COMPUTER HARDWARE

Registers, Register Transfers and Counters

Overview

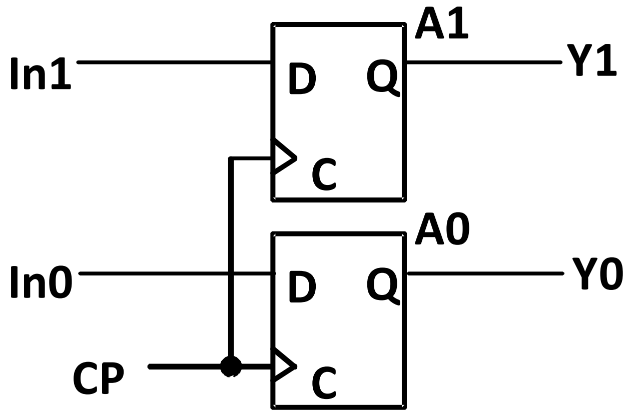
- Registers, Microoperations and Implementations
  - Registers and load enable
  - Register transfer operations
  - Microoperations - arithmetic, logic, and shift
  - Microoperations on a single register
    - Multiplexer-based transfers
    - Shift registers
- Register Cells, Buses, & Serial Operations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

Registers

- Register – a <u>collection</u> 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

Example: 2-bit Register

- How many states are there?
- How many input combinations?
  Output combinations?
- What is the output function?
- What is the next state function?
- Moore or Mealy?

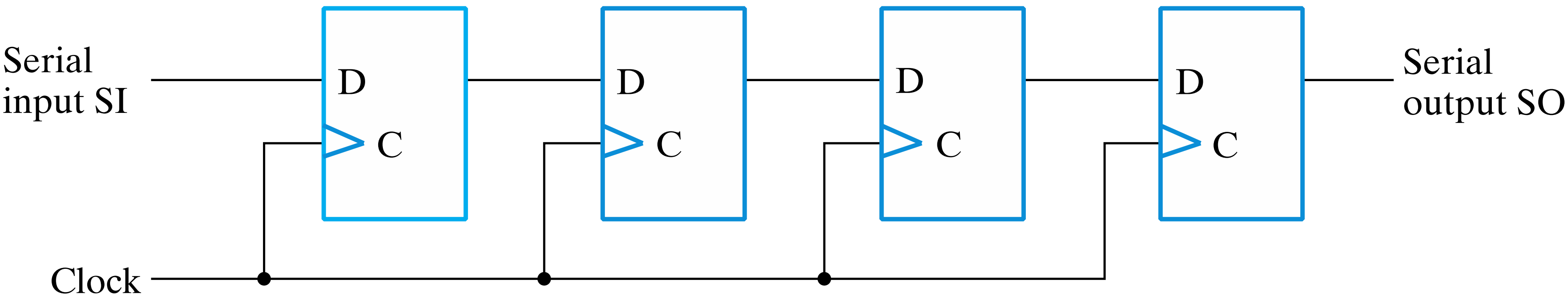


State Table:

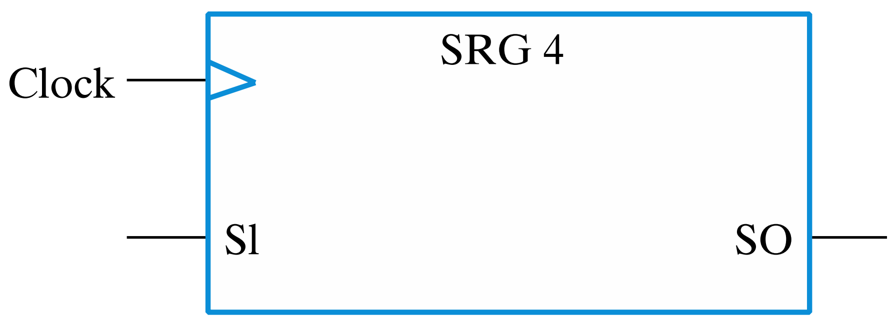
Current State A1 A0	Next State A1(t+1) A0(t+1) For In1 In0 = 00 01 10 11	Output (=A1 A0) Y1 Y0
0 0	00 01 10 11	0 0
0 1	00 01 10 11	0 1
1 0	00 01 10 11	1 0
1 1	00 01 10 11	1 1

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

Simple Shift Register



(a) Logic diagram



(b) Symbol

Register Design Models

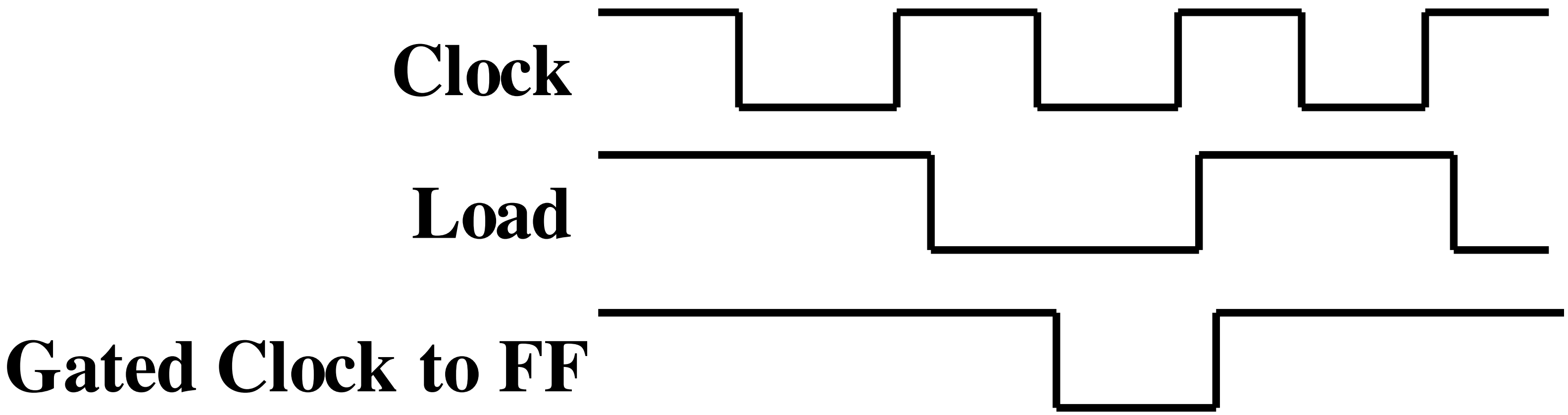
- Due to the large numbers of states and input combinations as *n* becomes large, the state diagram/state table model is not feasible!
- What are methods we can use to design registers?
  - Add predefined combinational circuits to registers
    - Example: To count up, connect the register flip-flops 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

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:
  - Use a signal to block the clock to the register,
  - Use a signal to control feedback of the output of the register back to its inputs, or
  - Use other SR or JK flip-flops, that for (0,0) applied, store their state
- <u>Load</u> is a frequent name for the signal that controls register storage and loading
  - Load = 1: Load the values on the data inputs
  - Load = 0: Store the values in the register

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:

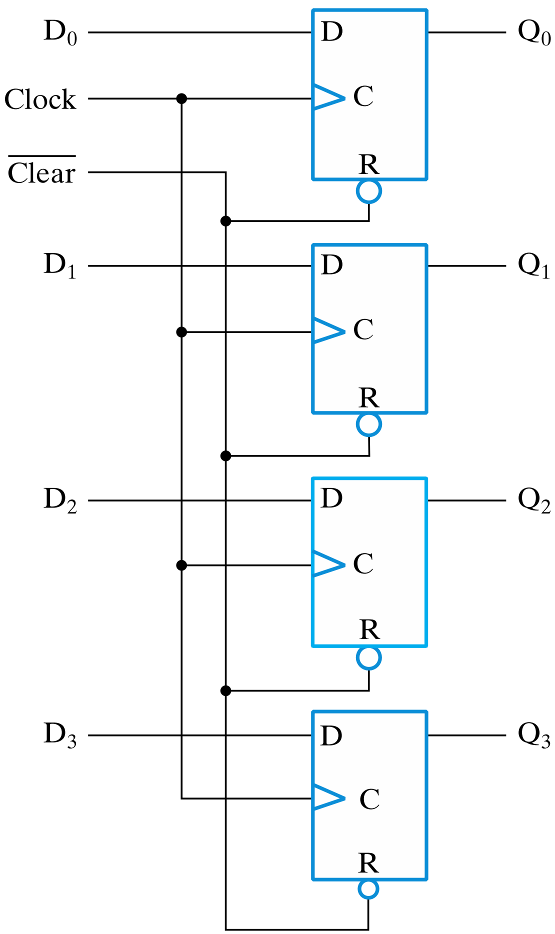


- What logic is needed for gating?
- What is the problem?

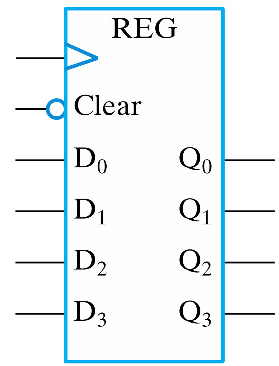
Gated Clock = Clock + $\overline{\text{Load}}$

Clock Skew of gated clocks with respect to clock or each other

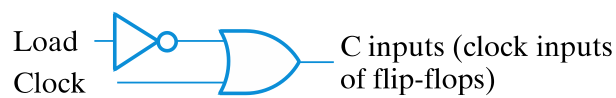
Registers with Clock Gating



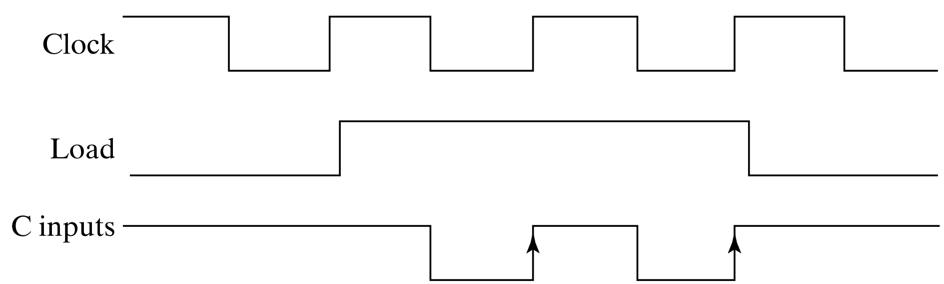
(a) Logic diagram



(b) Symbol



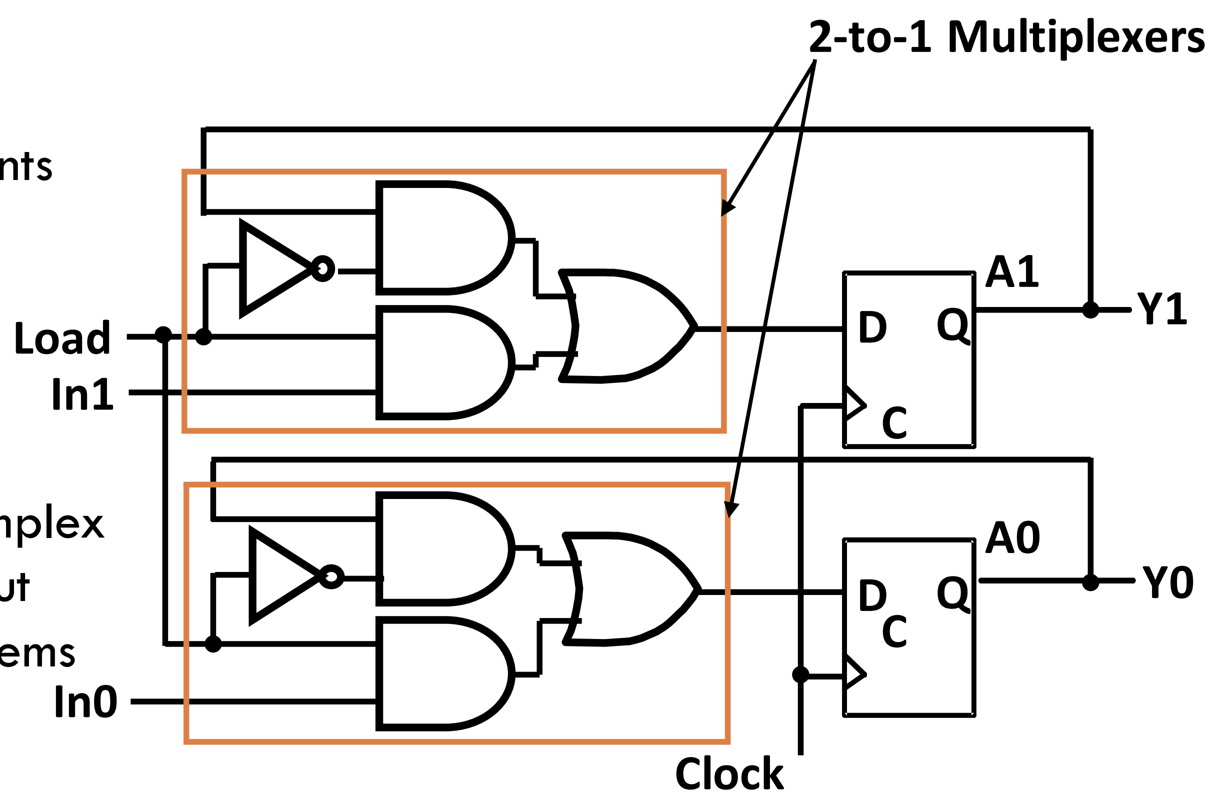
(c) Load control input



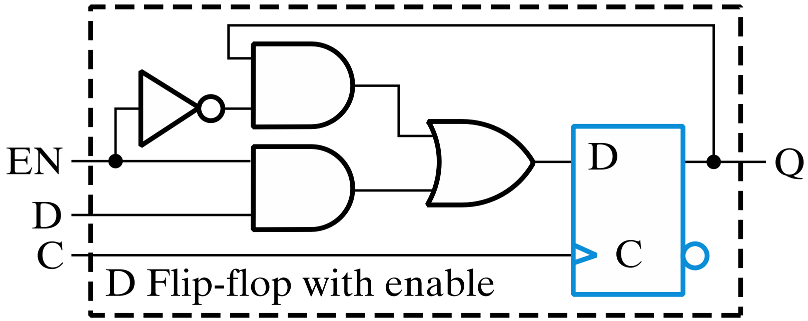
(d) Timing diagram

Registers with Load-Controlled Feedback

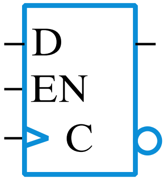
- A more reliable way to selectively load a register:
  - Run the clock continuously, and
  - Selectively use a load control to change the register contents.
- Example: 2-bit register with Load Control:
- For Load = 0, loads register contents (hold current values)
- For Load = 1, loads input values (load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



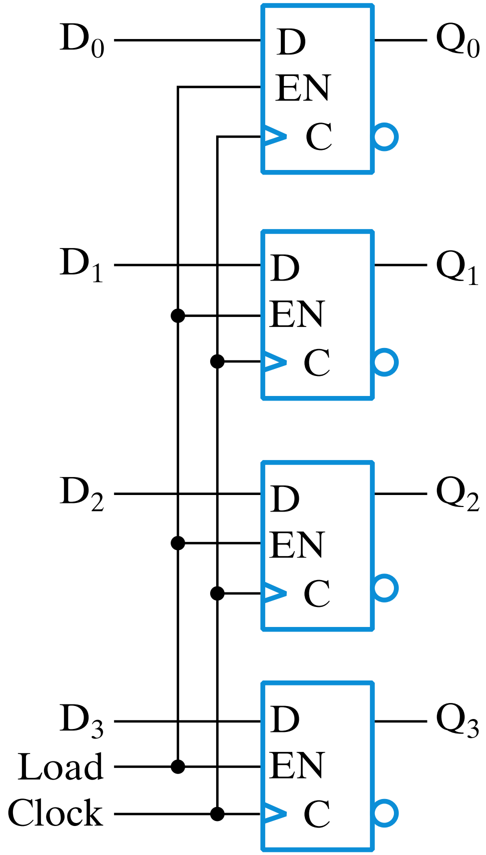
Registers with Load-Controlled Feedback



(a)



(b)

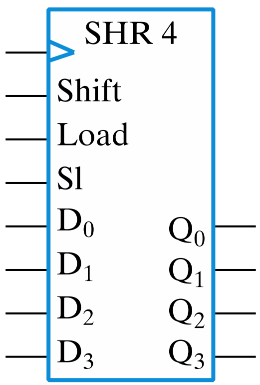


(c)

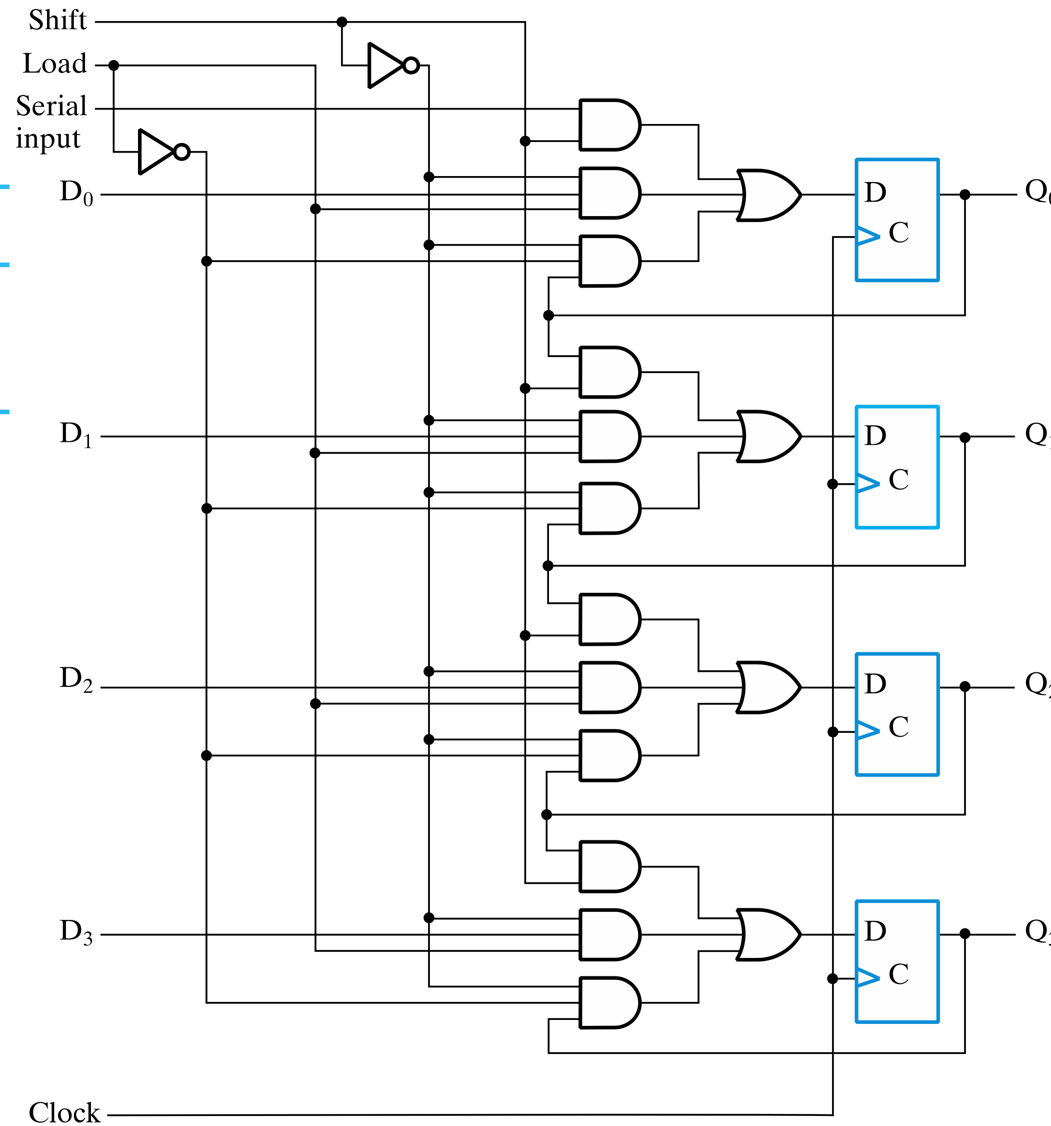
Registers

Function Table

Shift	Load	Operation
0	0	No change
0	1	Load parallel data
1	X	Shift down from Q_0 to Q_3



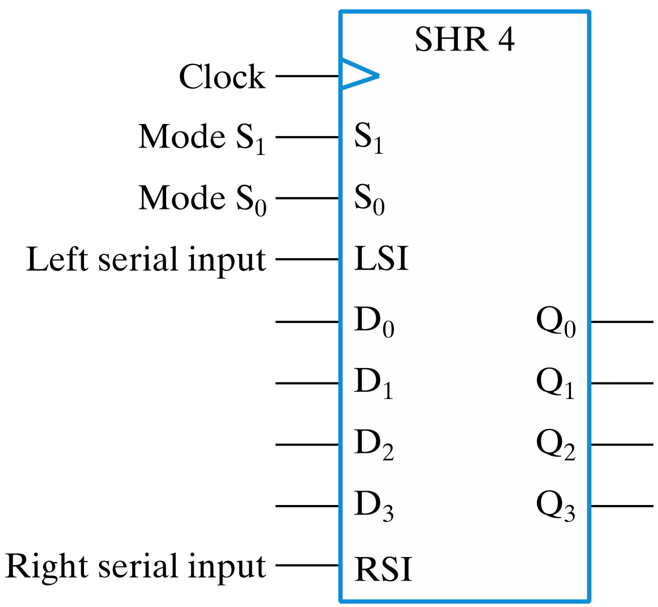
(b) Symbol



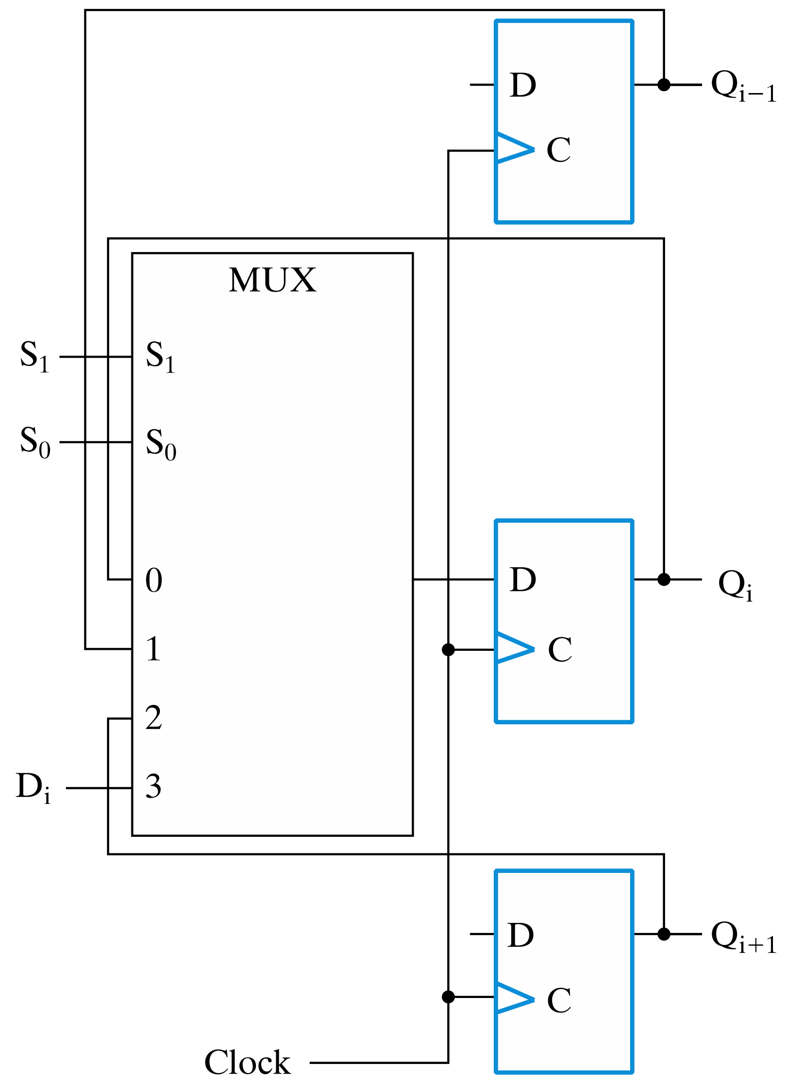
Registers

Function Table

Mode control		Register Operation
S_1	S_0	
0	0	No change
0	1	Shift down
1	0	Shift up
1	1	Parallel load



(b) Symbol



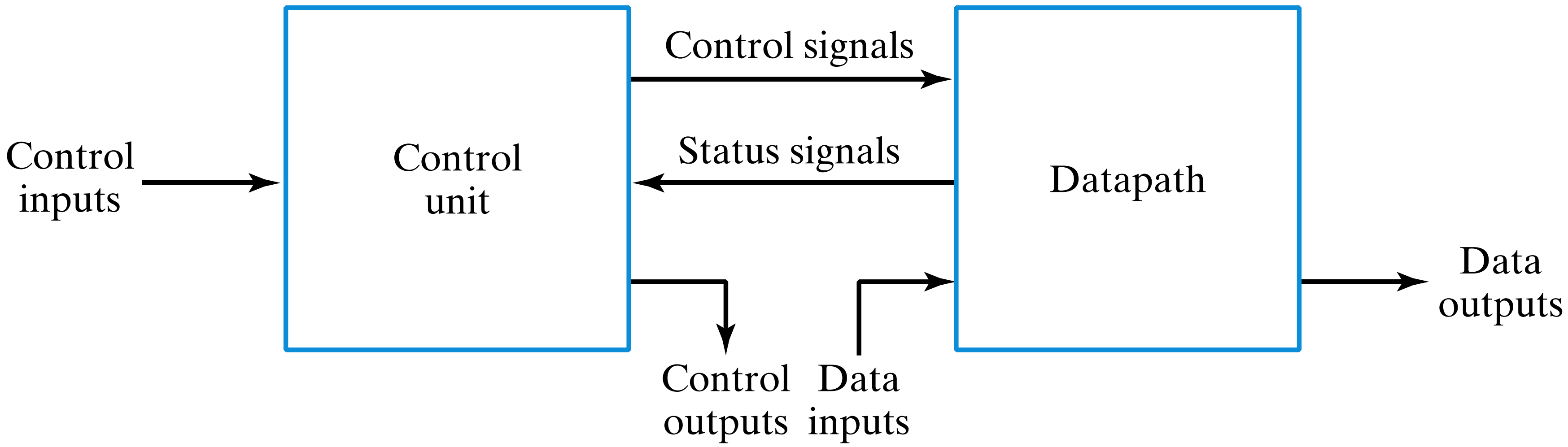
(a) Logic diagram of one typical stage

Register Transfer Operations

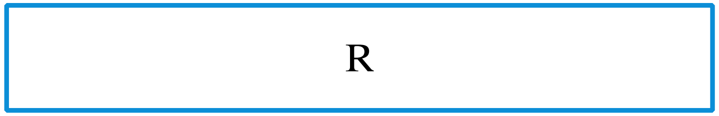
- Register Transfer Operations – The movement and processing of data stored in registers
- Three basic components:
  - set of registers
  - operations
  - control of operations
- Elementary Operations -- load, count, shift, add, bitwise "OR", etc.
  - Elementary operations called *microoperations*

Register Transfer Operations

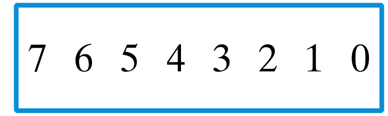
- The system is partitioned into 2 types of modules:
  - Datapath: performs data processing operations.
  - Control unit: determines the sequence of those operations.
- Datapaths are defined by their registers and the operations performed on binary data stored in the registers



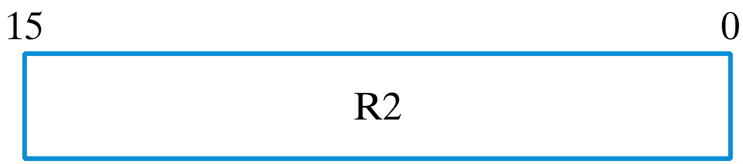
Register Notation



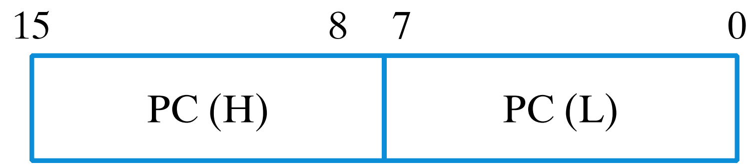
(a) Register R



(b) Individual bits of 8-bit register



(c) Numbering of 16-bit register



(d) Two-part 16-bit register

Basic Symbols for Register Transfers

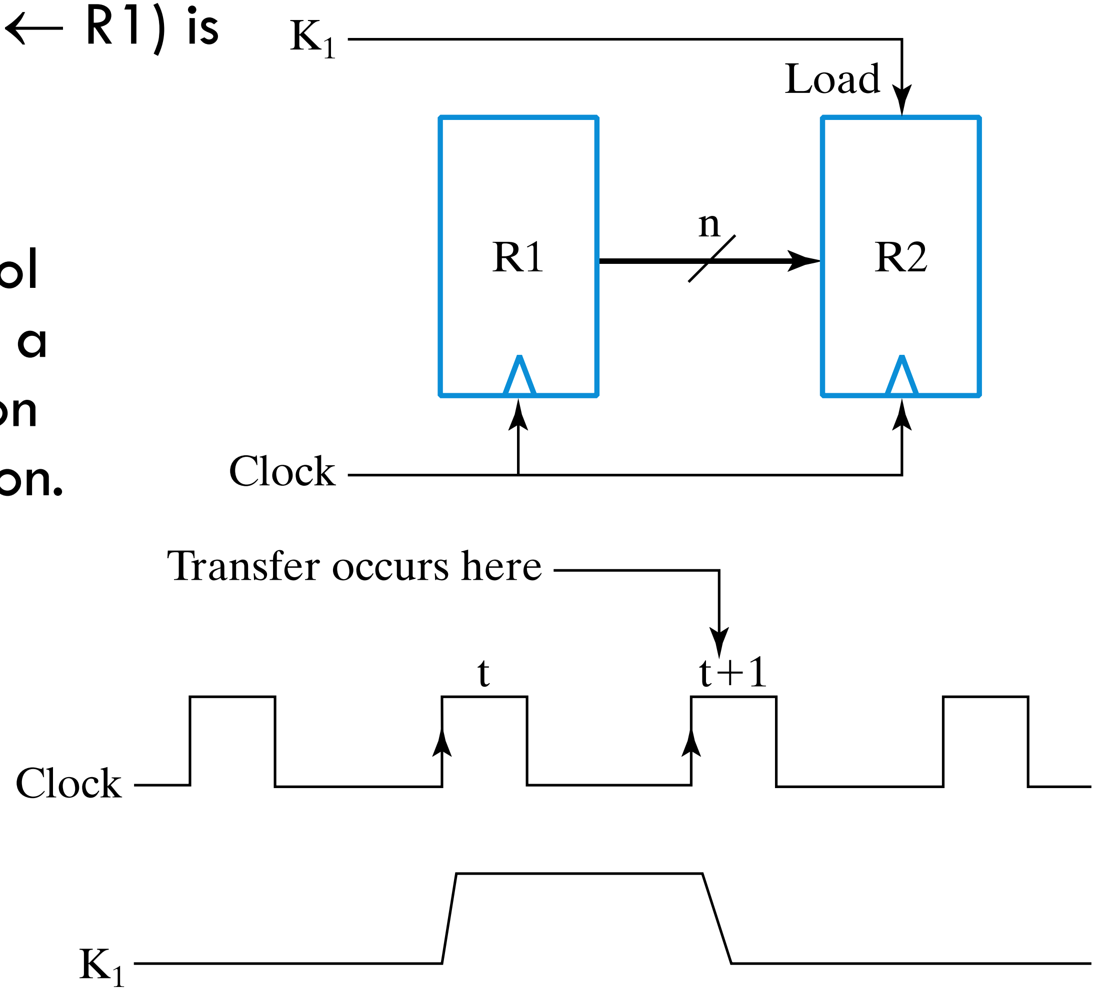
Symbol	Description	Examples
Letters (and numerals)	Denotes a register	AR, $R2$, DR, IR
Parentheses	Denotes a part of a register	$R2(1)$, $R2(7{:}0)$, $AR(L)$
Arrow	Denotes transfer of data	$R1 \leftarrow R2$
Comma	Separates simultaneous transfers	$R1 \leftarrow R2,\ R2 \leftarrow R1$
Square brackets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

Conditional Transfer

- If (K1 =1) then (R2 ← R1) is shortened to

  K1: (R2 ← R1)

  where K1 is a control variable specifying a conditional execution of the microoperation.



Microoperations

- A microoperation is an elementary operation performed on data stored in register or in memory.
- Types of microoprtations:
  - 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

Register Trasfers

Textbook RTL, VHDL, and Verilog Symbols for Register Transfers

Operation	Text RTL	VHDL	Verilog
Combinational assignment	=	<= (concurrent)	assign = (nonblocking)
Register transfer	←	<= (concurrent)	<= (nonblocking)
Addition	+	+	+
Subtraction	–	–	–
Bitwise AND	^	and	&
Bitwise OR	∨	or	|
Bitwise XOR	⊕	xor	^
Bitwise NOT	‾ (overline)	not	~
Shift left (logical)	sl	sll	<<
Shift right (logical)	sr	srl	>>
Vectors/registers	*A*(3:0)	*A*(3 down to 0)	*A*[3:0]
Concatenation	||	&	{ , }

Example Microoperations

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

  $R1 \leftarrow R1 + R2$
- Subtract the content of R6 from R1 and place the result in R1

  $R1 \leftarrow R1 + R6' + 1$
- 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 K1 OR K2, the content of R1 is Logic bitwise Ored with the content of R3 and the result placed in R1.

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

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

  In $R1 \leftarrow R1 + R3$, + means "plus."

Control Expressions

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

- **<u>Example</u>:**

  $\overline{X}K1 : R1 \leftarrow R1 + R2$

  $X K1 : R1 \leftarrow R1 + \overline{R2} + 1$
- **Variable K1 enables the add or subtract operation.**
- **<u>If</u> X =0, then $\overline{X}$ =1 so $\overline{X}$ K1 = 1, activating the addition of R1 and R2.**
- **If X = 1, then X K1 = 1, activating the addition of R1 and the two's complement of R2 (subtract).**

Arithmetic Microoperations

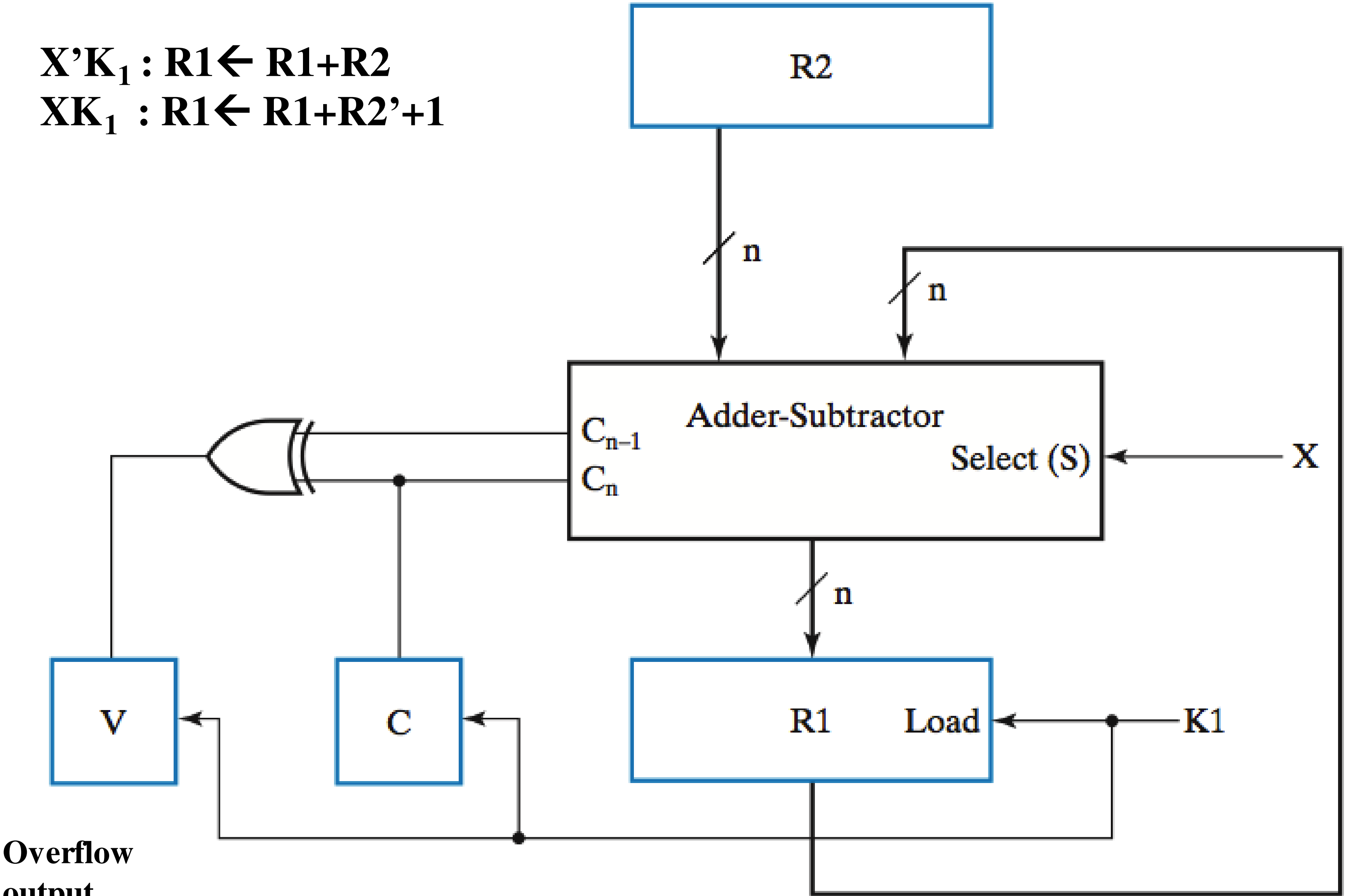
Arithmetic Microoperations

Symbolic designation	Description
$R0 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R0$
$R2 \leftarrow \overline{R2}$	Complement of the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement of the contents of $R2$
$R0 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus 2's 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)

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

Adder/ Subtracter Unit

$X'K_1$: R1← R1+R2
XK_1 : R1← R1+R2'+1



Logical Microoperations

Logic Microoperations

Symbolic designation	Description
$R0 \leftarrow \overline{R1}$	Logical bitwise NOT (1's 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)

- 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

Examples of Shifts

		Eight-bit examples	
Type	Symbolic designation	Source *R*2	After shift: Destination *R*1
shift left	$R1 \leftarrow sl\ R2$	10011110	00111100
shift right	$R1 \leftarrow sr\ R2$	11100101	01110010

- 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).

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
- <u>Register cell design</u> 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: $A \leftarrow B$

    Shift: $A \leftarrow \text{sr } B$

    Add: $A \leftarrow A + B$
- A hold state specification
  - Example:
    - Control inputs: Load, Shift, Add
    - If all control inputs are 0, hold the current register state

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:
  - $CX : A \leftarrow B \vee A$
  - $CY : A \leftarrow B \oplus A$
  - 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:

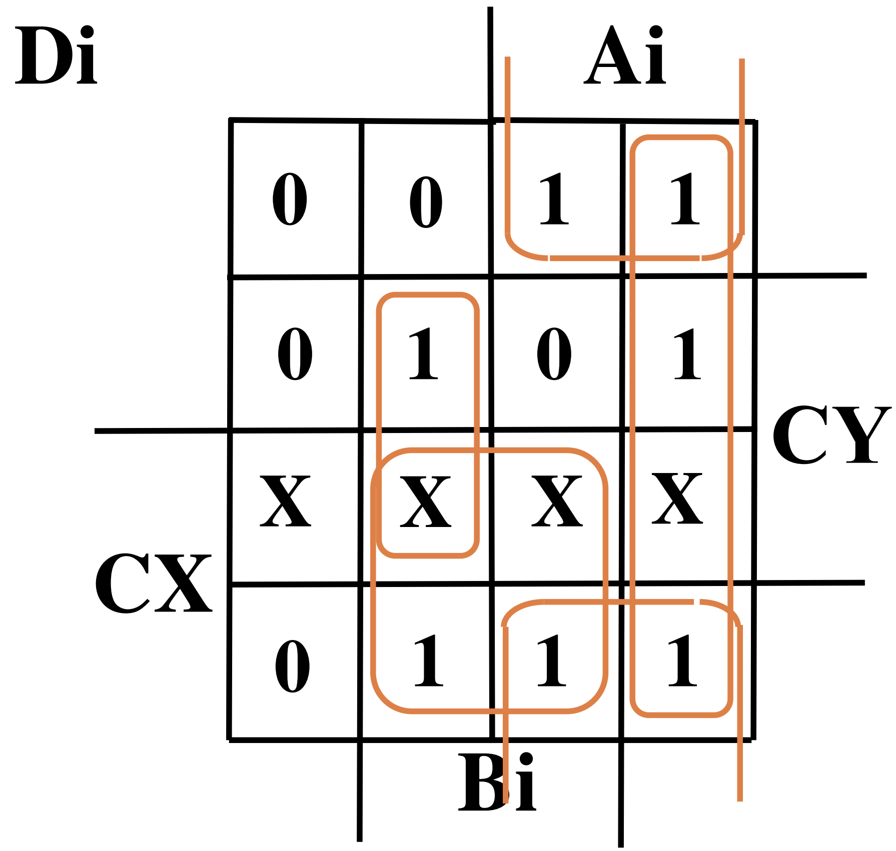
	Hold	$Ai \vee Bi$		$Ai + Bi$	
A_i	$CX = 0$ $CY = 0$	$CX = 1$ $B_i = 0$	$CX = 1$ $B_i = 1$	$CY = 1$ $B_i = 0$	$CY = 1$ $B_i = 1$
0	0	0	1	0	1
1	1	1	1	1	0

- Four variables 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



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

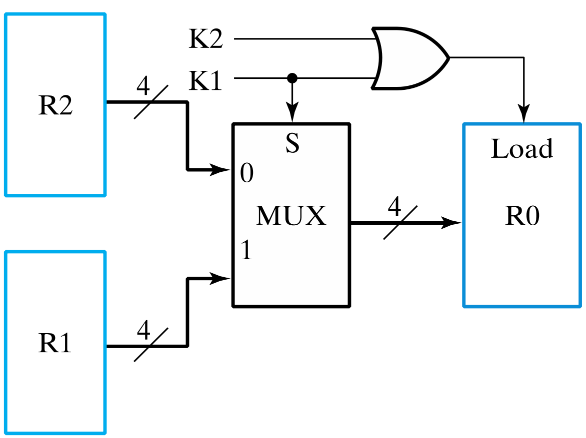
  Shared decoder logic: 8
- Cost gain by sequential design > 5 per cell
- Also, no Enable on the flip-flop makes it cost less

Register Transfer Structures

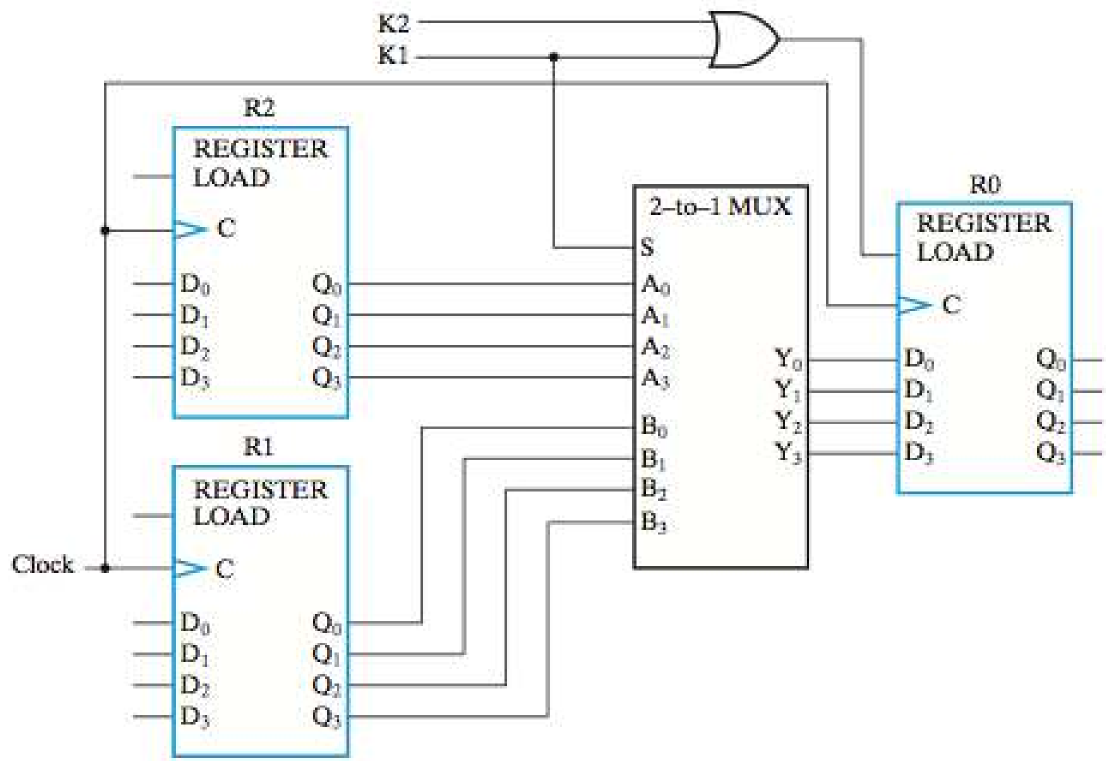
- Multiplexer-Based Transfers - Multiple inputs are selected by a multiplexer dedicated to the register
- Bus-Based Transfers - Multiple inputs are selected by a shared multiplexer driving a bus that feeds inputs to multiple registers
- Three-State Bus - Multiple inputs are selected by 3-state drivers 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 × K1: R0 ← R2



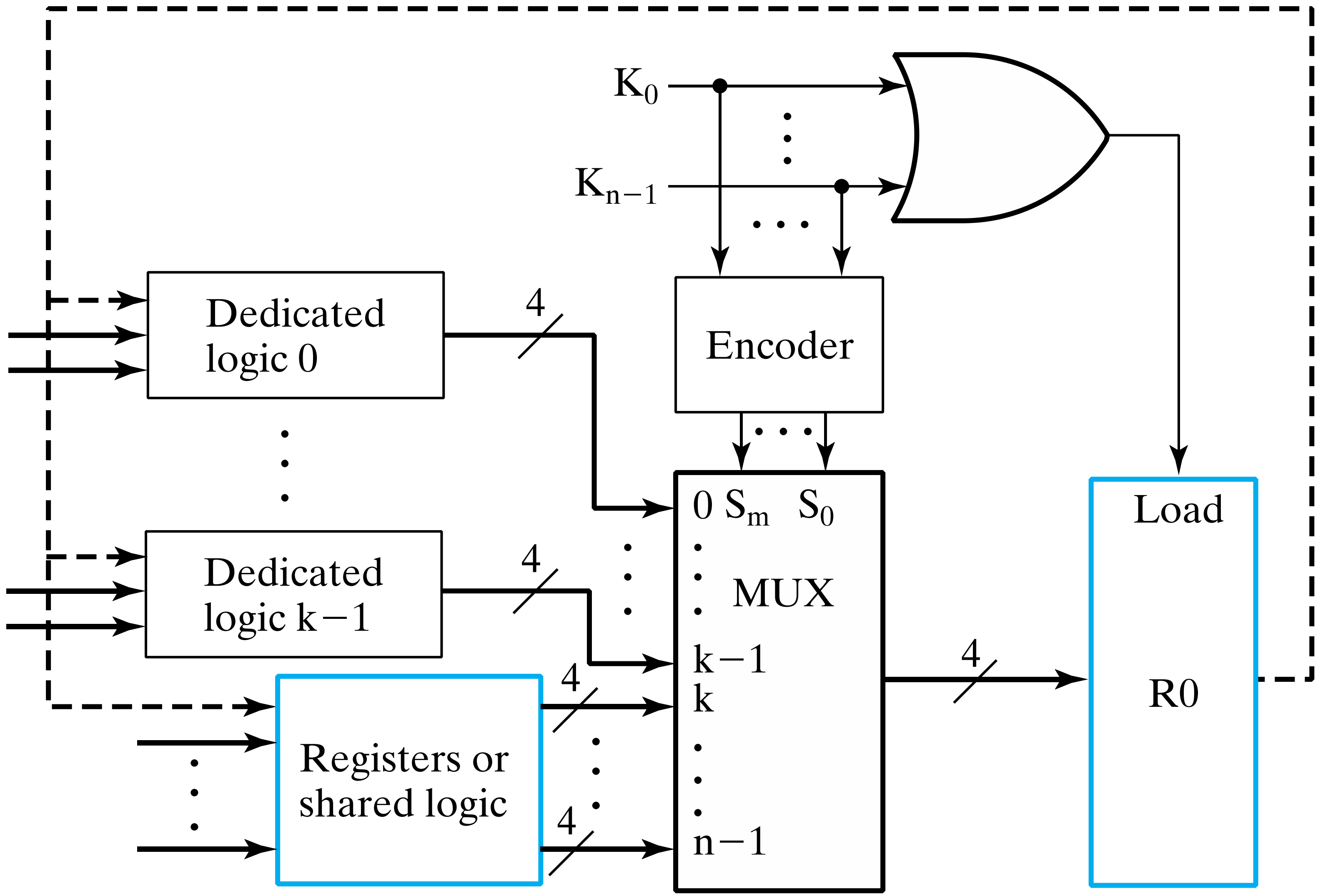
(a) Block diagram



(b) Detailed logic

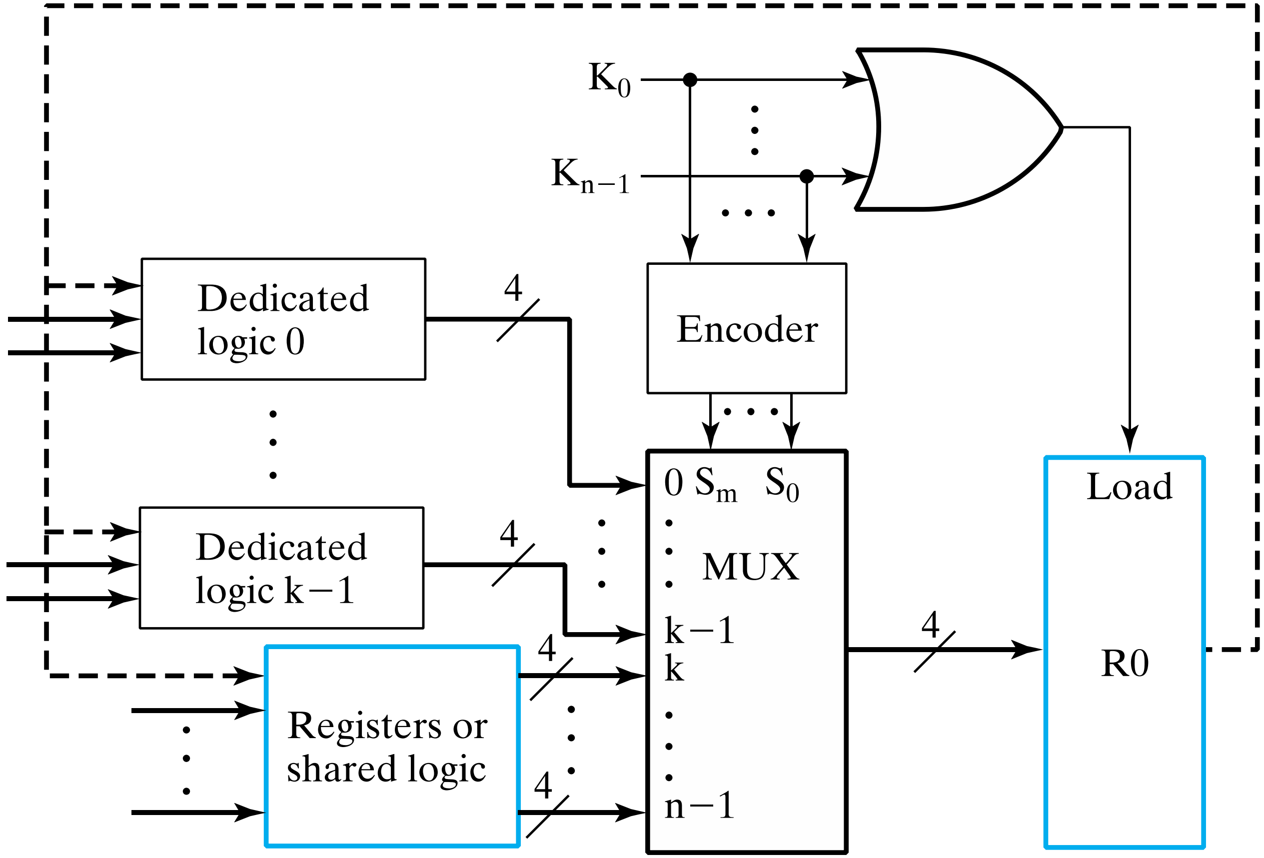
Multiplexer Approach

- 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

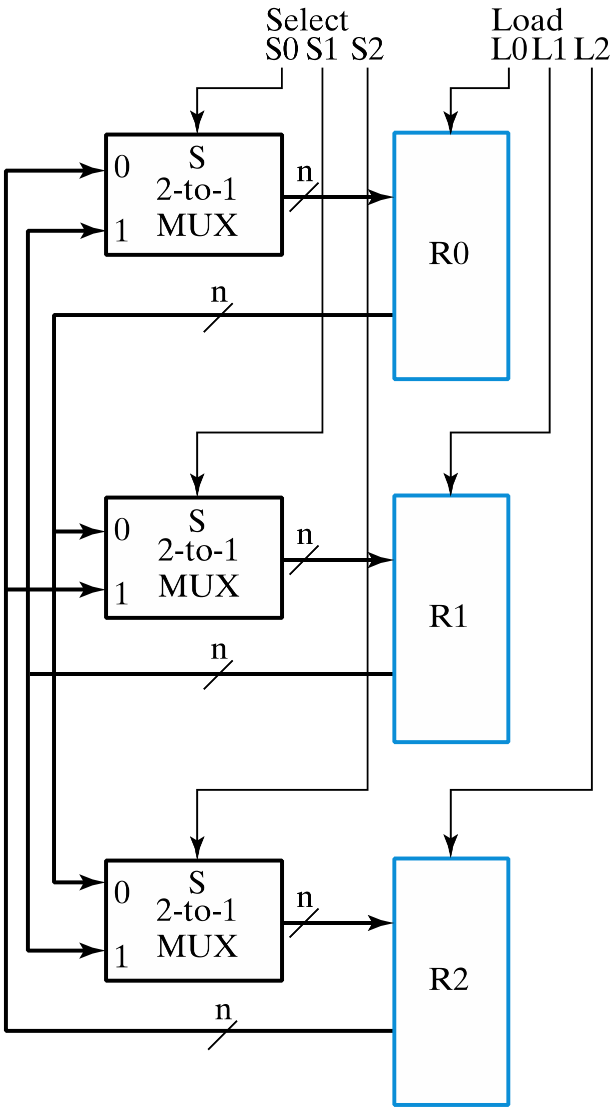


Multiplexer and Bus-Based Transfers for Multiple Registers

- 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.

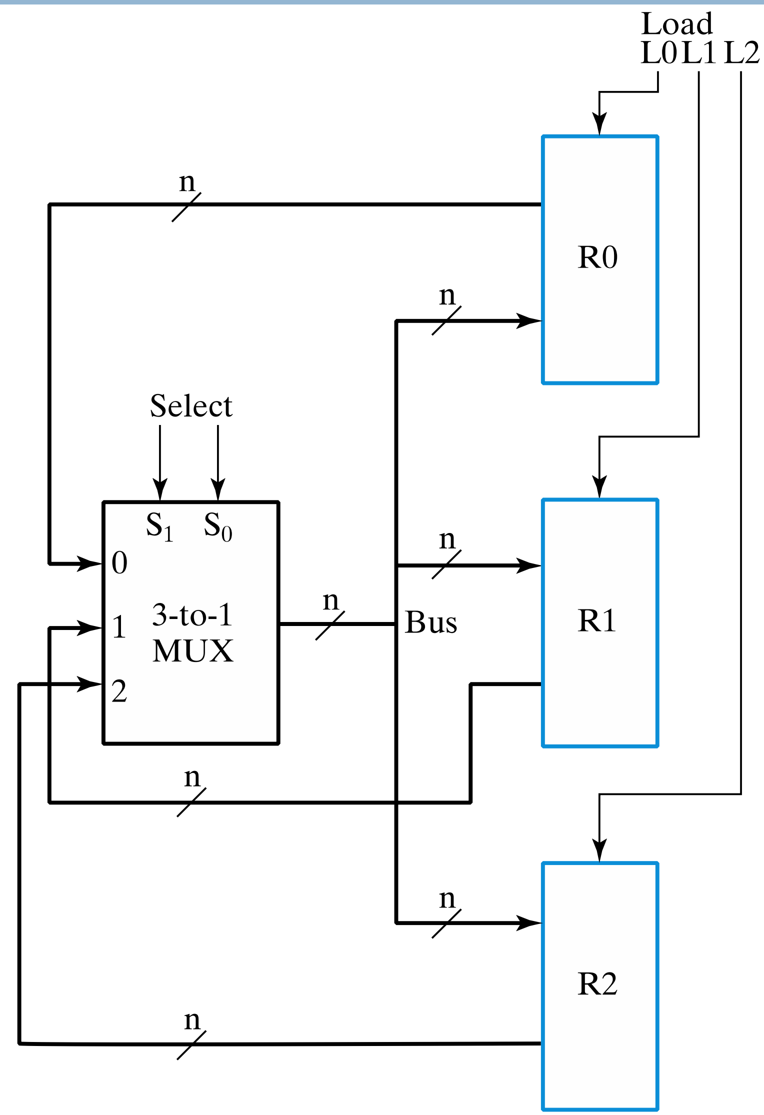


(a) Dedicated multiplexers

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

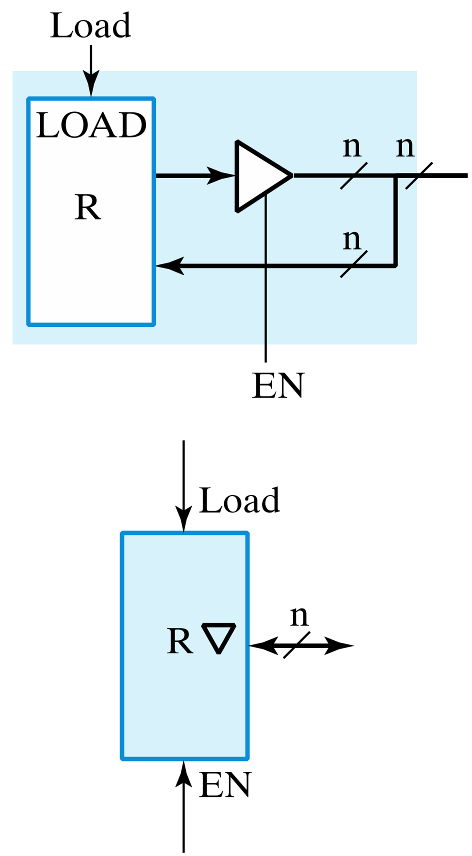
	Select		Load		
Register Transfer	**S1**	**S0**	**L2**	**L1**	**L0**
$R0 \leftarrow R2$	1	0	0	0	1
$R0 \leftarrow R1,\ R2 \leftarrow R1$	0	1	1	0	1
$R0 \leftarrow R1,\ R1 \leftarrow R0$			Impossible		



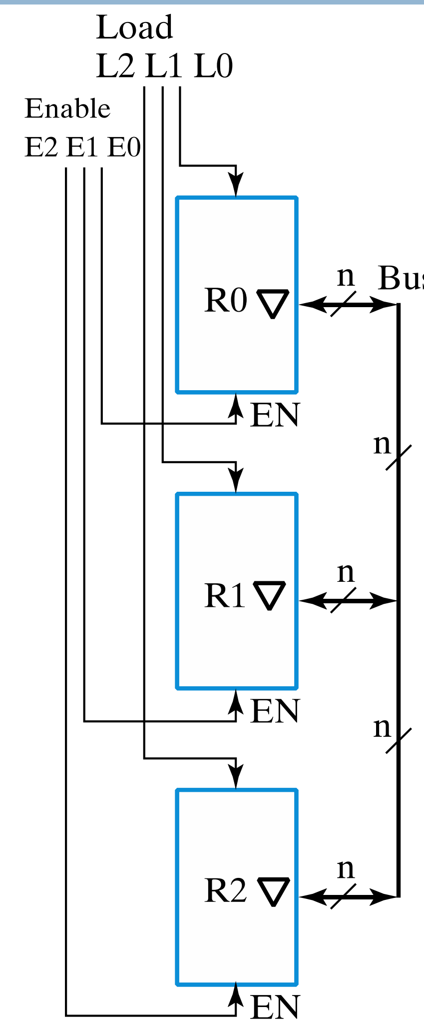
(b) Single bus

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



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

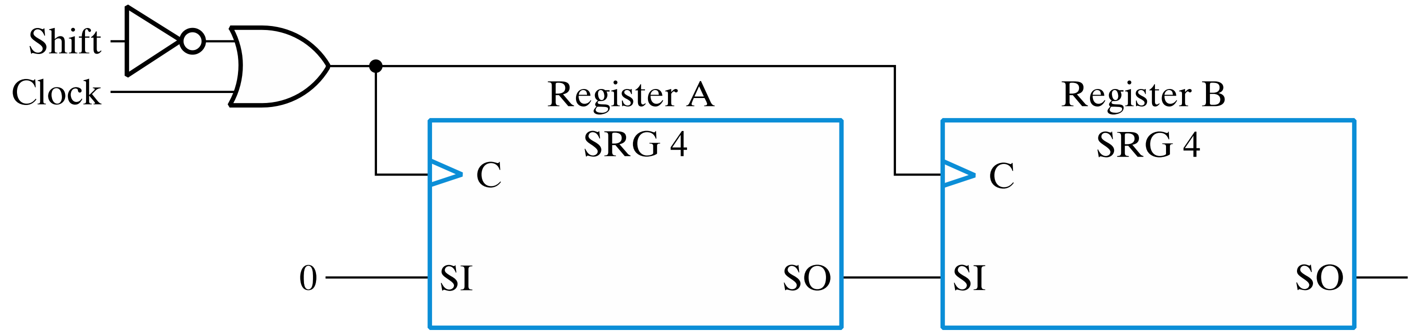


(c) Three-state bus using registers with bidirectional lines

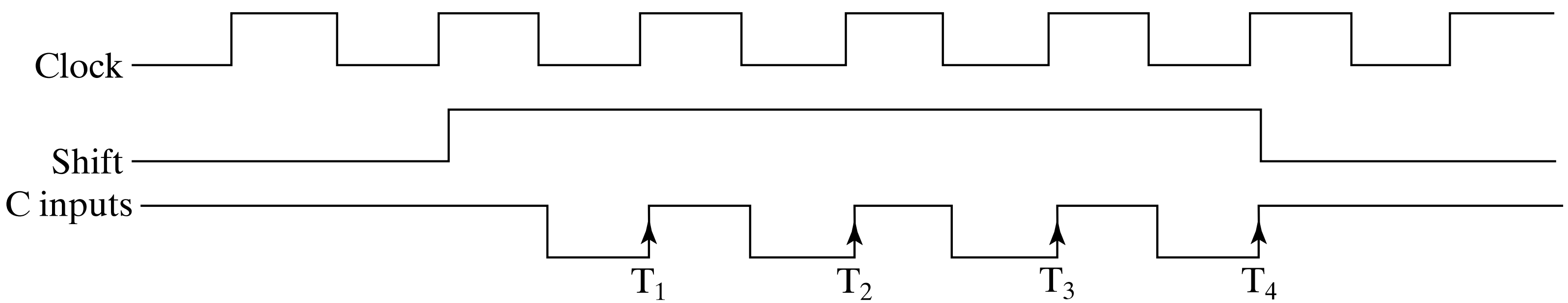
Serial Transfers and Microoperations

- Serial Transfers
  - Used for "narrow" transfer paths
  - Example 1: Telephone or cable line
    - <u>Parallel-to-Serial</u> conversion at source
    - <u>Serial-to-Parallel</u> 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 Transfer



(a) Block diagram



(b) Timing diagram

Example of Serial Transfer

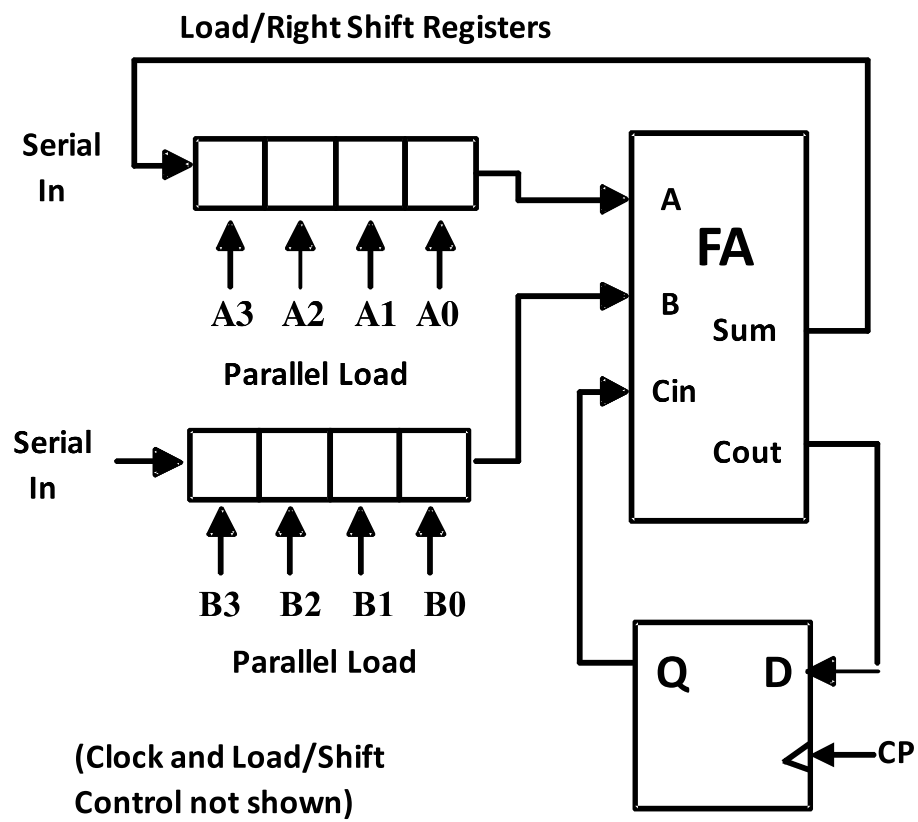
Timing pulse	Shift Register A				Shift Register B			
Initial value	1	0	1	1	0	0	1	0
After T_1	0	1	0	1	1	0	0	1
After T_2	0	0	1	0	1	1	0	0
After T_3	0	0	0	1	0	1	1	0
After T_4	0	0	0	0	1	0	1	1

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 <u>left</u> is equivalent to <u>multiplying by 2</u>.
- Shifting a binary number <u>right</u> is equivalent to <u>dividing by 2</u>.

Serial Adder

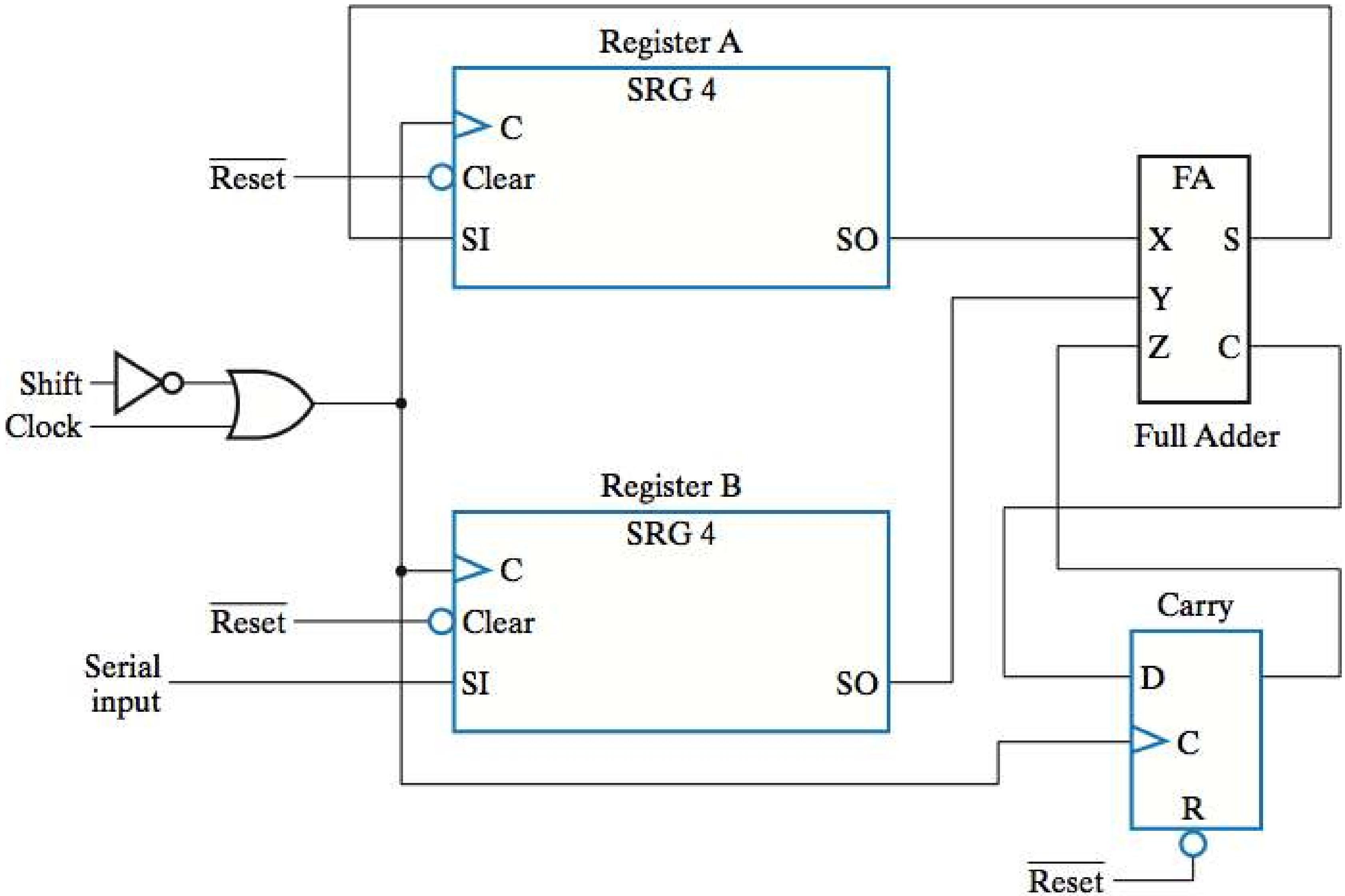
- The circuit shown uses two shift registers for operands A(3:0) and B(3:0).
- A full adder, and one more flip flop (for the carry) is used to compute the sum.
- The result is stored in the A register and the final carry in the flip-flop



(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.

Serial Adder

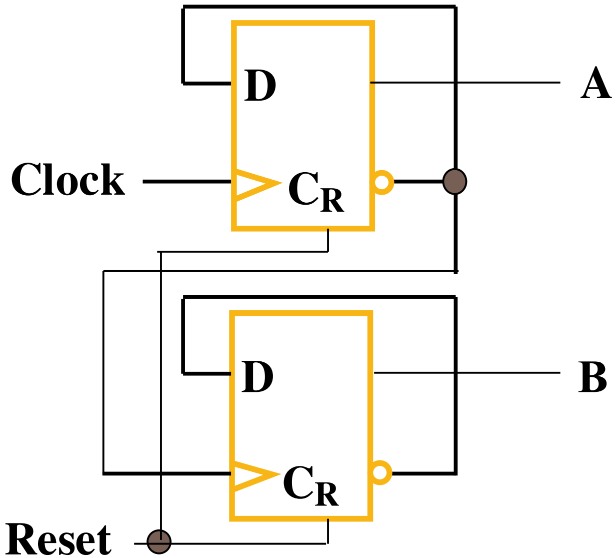


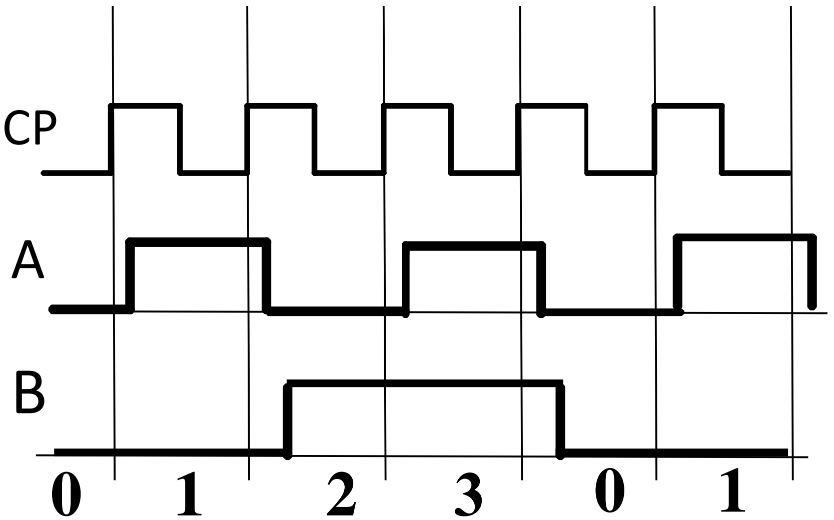
Counters

- 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

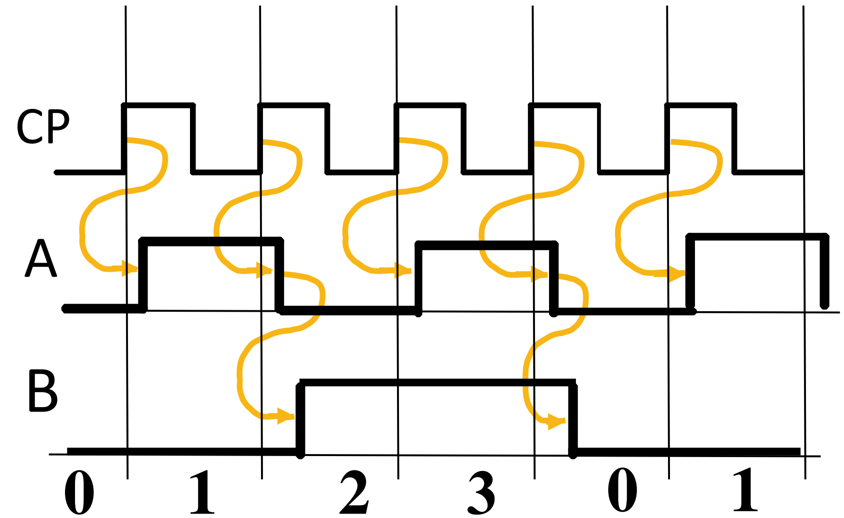
- How does it work?
  - When there is a positive edge on the clock input of A, A complements
  - The clock input for flip-flop B is the complemented output of flip-flop A
  - When flip A changes from 1 to 0, there is a positive edge on the clock input of B causing B to complement





Ripple Counter (continued)

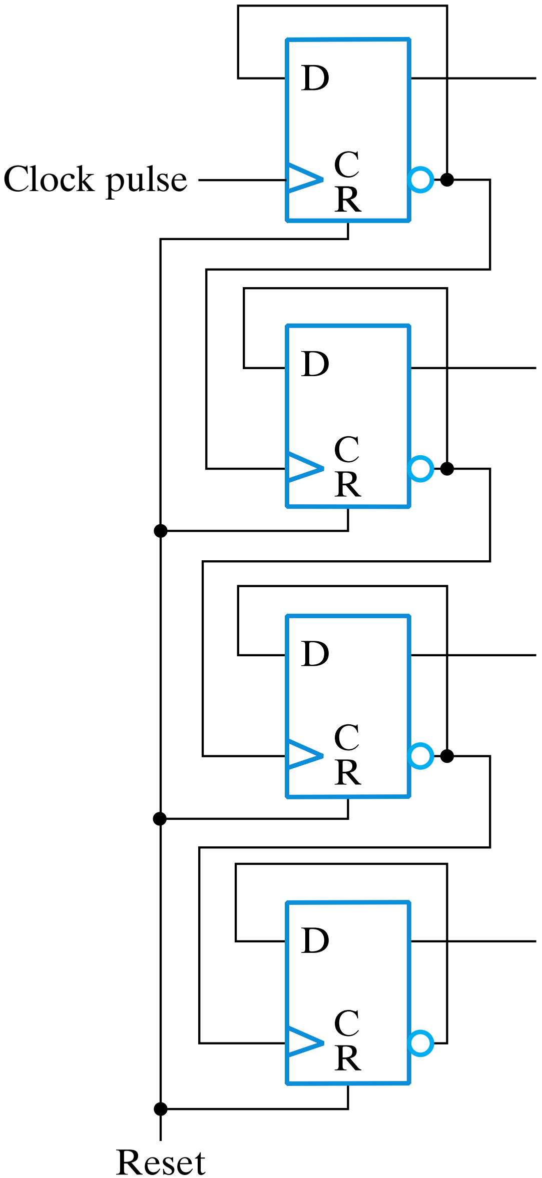
- The arrows show the cause-effect relationship from the prior slide =>



- 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), …

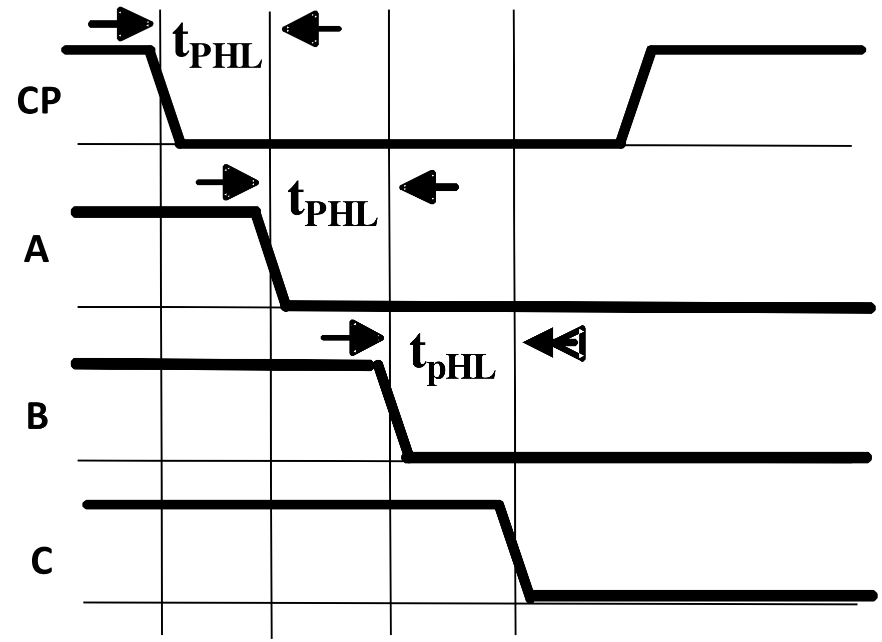
Ripple Counter (continued)

- 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 $n\ t_{PHL}$.

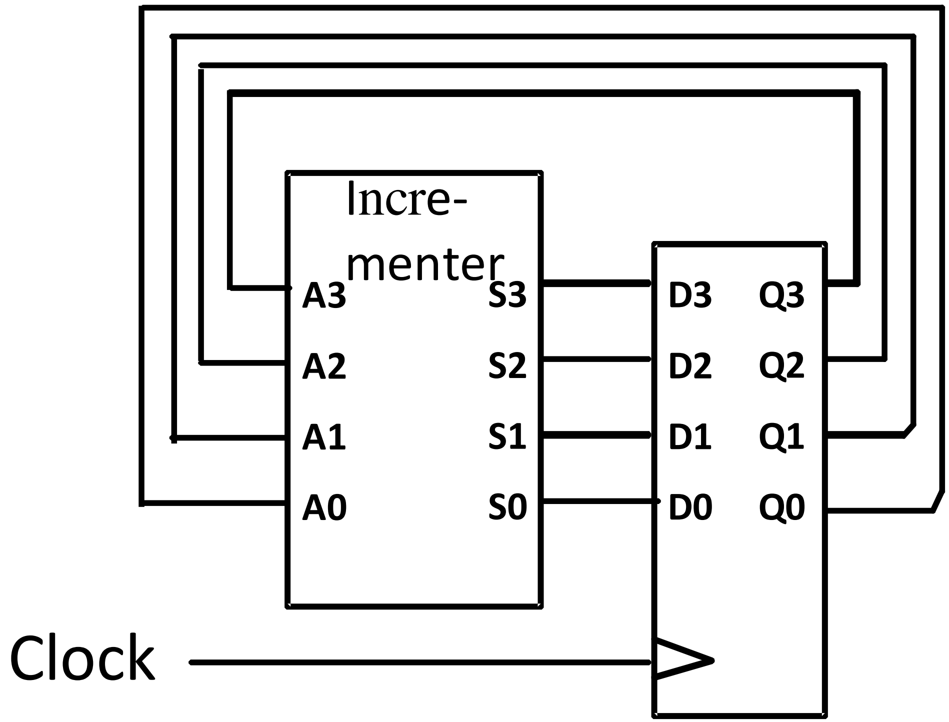


Synchronous Counters

- 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 =>

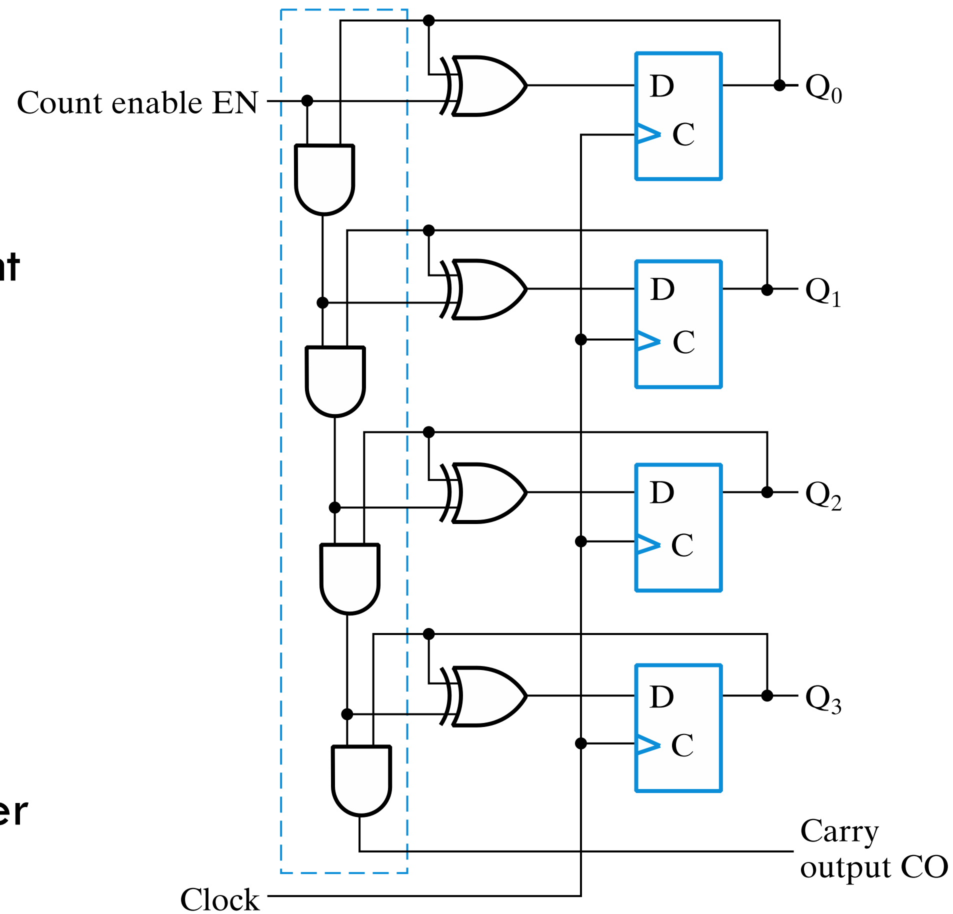
Counting Sequence of Binary Counter

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



Synchronous Counters (continued)

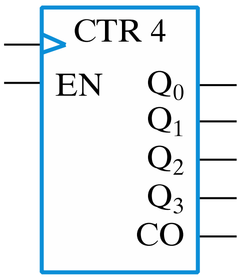
- Internal details =>
- Internal Logic
  - XOR complements each bit
  - AND chain causes complement of a bit if all bits toward LSB from it equal 1
- Count Enable
  - Forces all outputs of AND chain to 0 to "hold" the state
- Carry Out
  - Added as part of incrementer
  - Connect to Count Enable of additional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



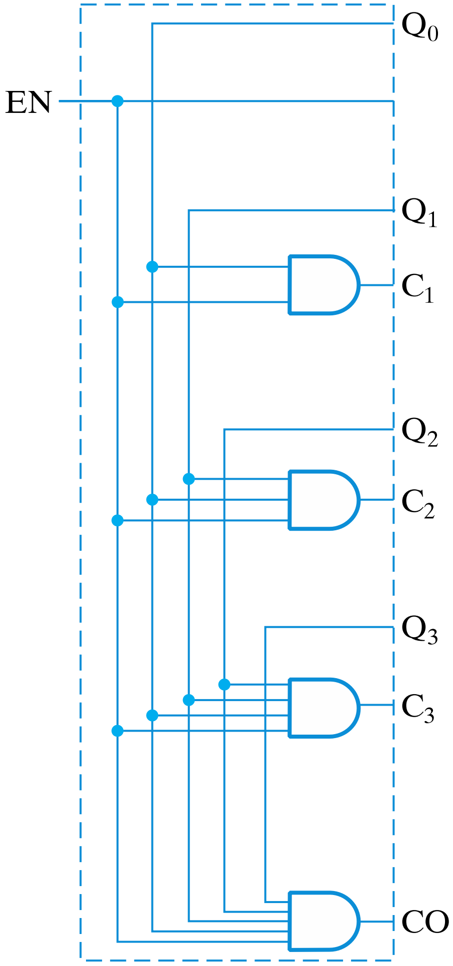
(a) Logic diagram-serial gating

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



(c) Symbol



(b) Logic diagram-parallel gating

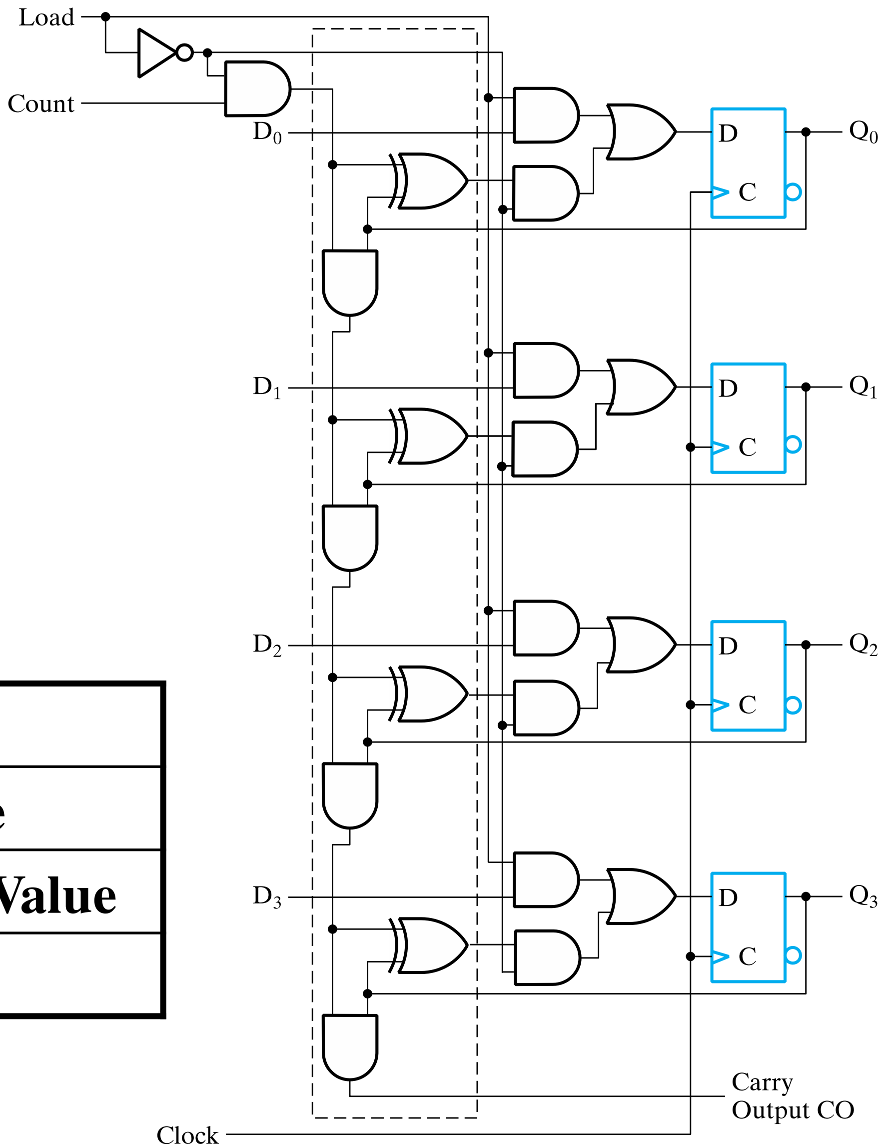
Other Counters

- 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



Counter w/ Unused States

- *n* flip-flops ⇒ 2*n* binary states
- Unused states: — states that are not used in specifying the sequential ckt — maybetreatedasdon't-careconditionsor
- may be assigned specific next states ␣ Self-correcting counter:
  - Ensure that when a ckt enter one of its unused states, it eventually goes into one of the valid states after one or more clock pulses so it can resume normal operation.
    - Analyze the ckt to determine the next state from an unused state after it is designed.

Counter w/ Unused States

- Example:

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

The simplified f-f input eqs

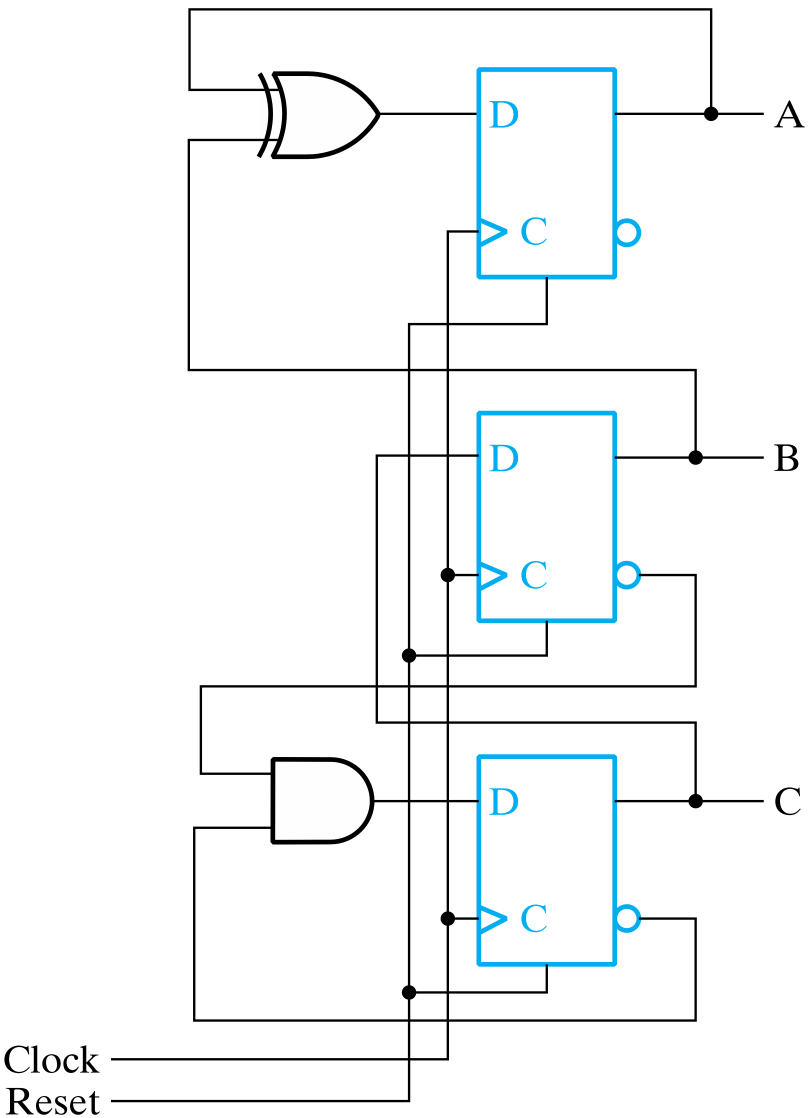
$$D_A = A \oplus B$$

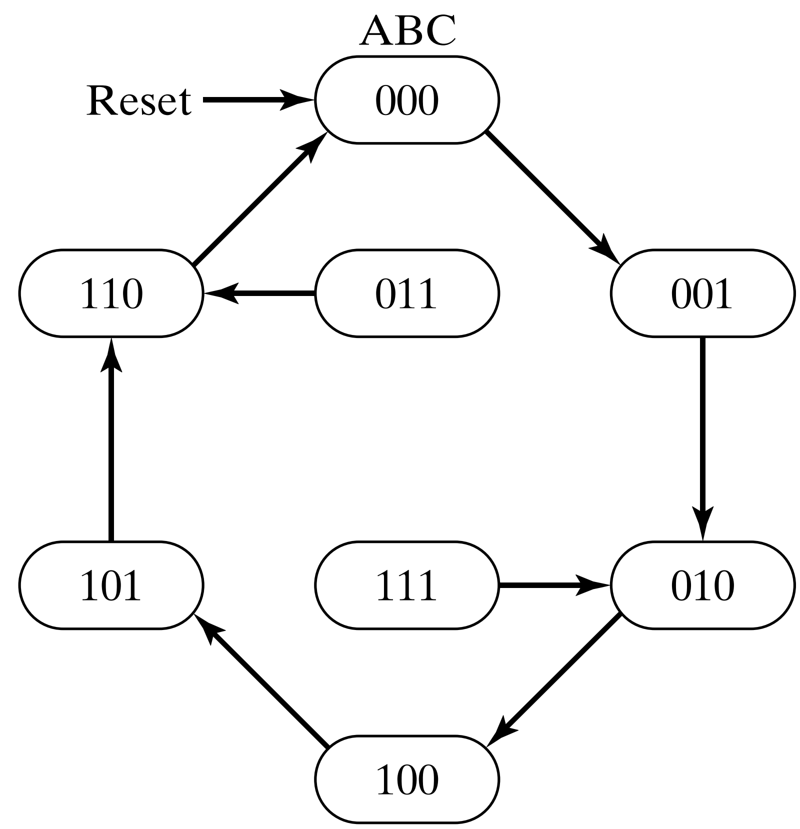
$$D_B = C$$

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

$\boxed{\text{Two unused states: } 011 \ \& \ 111}$

Counter w/ Unused States





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 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

Synchronous BCD (continued)

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

$$D1 = \overline{Q1}$$

$$D2 = Q2 \oplus Q1\overline{Q8}$$

$$D4 = Q4 \oplus Q1Q2$$

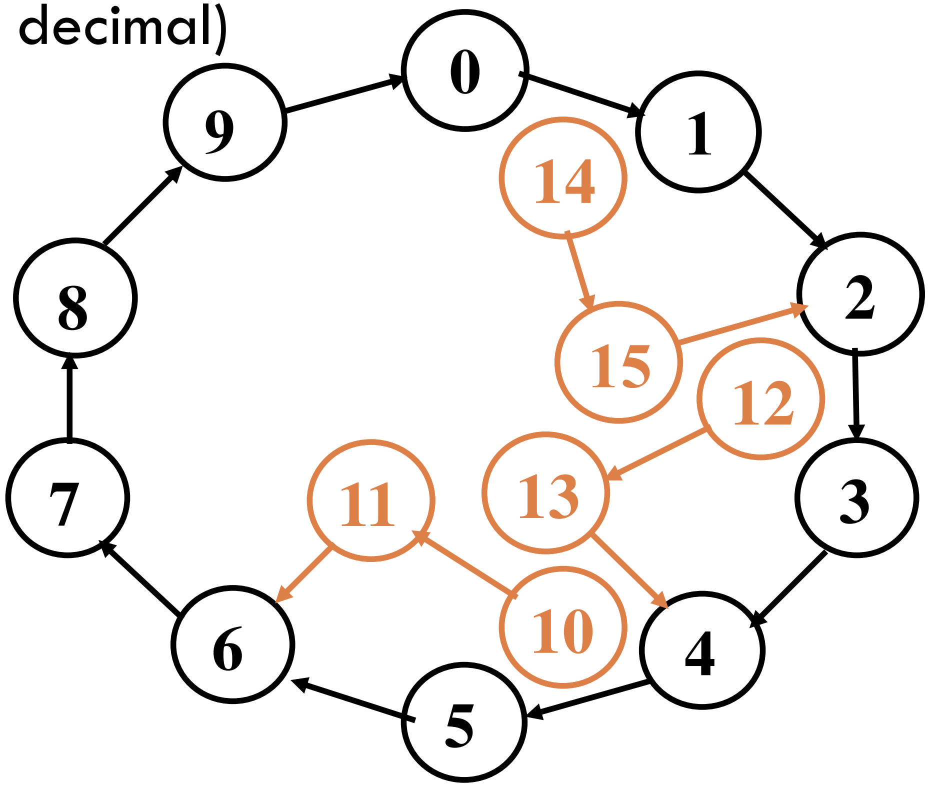
$$D8 = Q8 \oplus (Q1Q8 + Q1Q2Q4)$$

- 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)

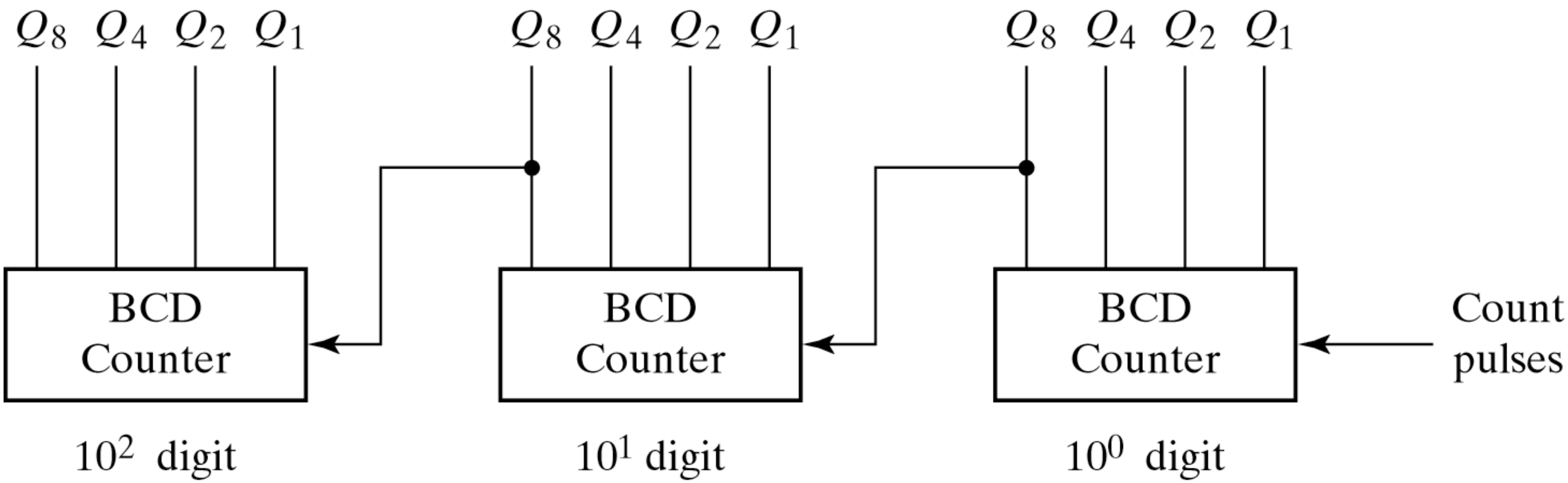
Present State				Next State			
Q8	Q4	Q2	Q1	Q8	Q4	Q2	Q1
1	0	1	0	1	0	1	1
1	0	1	1	0	1	1	0
1	1	0	0	1	1	0	1
1	1	0	1	0	1	0	0
1	1	1	0	1	1	1	1
1	1	1	1	0	0	1	0



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.

Three Decade Decimal Counter



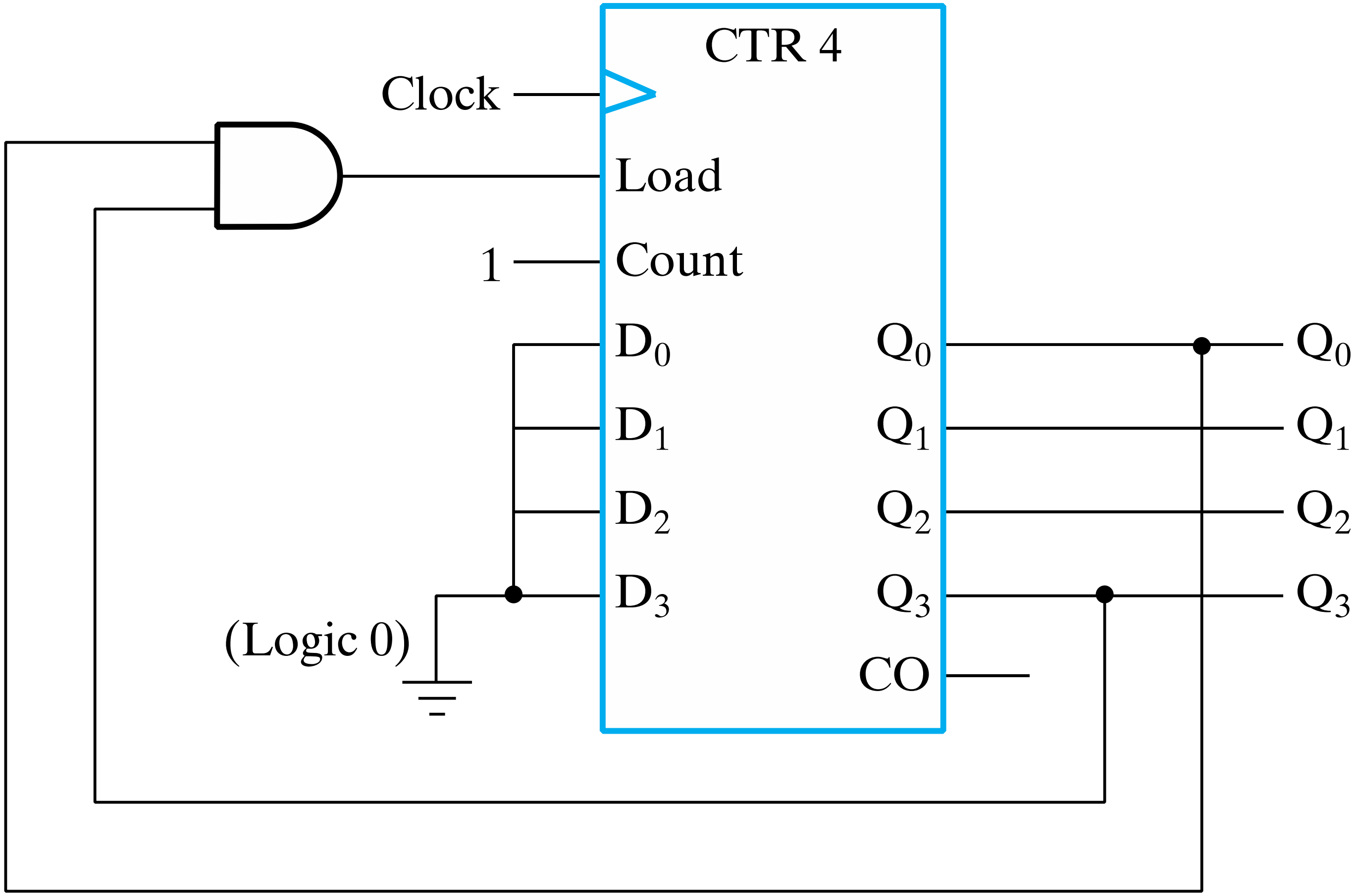
Block Diagram of a Three-Decade Decimal BCD Counter

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

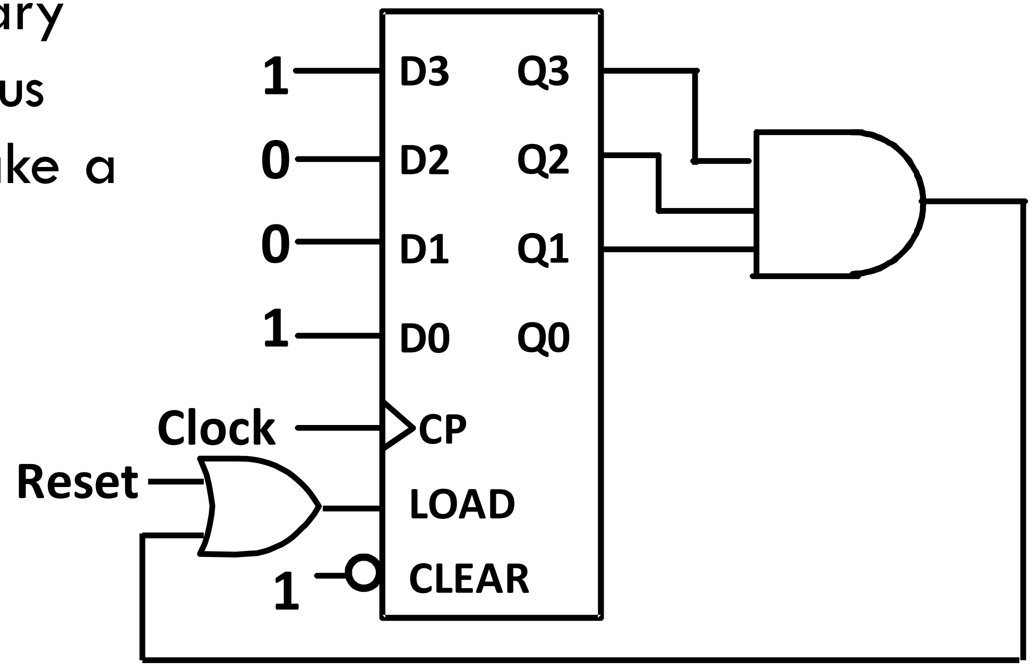
A BCD Counter

- Generate any count sequence:
  - E.g.: design a BCD counter by using a counter w/ parallel load & async clear



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

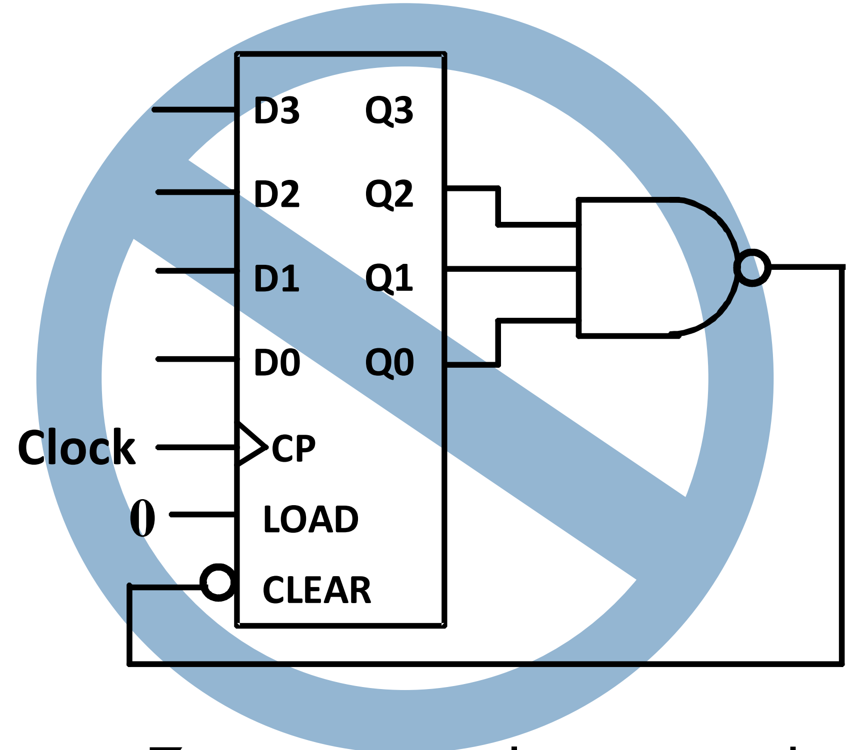
- A synchronous, 4-bit binary counter with a synchronous Load is to be used to make a Modulo 6 counter.
- Use the Load feature to preset the count to 9 on Reset and detection of count 14.



- 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)

Counting Modulo 7: Detect 7 and Asynchronously Clear

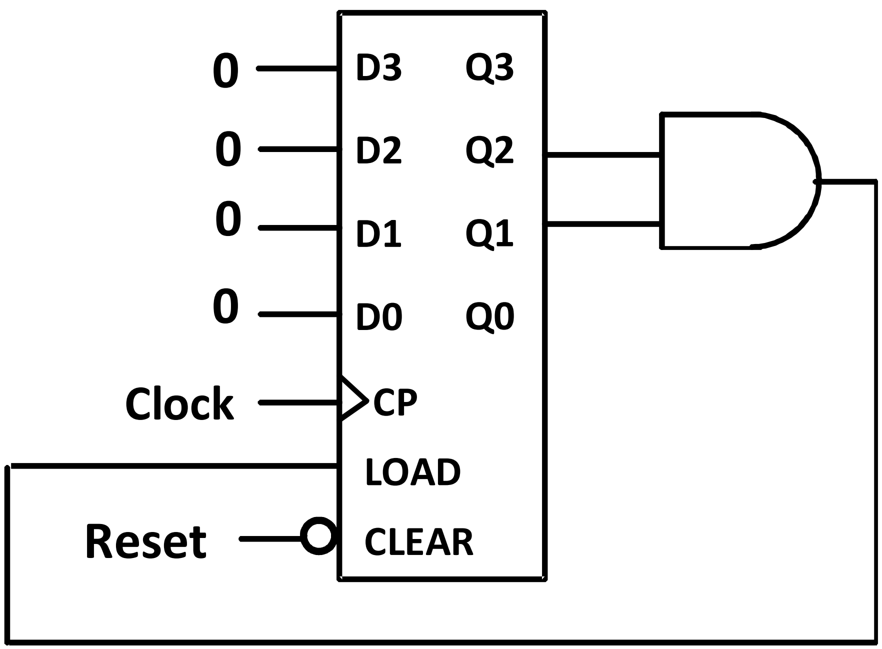
- A synchronous 4-bit binary counter with an asynchronous Clear is used to make a Modulo 7 counter.
- Use the Clear feature to detect the count 7 and clear the count to 0. This gives a count of 0, 1, 2, 3, 4, 5, 6, 7(short)0, 1, 2, 3, 4, 5, 6, 7(short)0, etc.



- 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

- 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 = Q1 Q2



4-bit Shift Register with Reset

```
library ieee;
use ieee.std_logic_1164.all;

entity srg_4_r is
   port(CLK, RESET, SI : in std_logic;
           Q : out std_logic_vector(3  downto  0);
         SO : out std_logic);
end srg_4_r;
```

4-bit Shift Register with Reset

```
architecture behavioral of srg_4_r is
signal shift : std_logic_vector (3 downto 0);
begin
process (RESET, CLK)
begin
  if (RESET = '1') then
        shift <= "0000”;
  elsif (CLK'event and (CLK = '1')) then
        shift <= shift(2 downto 0) & SI;
  end if;
end process;
Q <= shift;
SO <= shift(3);
end behavioral;
```

4-bit Binary Counter with Reset

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity count_4_r is
   port(CLK, RESET, EN : in std_logic;
         Q              : out std_logic_vector(3  downto  0);
         CO             : out std_logic);

end count_4_r;
```

4-bit Binary Counter with Reset

```
architecture behavioral of count_4_r is
signal count : std_logic_vector(3 downto 0);
begin
process (RESET, CLK)
begin
  if (RESET = '1') then
        count <= "0000”;
  elsif (CLK'event and (CLK = '1') and (EN = '1')) then
        count <= count + "0001”;
  end if;
end process;
CO <= '1' when count = "1111" and EN = '1' else '0’;
Q <= count;
end behavioral;
```