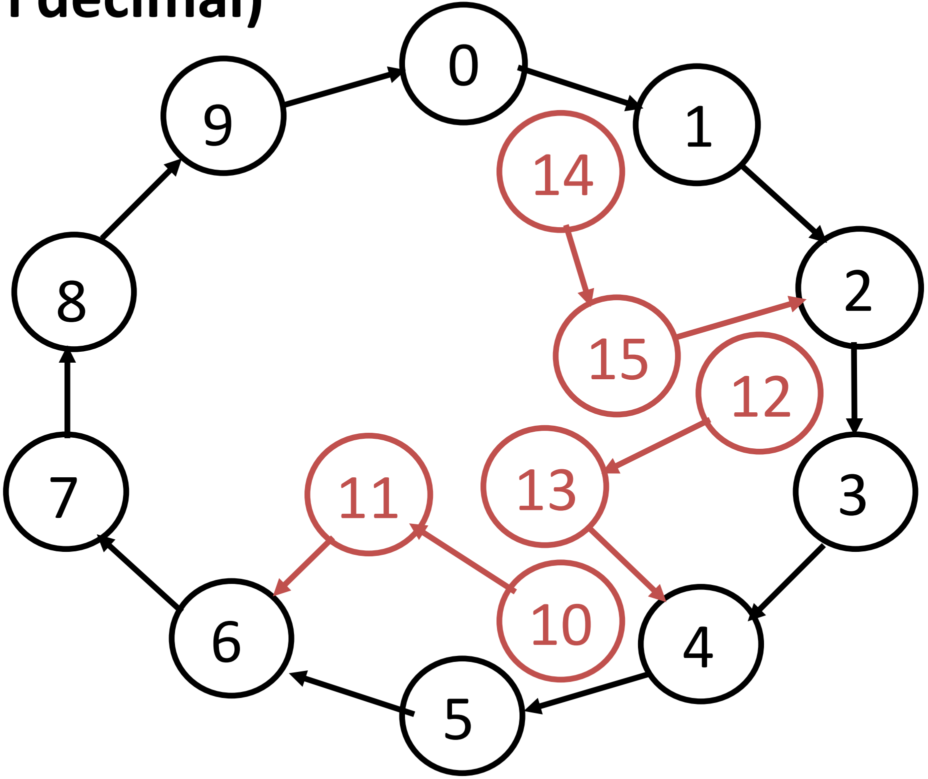
  $$D_8 = Q_8 \oplus (Q_1Q_8 + Q_1Q_2Q_4)$$

- **The logic diagram can be draw from these equations**
  - An asynchronous or synchronous reset should be added
- **What happens if the counter is perturbed by a power disturbance or other interference and it enters a state other than 0000 through 1001?**

Synchronous BCD (continued)

- **Find the actual values of the six next states for the don't care combinations from the equations**
- **Find the overall state diagram to assess behavior for the don't care states (states in decimal)**

Present State				Next State			
Q8	Q4	Q2	Q1	Q8	Q4	Q2	Q1
1	0	1	0	1	0	1	1
1	0	1	1	0	1	1	0
1	1	0	0	1	1	0	1
1	1	0	1	0	1	0	0
1	1	1	0	1	1	1	1
1	1	1	1	0	0	1	0



Synchronous BCD (continued)

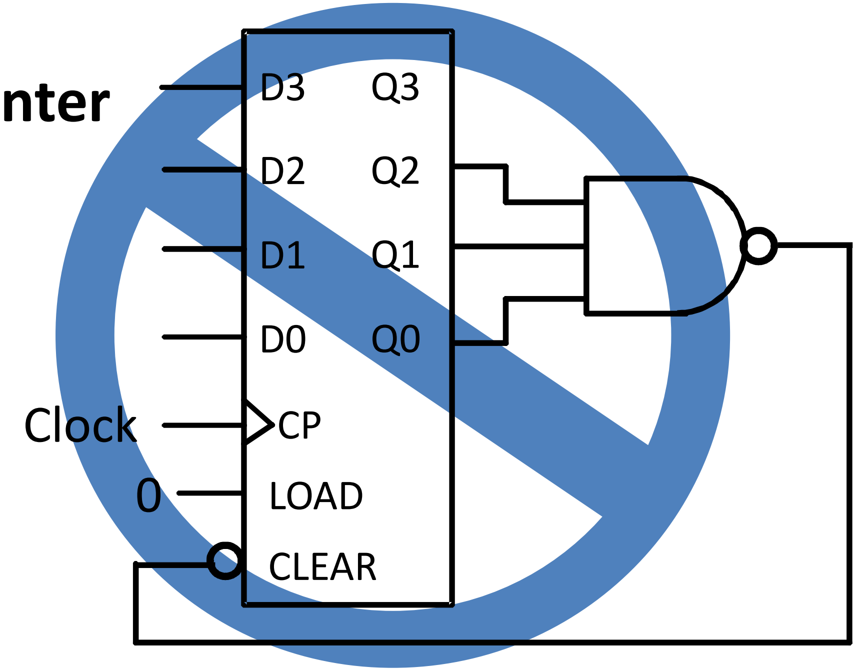
- **For the BCD counter design, if an invalid state is entered, return to a valid state occurs within two clock cycles**
- **Is this adequate? If not:**
  - Is a signal needed that indicates that an invalid state has been entered? What is the equation for such a signal?
  - Does the design need to be modified to return from an invalid state to a valid state in one clock cycle?
  - Does the design need to be modified to return from a invalid state to a specific state (such as 0)?
- **The action to be taken depends on:**
  - the application of the circuit
  - design group policy
- **See pages 244 of the text.**

Counting Modulo N

- **The following techniques use an *n*-bit binary counter with asynchronous or synchronous clear and/or parallel load:**
  - □ Detect a *terminal count* of N in a Modulo-N count sequence to asynchronously Clear the count to 0 or asynchronously Load in value 0 (These lead to counts which are present for only a very short time and can fail to work for some timing conditions!)
  - □ Detect a terminal count of N - 1 in a Modulo-N count sequence to Clear the count synchronously to 0
  - □ Detect a terminal count of N - 1 in a Modulo-N count sequence to synchronously Load in value 0
  - □ Detect a terminal count and use Load to preset a count of the terminal count value minus (N - 1)
- **Alternatively, custom design a modulo N counter as done for BCD**

Counting Modulo 7: Detect 7 and Asynchronously Clear

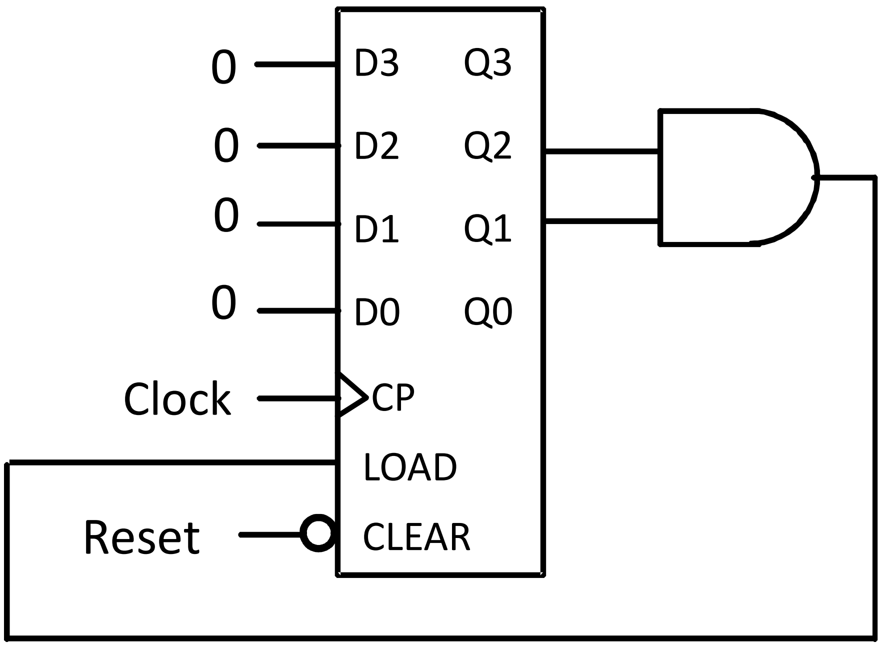
- **A synchronous 4-bit binary counter with an asynchronous Clear is used to make a Modulo 7 counter.**
- **Use the Clear feature to detect the count 7 and clear the count to 0. This gives a count of 0, 1, 2, 3, 4, 5, 6, 7(short)0, 1, 2, 3, 4, 5, 6, 7(short)0, etc.**



- **DON'T DO THIS! Existence of state 7 may not be long enough to reliably reset all flip-flops to 0. Referred to as a "suicide" counter! (Count "7" is "killed," but the designer's job may be dead as well!)**

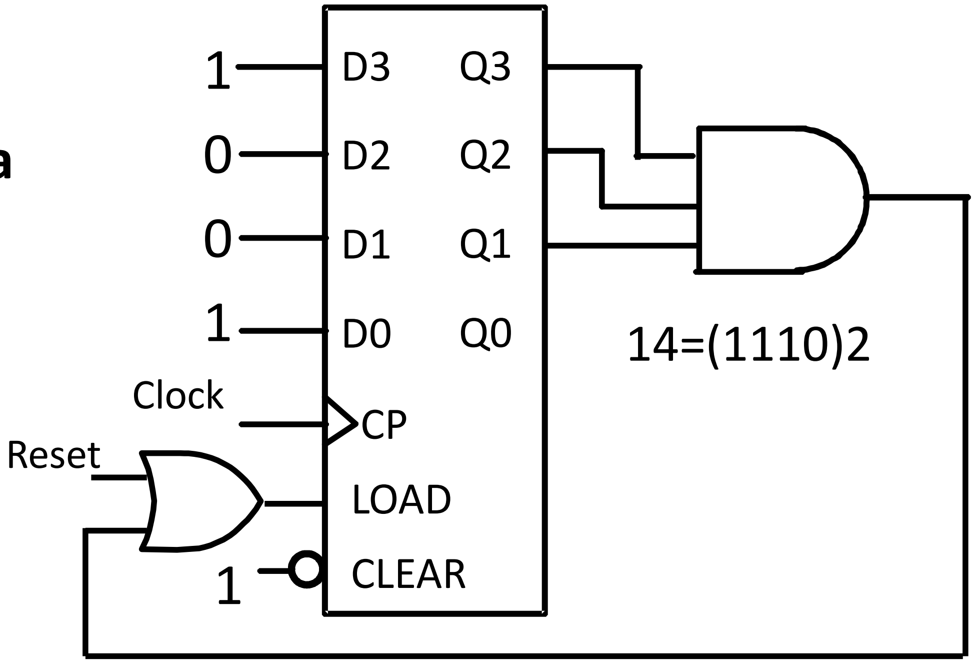
Counting Modulo 7: Synchronously Load on Terminal Count of 6(N-1)

- A synchronous 4-bit binary counter with a synchronous load and an asynchronous clear is used to make a Modulo 7 counter
- Use the Load feature to detect the count "6" and load in "zero". This gives a count of 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, ...
- Using don't cares for states above 0110, detection of 6 can be done with Load = Q_2 Q_1



Counting Modulo 6: Synchronously Preset 9 on Reset and Load 9 on Terminal Count 14

- A synchronous, 4-bit binary counter with a synchronous Load is to be used to make a Modulo 6 counter.
- Use the Load feature to preset the count to 9 on Reset and detection of count 14.



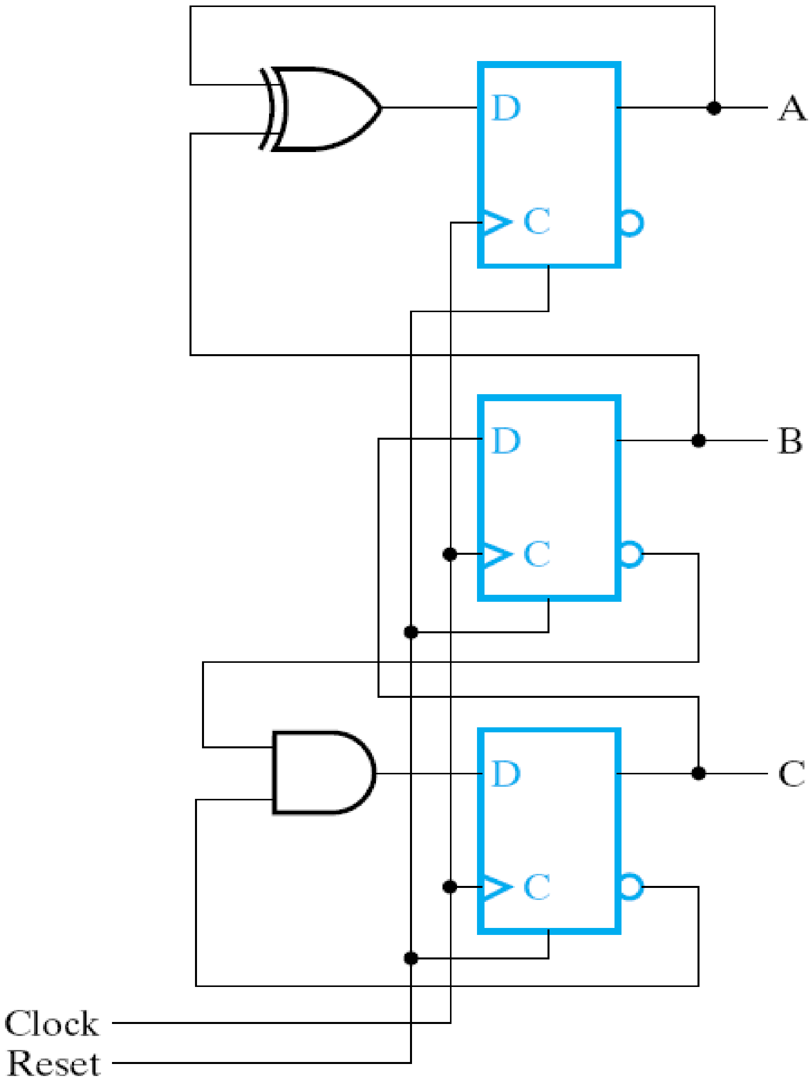
- This gives a count of 9, 10, 11, 12, 13, 14, 9, 10, 11, 12, 13, 14, 9, …
- If the terminal count is 15 detection is usually built in as Carry Out (CO)

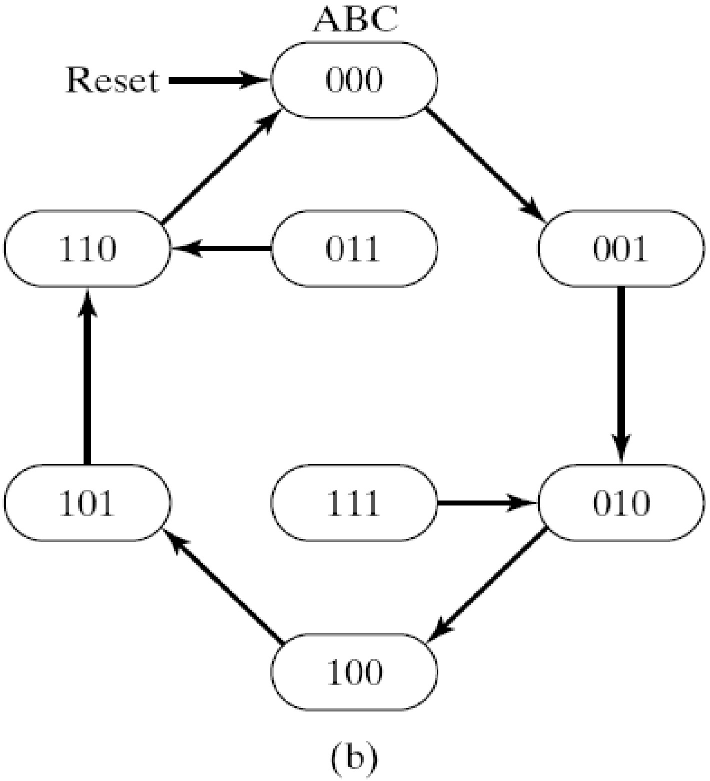
Fixed Sequences Modulo 6

$$D_A = A \oplus B$$

$$D_B = C$$

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





(b)

Register Cell Design

- **Assume that a register consists of identical cells**
- **Then register design can be approached as follows:**
  - Design representative cell for the register
  - Connect copies of the cell together to form the register
  - Applying appropriate “boundary conditions” to cells that need to be different and contract if appropriate
- **Register cell design is the first step of the above process**

Register Cell Specifications

- **A register**
- **Data inputs to the register**
- **Control input combinations to the register**
  - Example 1: Not encoded
    - Control inputs: Load, Shift, Add
    - At most, one of Load, Shift, Add is 1 for any clock cycle
      (0,0,0), (1,0,0), (0,1,0), (0,0,1)
  - Example 2: Encoded
    - Control inputs: S1, S0
    - All possible binary combinations on S1, S0
      (0,0), (0,1), (1,0), (1,1)

Register Cell Specifications

- **A set of register functions (typically specified as register transfers)**
  - Example:

    Load: A ← B

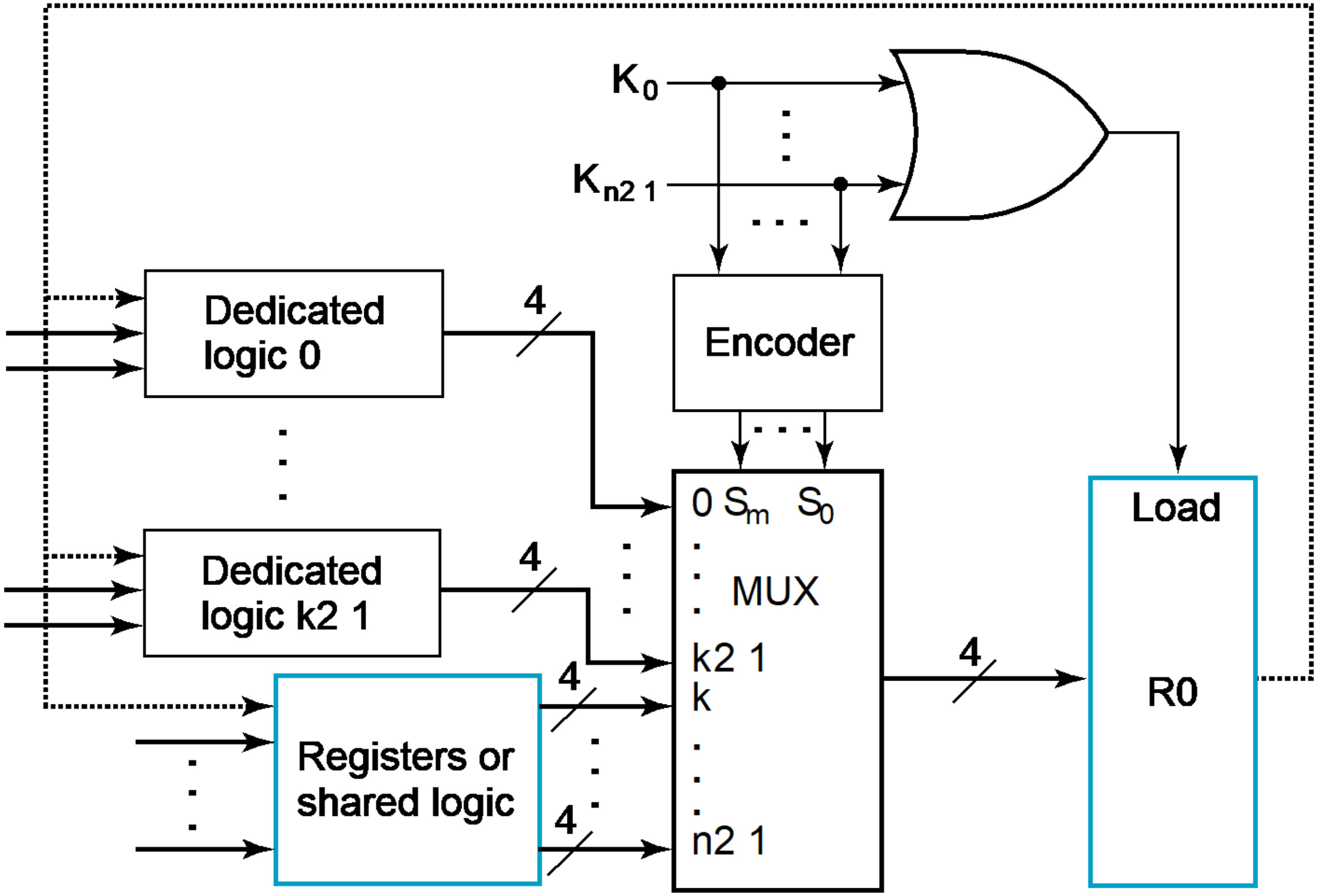
    Shift: A ← sr B

    Add: A ← A + B

- **A hold state specification**
  - Example:
    - Control inputs: Load, Shift, Add
    - If all control inputs are 0, hold the current register state

Multiplexer Approach

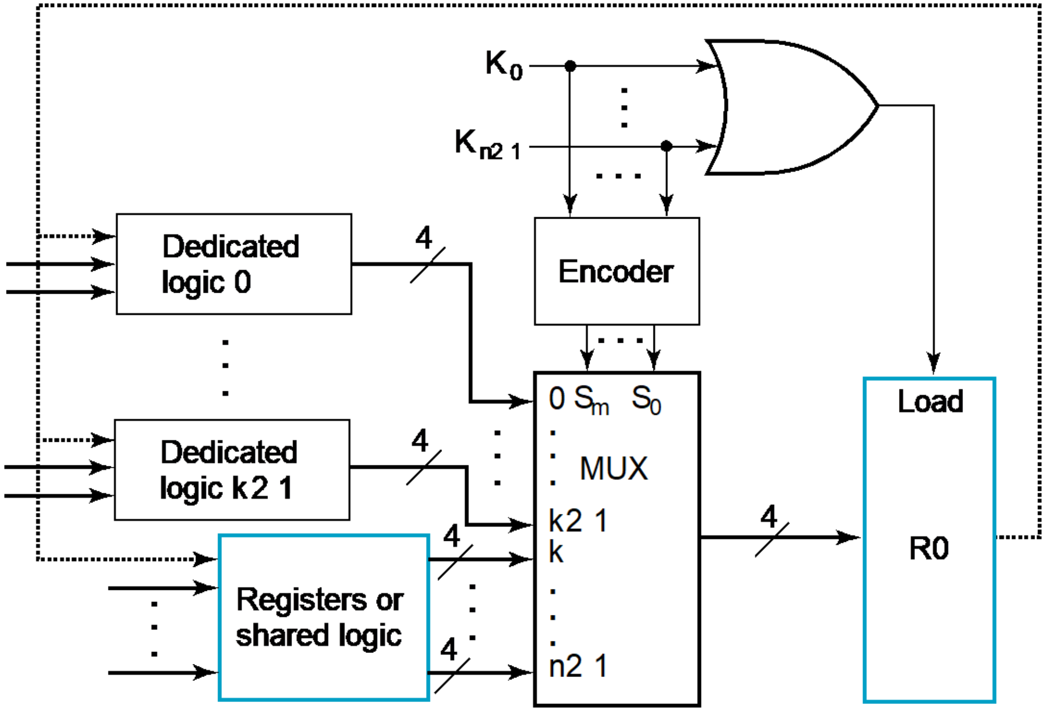
- **Uses an n-input multiplexer with a variety of transfer sources and functions**



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



Example 1: Register Cell Design

- **Register A (m-bits) Specification:**
  - Data input: B
  - Control inputs (CX, CY)
  - Control input combinations (0,0), (0,1) (1,0)
  - Register transfers:
  - CX: A ← B v A
  - CY :A ← B⊕A
  - Hold state: (0,0)

Example 1: Register Cell Design (continued)

- **Load Control**

  Load = CX + CY

- **Since all control combinations appear as if encoded (0,0), (0,1), (1,0) can use multiplexer without encoder:**

  S1 = CX
  S0 = CY

D0 = A_i	**Hold A**
D1 = $A_i \leftarrow B_i \oplus A_i$	**CY = 1**
D2 = $A_i \leftarrow B_i \vee A_i$	**CX = 1**

- **Note that the decoder part of the 3-input multiplexer can be shared between bits if desired**

Sequential Circuit Design Approach

- **Find a state diagram or state table**
  - Note that there are only two states with the state assignment equal to the register cell output value
- **Use the design procedure in Chapter 5 to complete the cell design**
- **For optimization:**
  - Use K-maps for up to 4 to 6 variables
  - Otherwise, use computer-aided or manual optimization

Example 1 Again

- **State Table:**

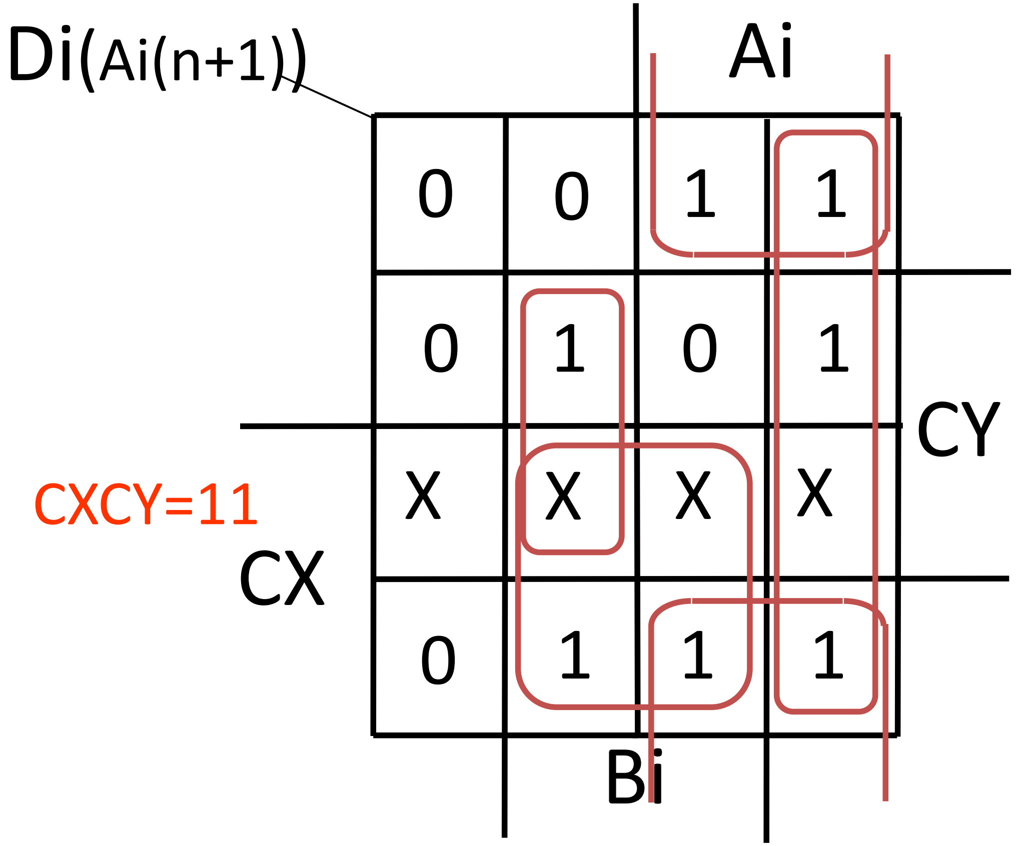
	Hold	Ai ∨ Bi		Ai⊕Bi	
A_i	CX = 0 CY = 0	CX = 1 B_i = 0	CX = 1 B_i = 1	CY = 1 B_i = 0	CY = 1 B_i = 1
0	0	0	1	0	1
1	1	1	1	1	0

- Four variables(CX、CY、A_i、B_i) give a total of 16 state table entries
- By using:
  - Combinations of variable names and values
  - Don’t care conditions (for CX = CY = 1)

only 8 entries are required to represent the 16 entries

Example 1 Again (continued)

- **K-map - Use variable ordering CX, CY, A_i B_i and assume a D flip-flop**



Di(Ai(n+1))

CXCY=11

0	0	1	1
0	1	0	1
X	X	X	X
0	1	1	1

Example 1 Again (continued)

- **The resulting SOP equation:**

  $$D_i = CX\,B_i + CY\,\overline{A_i}\,B_i + A_i\,\overline{B_i} + \overline{CY}\,A_i$$

- **Using factoring and DeMorgan's law:**

  $$D_i = CX\,B_i + \overline{A_i}\,(CY\,B_i) + A_i\overline{(CY\,B_i\,)}$$

  $$D_i = CX\,B_i + A_i \oplus (CY\,B_i)$$

  The gate input cost per cell = 2 + 8 + 2 + 2 = 14

- **The gate input cost per cell for the previous version is:**

  Per cell: 19

  Shared decoder logic: 8

- **Cost gain by sequential design > 5 per cell**
- **Also, no Enable on the flip-flop makes it cost less**

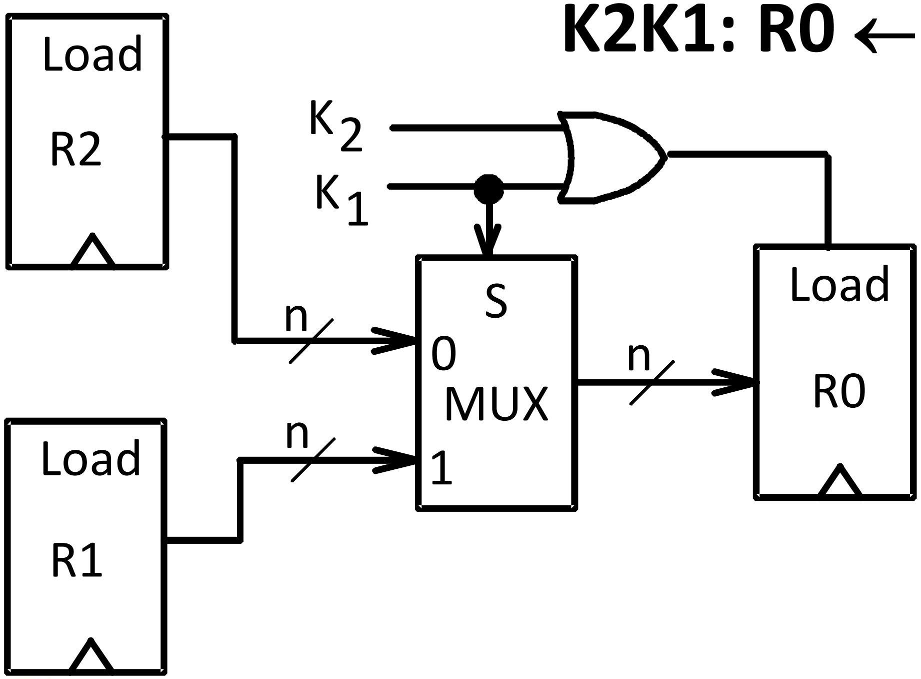
Register Transfer Structures

- 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)**
- **The transfers are: K1: R0 ← R1**

$K2\overline{K1}$: R0 ← R2

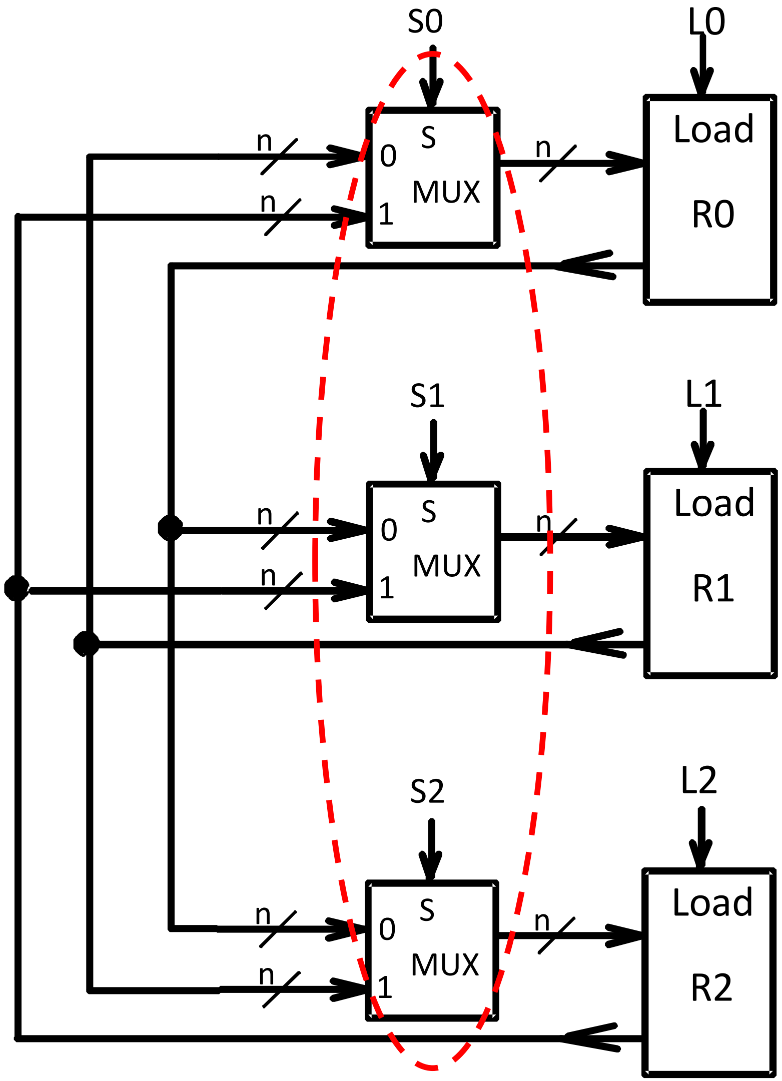


Multiplexer and Bus-Based Transfers for Multiple Registers

- **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**

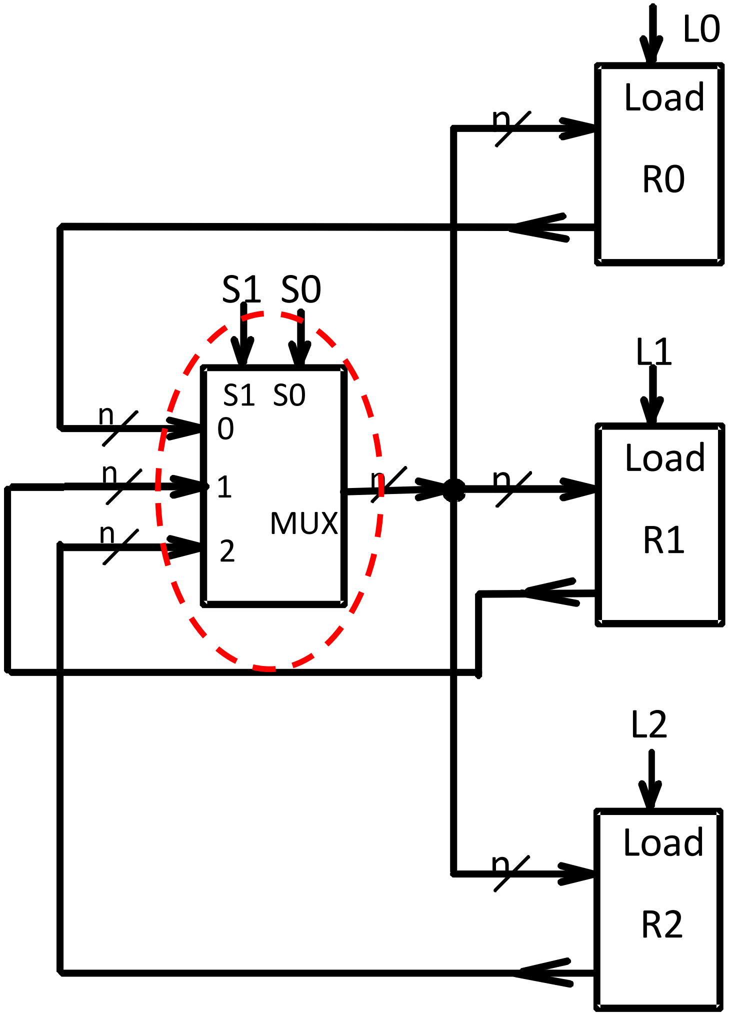
Dedicated MUX-Based Transfers

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



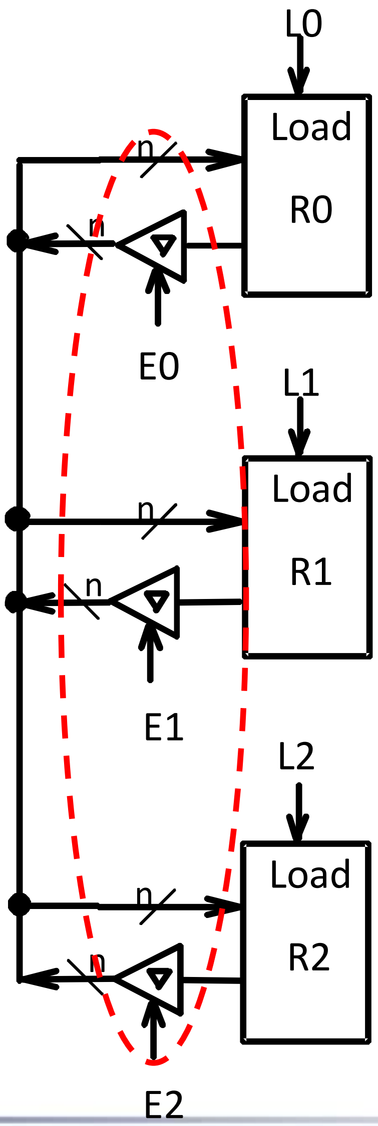
Multiplexer Bus

- **A single bus driven by a multiplexer lowers cost, but limits the available transfers =>**
- **Characterize the simultaneous transfers no 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 no possible with this structure.**
- **Characterize the cost savings and compare**
- **Other advantages?**



Serial Transfers and Microoperations

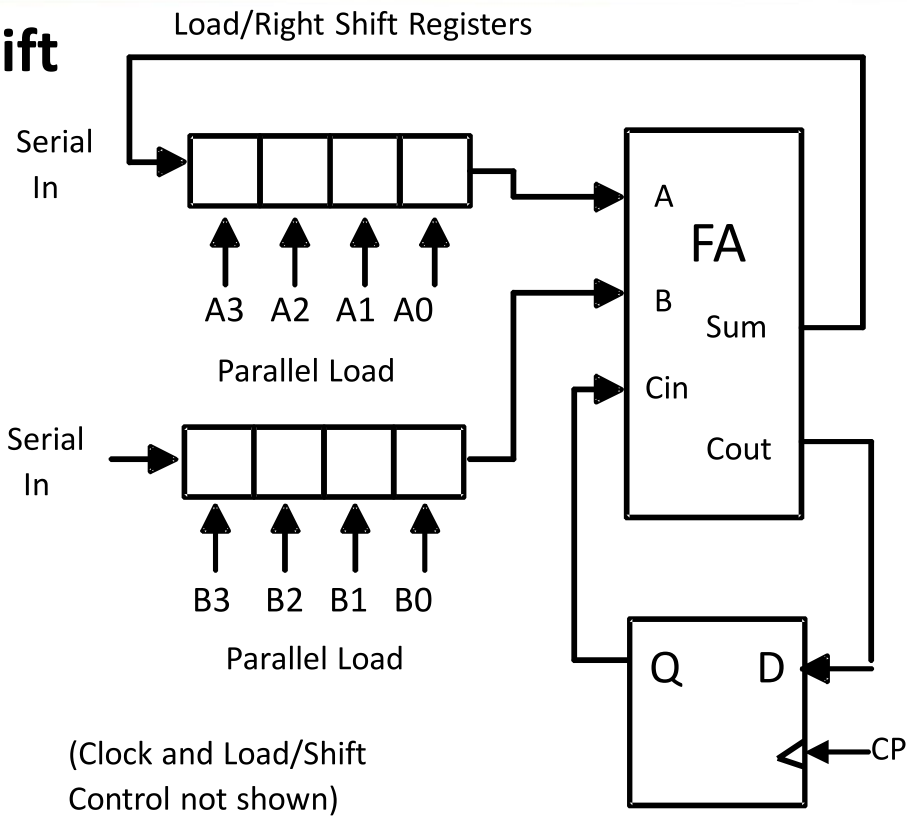
- **Serial Transfers**
  - Used for “narrow” transfer paths
  - Example 1: Telephone or cable line
    - Parallel-to-Serial conversion at source
    - Serial-to-Parallel conversion at destination
  - Example 2: Initialization and Capture of the contents of many flip-flops for test purposes
    - Add 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

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**

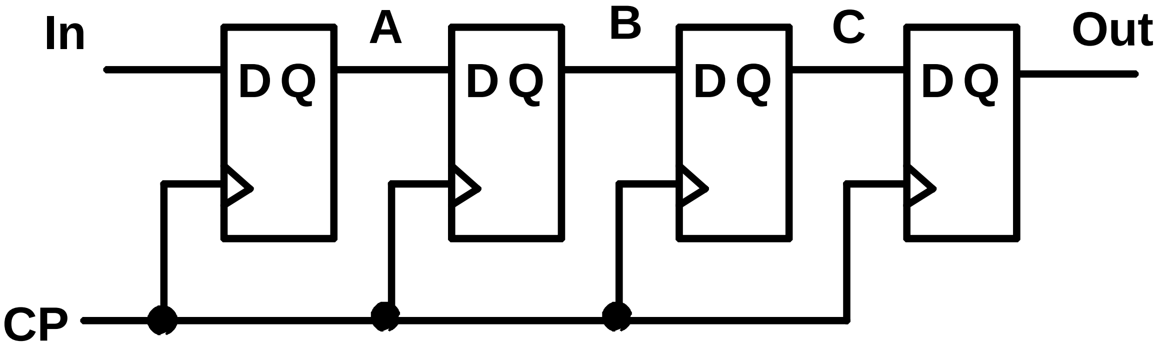


(Clock and Load/Shift Control not shown)

- **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.**

Shift Registers

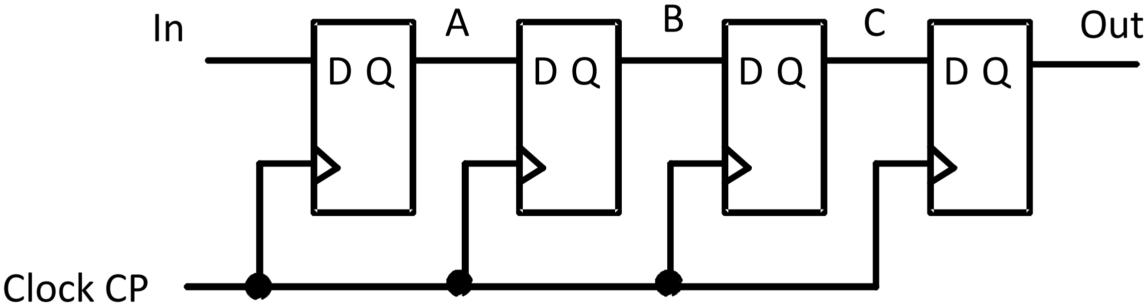
- **Shift Registers move data laterally within the register toward its MSB or LSB position**
- **In the simplest case, the shift register is simply a set of D flip-flops connected in a row like this:**



- **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*.**

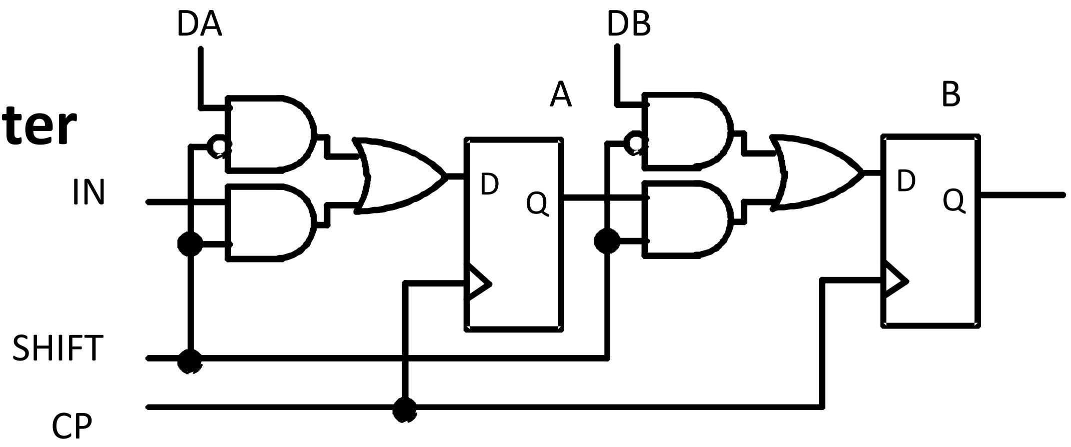
Shift Registers (continued)

- 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 “?”
- Complete the last three rows of the table

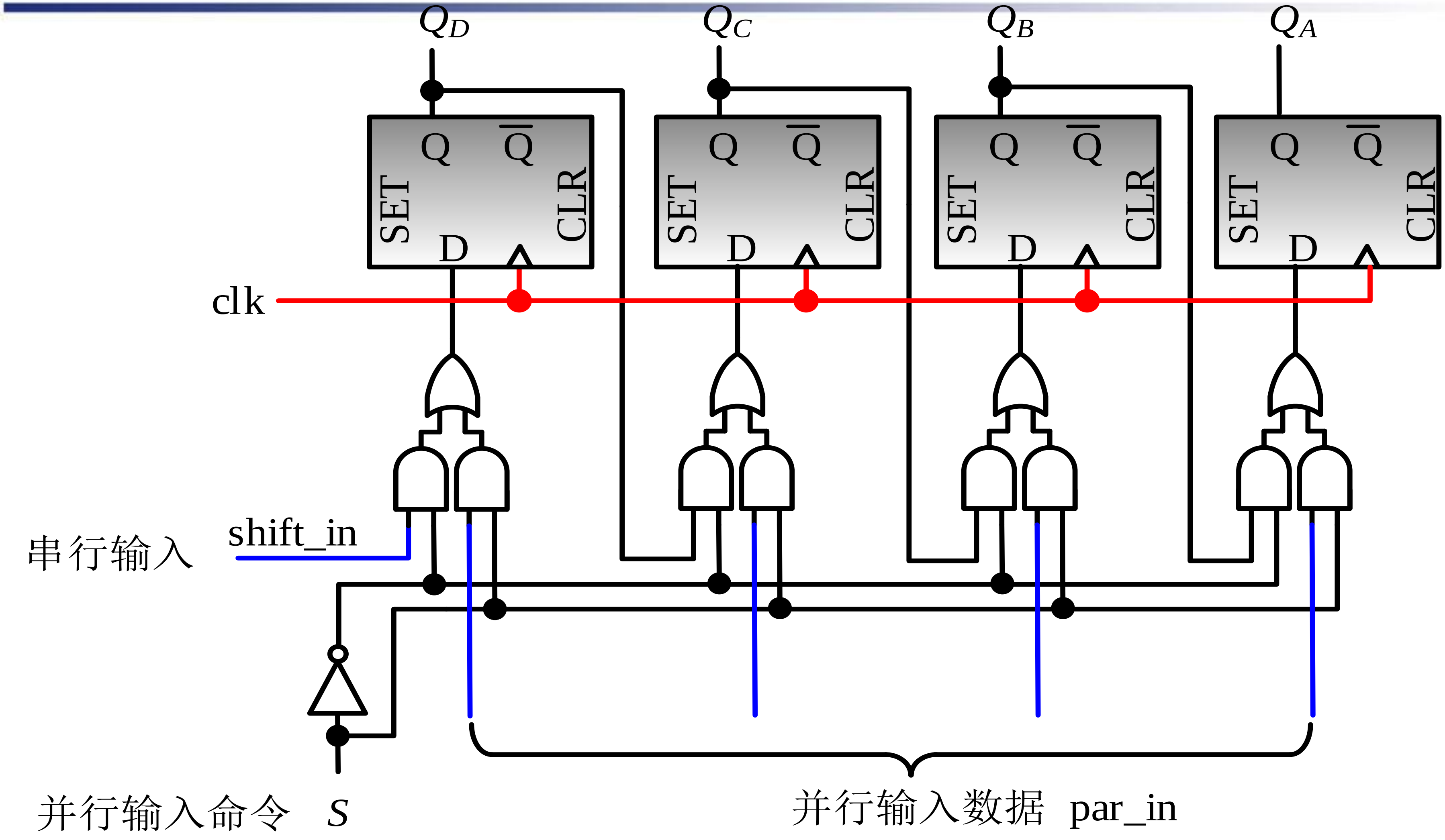


CP	In	A	B	C	Out
T0	0	?	?	?	?
T1	1	0	?	?	?
T2	1	1	0	?	?
T3	0	1	1	0	?
T4	1				
T5	1				
T6	1				

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


Q_D
Q_C
Q_B
Q_A
Q
$\overline{Q}$
SET
CLR
D
clk
串行输入
shift_in
并行输入命令 S
并行输入数据 par_in

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*.**

□谢 谢！