
Logic and Computer Design Fundamentals

Chapter 6 – Registers and Register Transfers

Part 1 – Registers, Microoperations and Implementations

Ming Cai

cm@zju.edu.cn

College of Computer Science and Technology,
Zhejiang University

Overview

- **Part 1 - 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**
- **Part 2 - Counters, Register Cells, Buses, & Serial Operations**
- **Part 3 – Control of Register Transfers**

Methods for Sequential Circuit Design

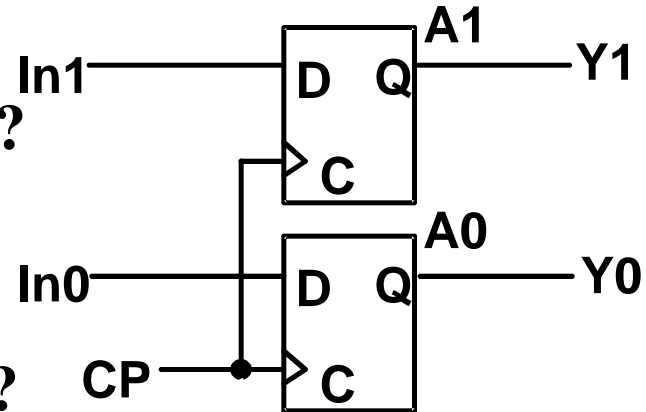
- **Two methodologies for sequential logic design:**
 - **Basic design approach:** uses **flip-flops** and **logic gates** as the basic building blocks.
 - **Register transfer level (RTL) design:** different types of **registers** (e.g., counter, shift register) and **functional blocks** (e.g., multiplexer, adder) are used as the basic building blocks.
- **RTL design is a method for dataflow in which we can transfer data from one register to another.**

Registers

- **Register – a collection of binary storage elements**
- **In theory, a register is a 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	Next State A1(t+1) A0(t+1) For In1 In0 =	Output (=A1 A0)
A1 A0	00 01 10 11	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?

Register Design Models

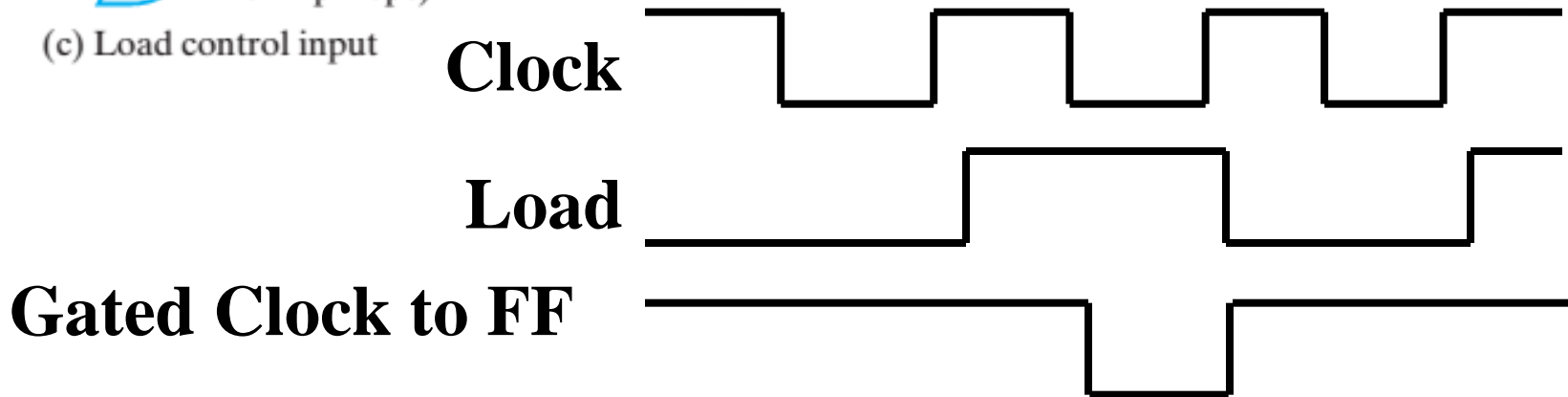
- Due to the large numbers of states and input combinations as n becomes large, the state diagram/state table model is not feasible!
- What are methods we can use to design registers?
 - Add predefined combinational circuits to registers
 - Example: To count up, connect the register flip-flops to an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

- **Reality:**
 - A D flip-flop register loads information on every clock cycle
- **Expectations:**
 - A register can store information for multiple clock cycles
 - To “store” or “load” information should be controlled by a signal
- **Realizing methods:**
 - 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
- **Load** 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 *Load* signal enables the clock signal to pass through if 0 and prevents the clock signal from passing through if 1.
- Example: a Positive Edge-Triggered Master-Slave Flip-flop:

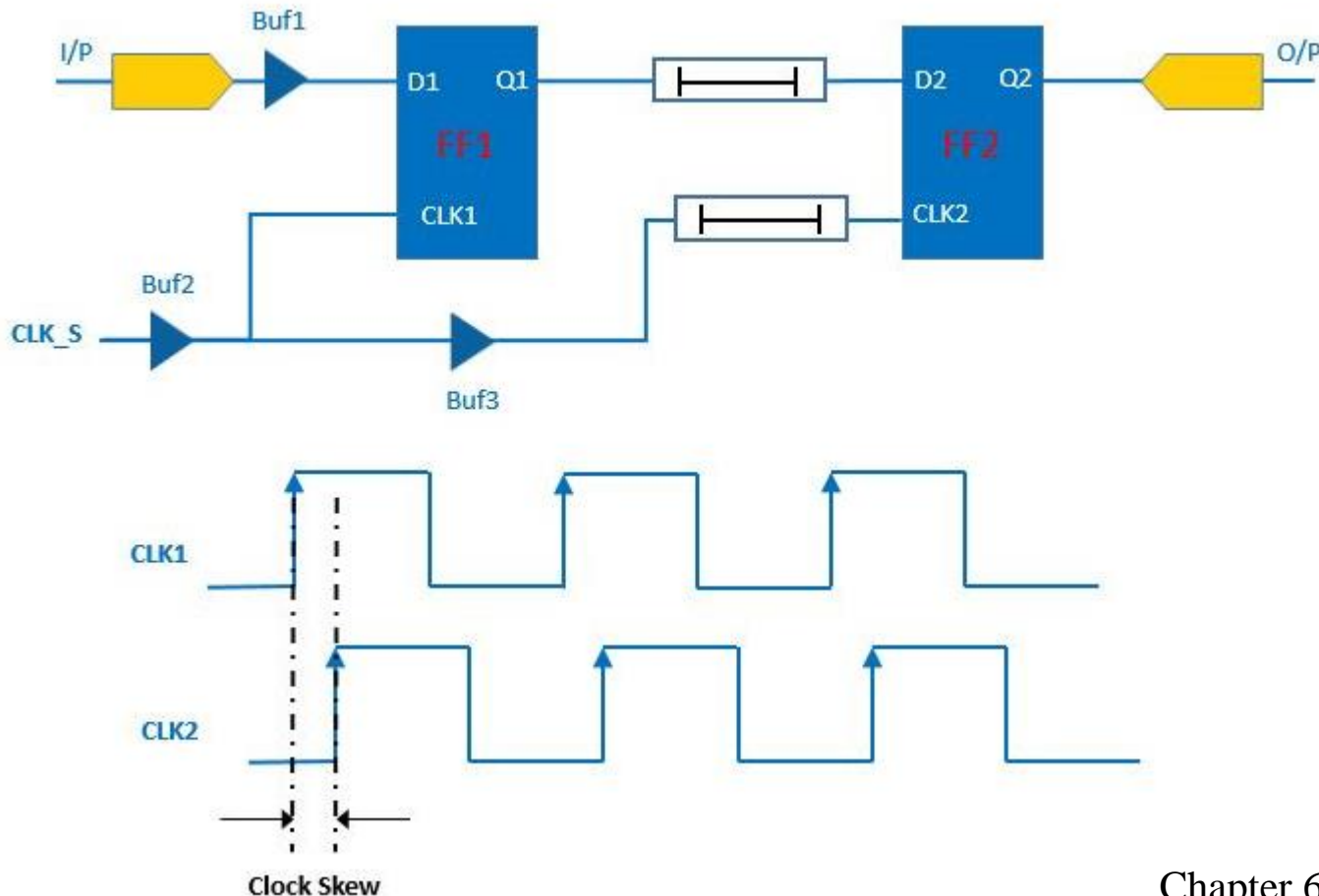


- What logic is needed for gating? **Gated Clock = Clock + $\overline{\text{Load}}$**
- What is the problem?

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

What is Clock Skew

- Clock skew** is a phenomenon in synchronous digital circuit in which the same sourced clock signal arrives at different components at different times.



Registers with Load-Controlled Feedback

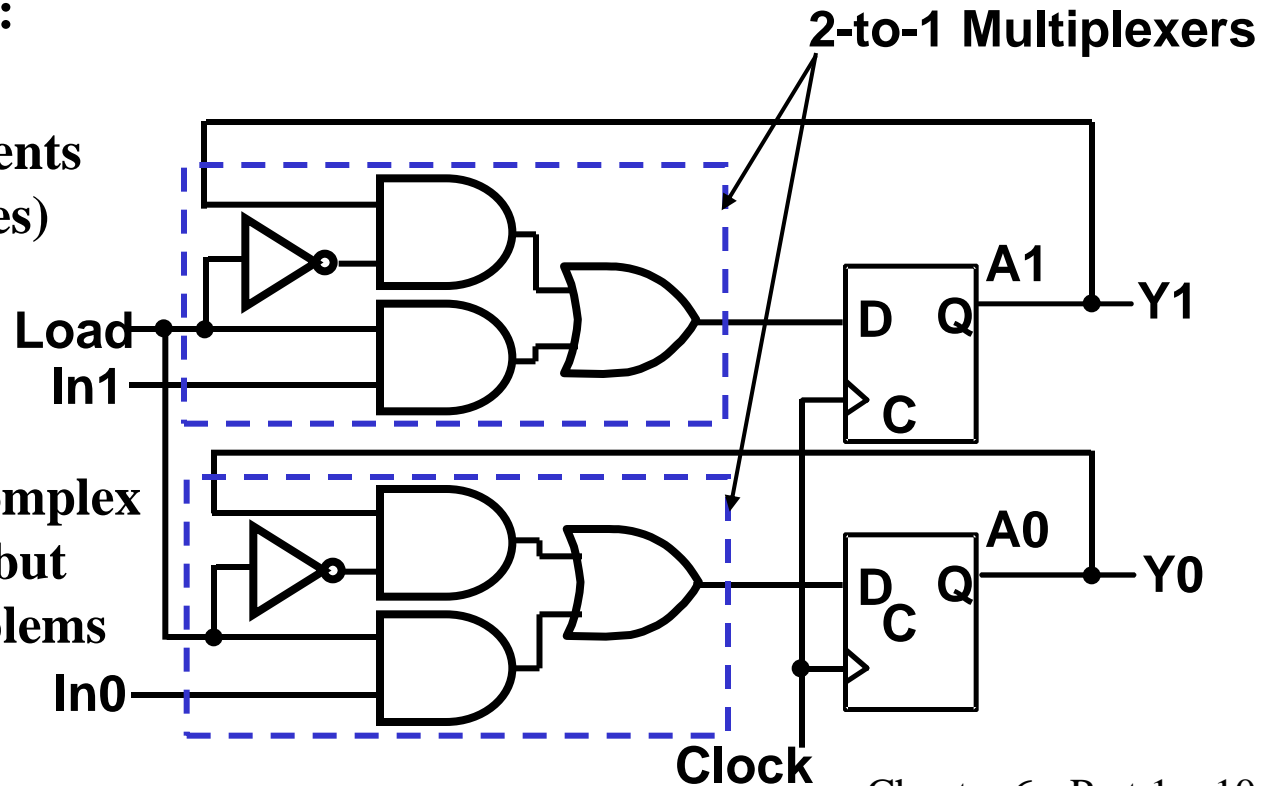
- A more reliable way to selectively load a register:
 - Run the clock continuously, and
 - Selectively use a load control to **change the register contents**.

- Example: 2-bit register with Load Control:

- For Load = 0, loads register contents (hold current values)

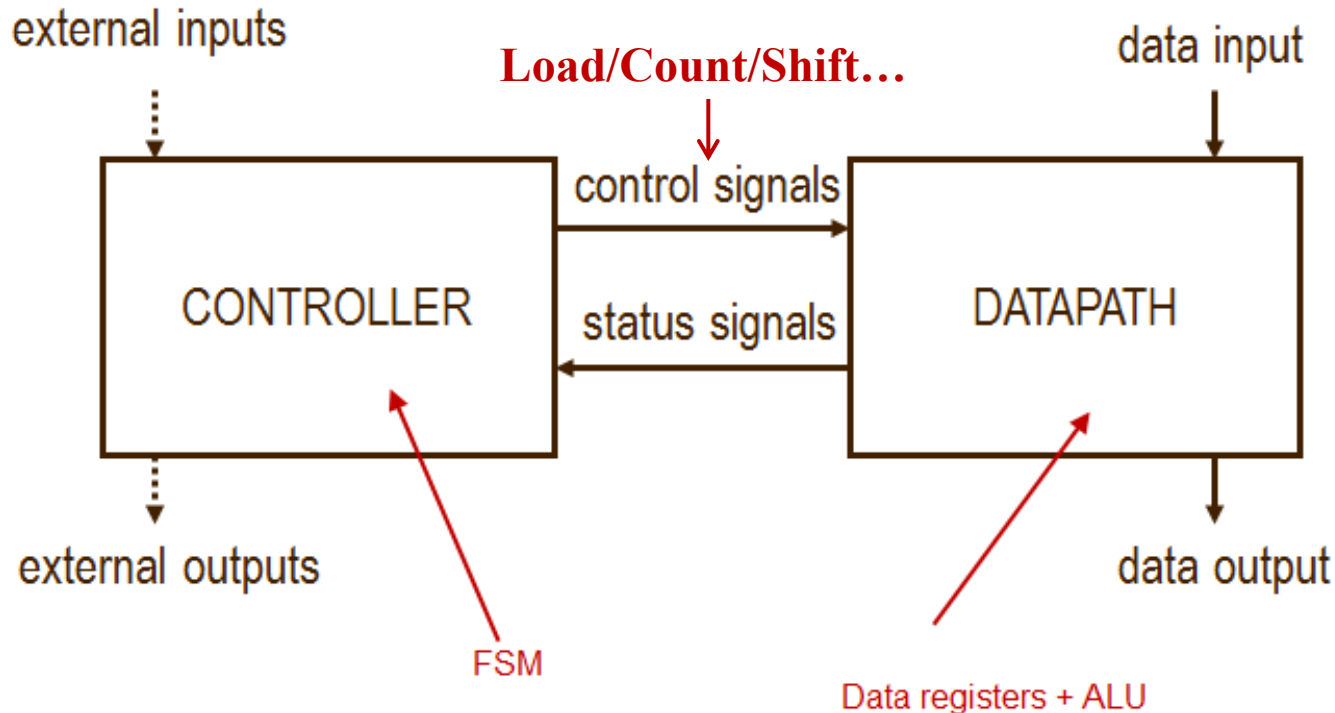
- For Load = 1, loads input values (load new values)

- Hardware more complex than clock gating, but free of timing problems



Introduction to Register Transfer Systems

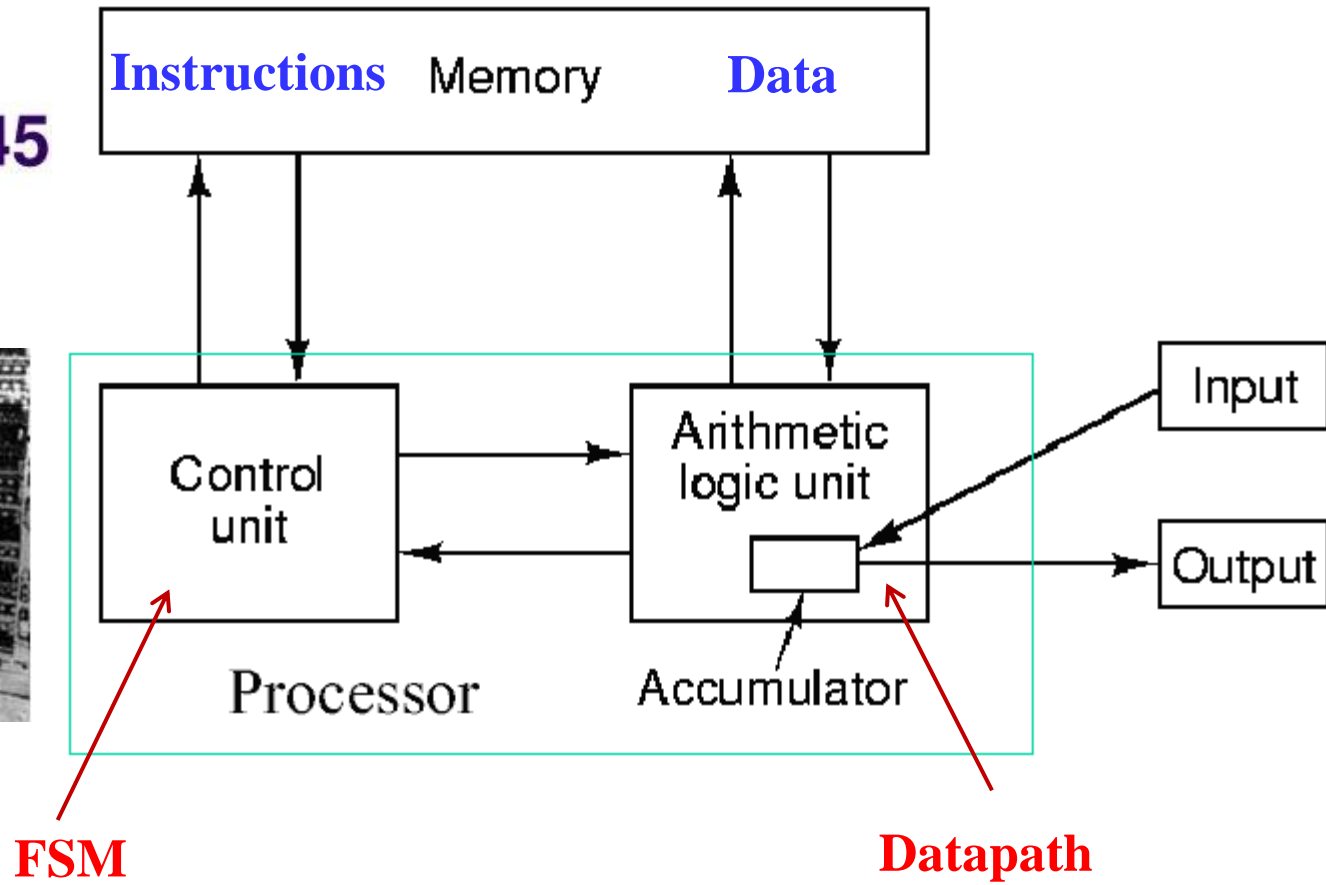
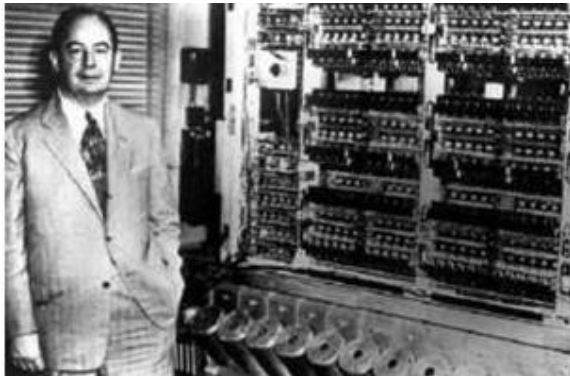
▪ Datapath and Control Unit



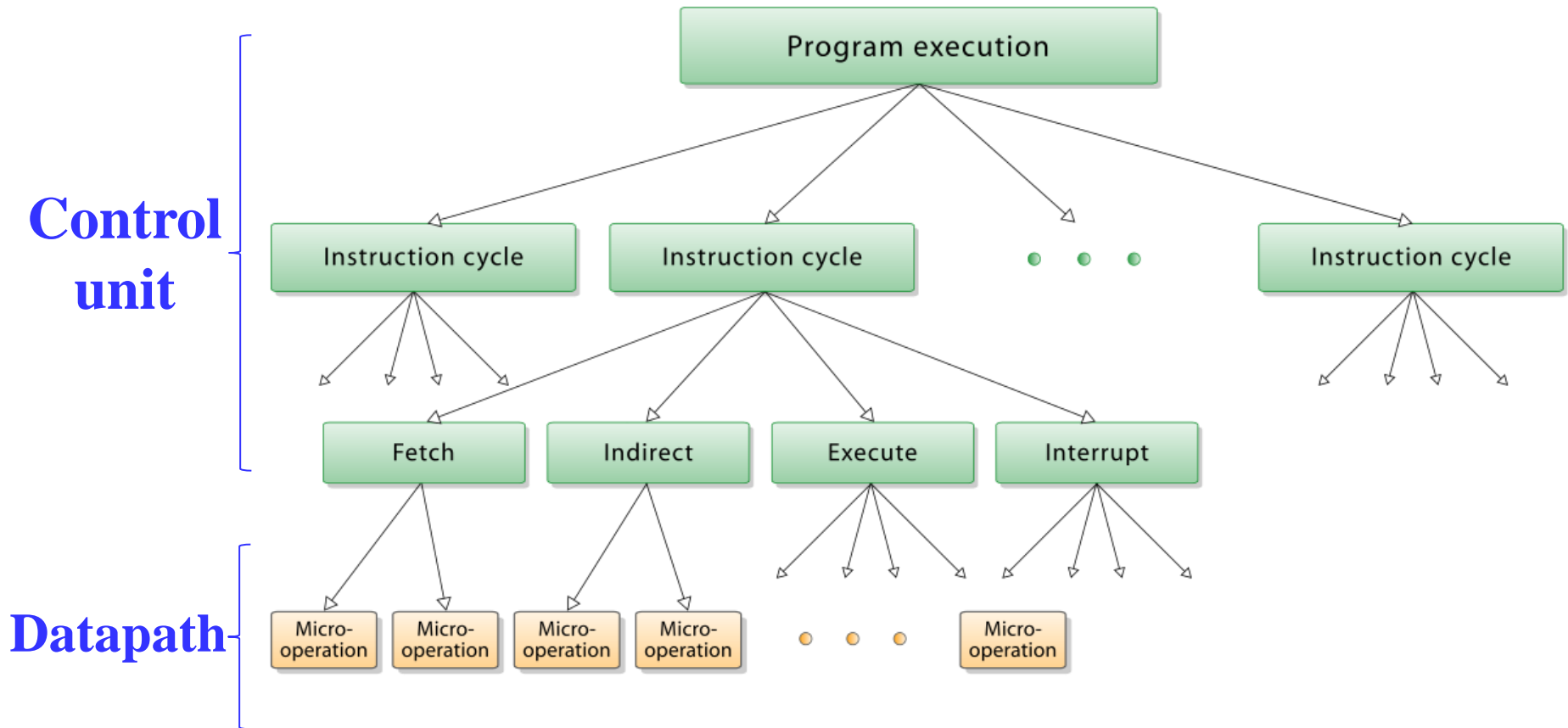
- **Datapath** is a set of functional units (e.g., ALU, MUX) that perform data processing operations, registers, and buses.
- The **control unit** determines the operation of the datapath by activating switches and passing control signals to specify how the data flows through the datapath.

Structure of von neumann machine

Von Neumann architecture 1945

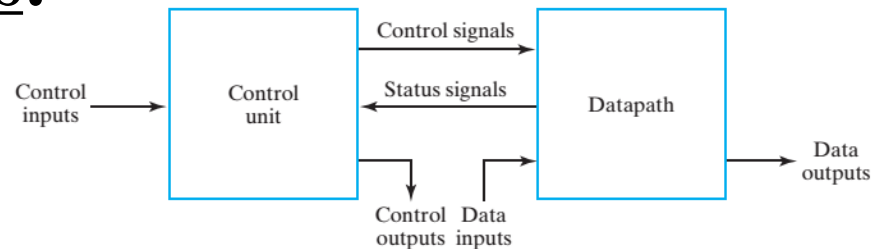


Decomposition of machine instructions

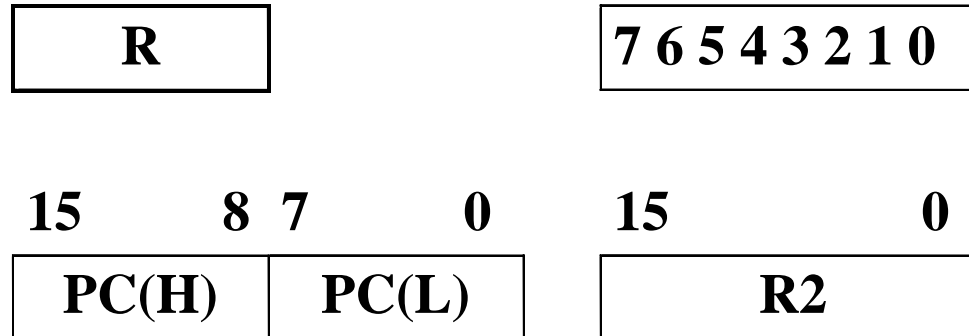


Register Transfer Operations

- 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 Language: RTL**



Register Notation



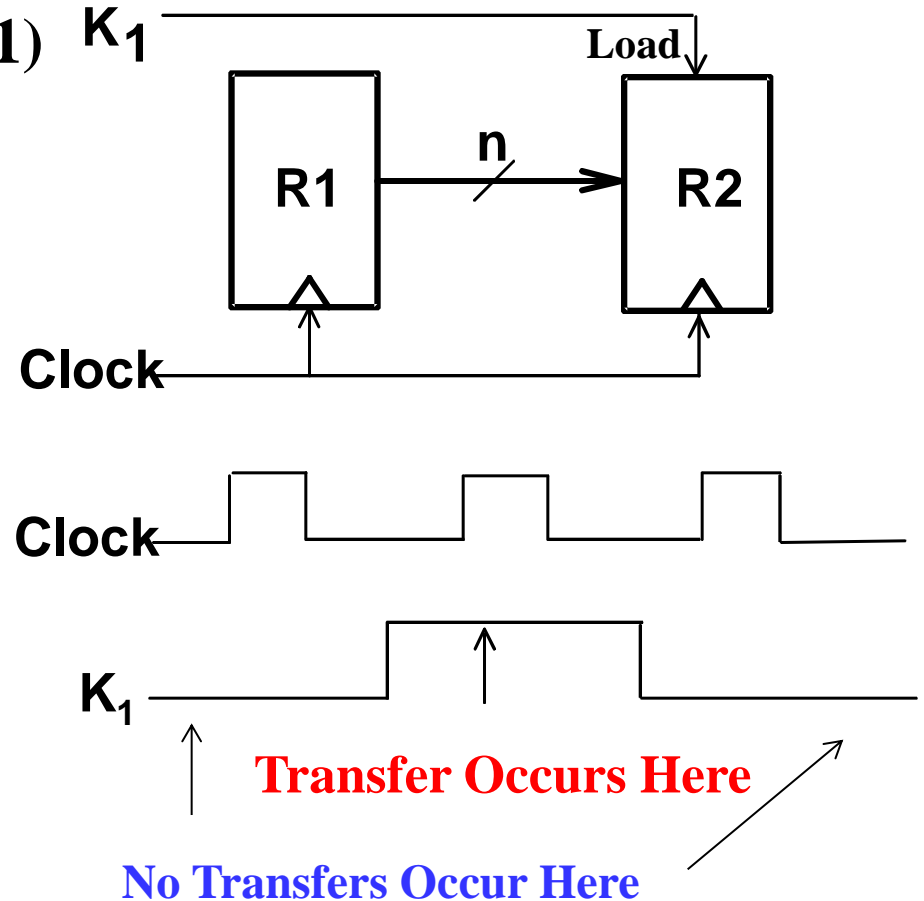
- ❑ Letters and numbers – denotes a register
 - ex. R2, PC, IR
- ❑ Parentheses () – denotes a range of register bits
 - ex. R1(1), PC(7:0), PC(L)
- ❑ Arrow (\leftarrow) – denotes data transfer
 - ex. $R1 \leftarrow R2$, $PC(L) \leftarrow R0$
- ❑ Comma – separates parallel operations
 - ex. $R1 \leftarrow R2$, $R2 \leftarrow R1$
- ❑ Brackets [] – Specifies a memory address
 - ex. $R0 \leftarrow M[AR]$, $R3 \leftarrow M[PC]$

Conditional Transfer

- If $(K1 = 1)$ then $(R2 \leftarrow R1)$ is shortened to

$K1: (R2 \leftarrow R1)$

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



Microoperations

■ Microoperations Groupings:

- **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

∨ Logical OR
∧ Logical AND
⊕ Logical Exclusive OR
– Not

Example Microoperations

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

$$PC \leftarrow R1 + R2$$

- Multiply the content of R1 by the content of R6 and place the result in R1.

$$R1 \leftarrow R1 * R6$$

- 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."

RTL, VHDL, 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	—	not	~
Shift left (logical)	sl	sll	<<
Shift right (logical)	sr	srl	>>
Vectors/Registers	A(3:0)	A(3 downto 0)	A[3:0]
Concatenation		&	{ , }

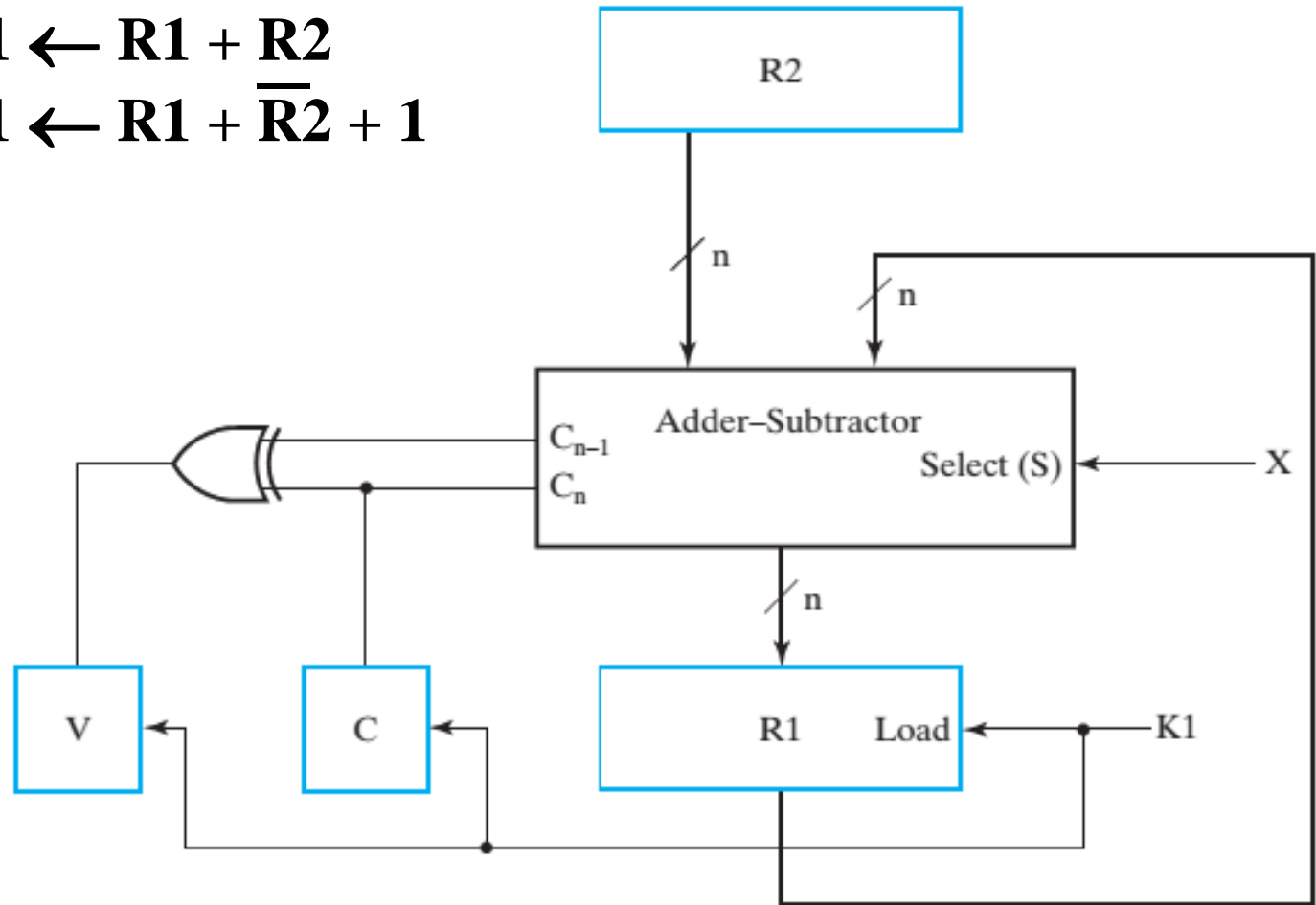
Control Expressions

- The control expression for an operation appears to the left of the operation and is separated from it by a colon
 - Control expressions specify the logical condition for the operation to occur
 - Control expression values of:
 - Logic "1" -- the operation occurs.
 - Logic "0" -- the operation is does not occur.
- Example:
 $\overline{X} \text{ K1} : R1 \leftarrow R1 + R2$
 $X \text{ K1} : R1 \leftarrow R1 + \overline{R2} + 1$
 - Variable K1 enables the add or subtract operation.
 - If $X = 0$, then $\overline{X} = 1$ so $\overline{X} \text{ K1} = 1$, activating the addition of R1 and R2.
 - If $X = 1$, then $X \text{ K1} = 1$, activating the addition of R1 and the two's complement of R2 (subtract).

Implementation of Add and Subtract Microoperations

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

$X K1 : R1 \leftarrow R1 + \overline{R2} + 1$



Arithmetic Microoperations

- **From Table 6-3:**

Symbolic Designation	Description
$R0 \leftarrow R1 + R2$	Addition
$R0 \leftarrow \overline{R1}$	Ones Complement
$R0 \leftarrow \overline{R1} + 1$	Two's Complement
$R0 \leftarrow R2 + \overline{R1} + 1$	R2 minus R1 (2's Comp)
$R1 \leftarrow R1 + 1$	Increment (count up)
$R1 \leftarrow R1 - 1$	Decrement (count down)
- Note that any register may be specified for source 1, source 2, or destination.
- These simple microoperations operate on the whole word

Logical Microoperations

- From Table 6-4:

Symbolic Designation	Description
$R0 \leftarrow \overline{R1}$	Bitwise NOT
$R0 \leftarrow R1 \vee R2$	Bitwise OR (sets bits)
$R0 \leftarrow R1 \wedge R2$	Bitwise AND (clears bits)
$R0 \leftarrow R1 \oplus R2$	Bitwise EXOR (complements bits)

Logical Microoperations (continued)

- 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

- From Table 6-5:
- Let $R2 = 11001001$
- Then after the operation, $R1$ becomes:

Symbolic Designation	Description
$R1 \leftarrow sl\ R2$	Shift Left
$R1 \leftarrow sr\ R2$	Shift Right

R1	Operation
10010010	$R1 \leftarrow sl\ R2$
01100100	$R1 \leftarrow sr\ R2$

- 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) (see Chapter 9).

Register Transfer Structures

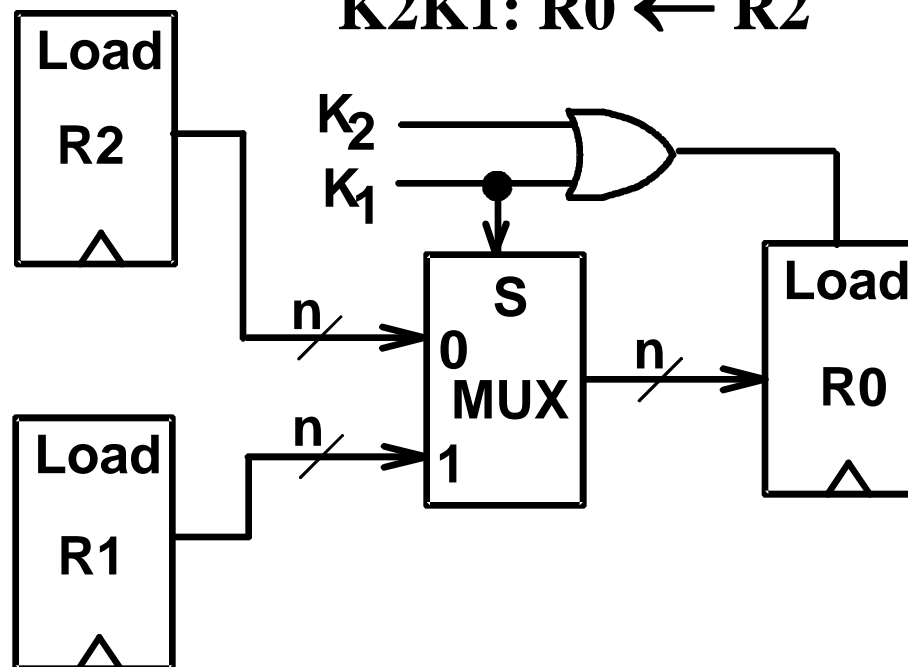
- **Multiplexer-Based Transfers** - Multiple inputs are selected by a multiplexer dedicated to the register
- **Bus-Based Transfers** - Multiple inputs are selected by a shared multiplexer driving a bus that feeds inputs to multiple registers
- **Three-State Bus** - Multiple inputs are selected by 3-state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

$K_1: R0 \leftarrow R1$

$K_2\bar{K}_1: R0 \leftarrow R2$

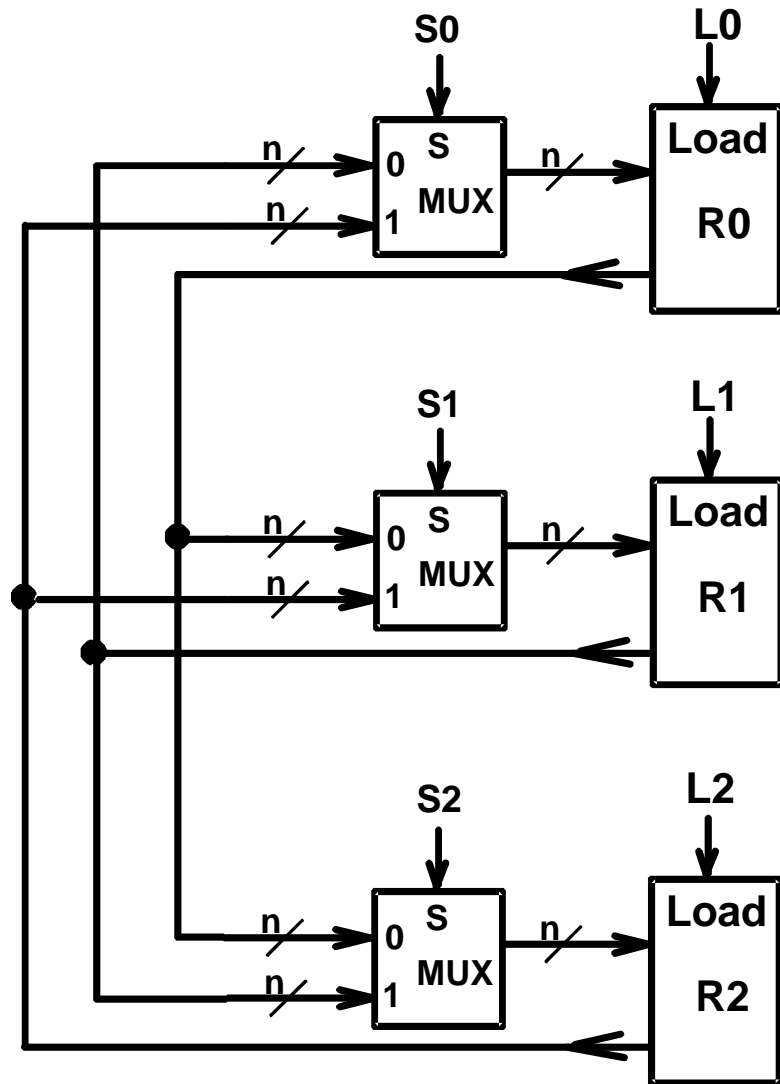


Multiplexer and Bus-Based Transfers for Multiple Registers

- **Multiplexer dedicated to** each register
- **Shared transfer paths** for registers
 - A shared transfer object is 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

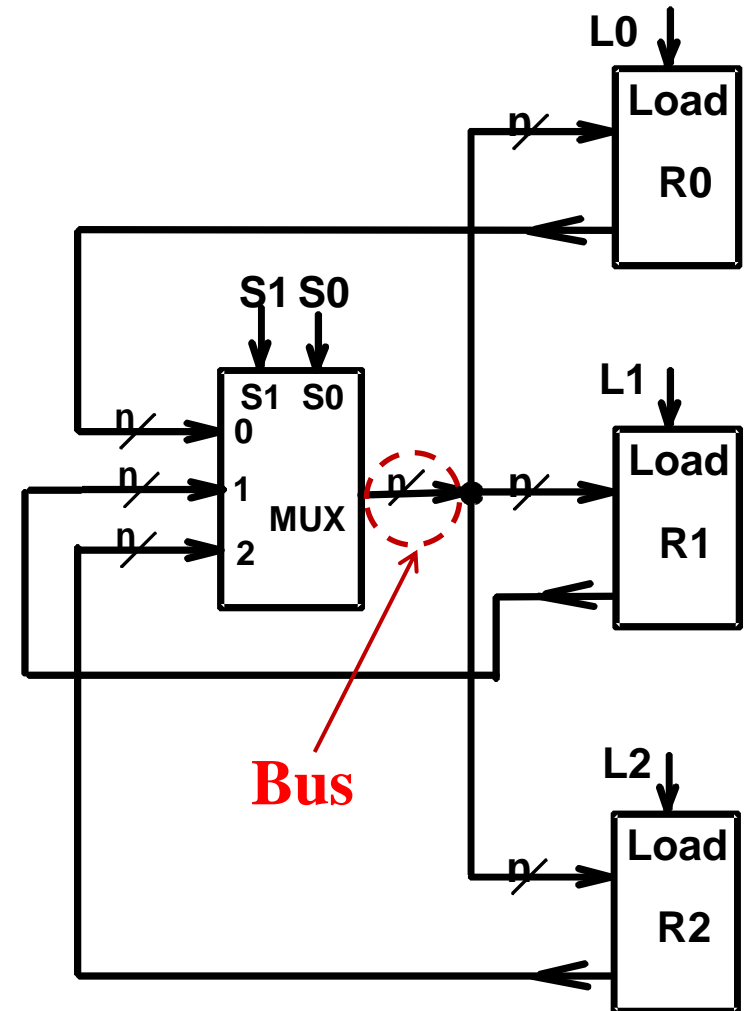
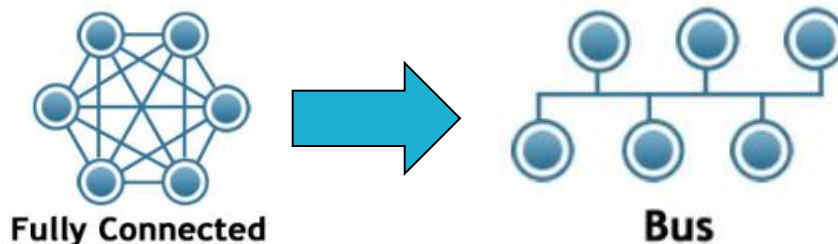
Dedicated MUX-Based Transfers

- Multiplexer connected to each register input produces a very flexible transfer structure.
- Characterize the simultaneous transfers with this structure.
- 18 gate inputs **per bit** plus 3 shared inverters with total of 3 inputs



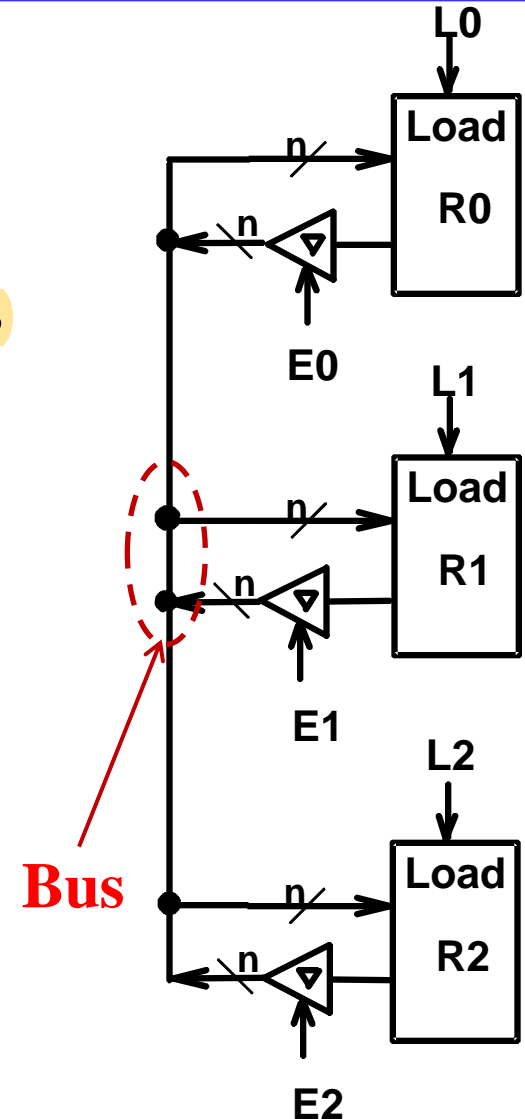
Multiplexer Bus

- 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
- shared decoder with 8 inputs + 9 gate inputs **per bit**



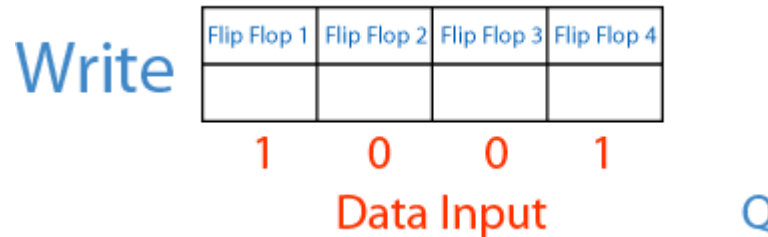
Three-State Bus

- 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
- 3 gate inputs per three state driver = 9 gate inputs
- Other advantages?

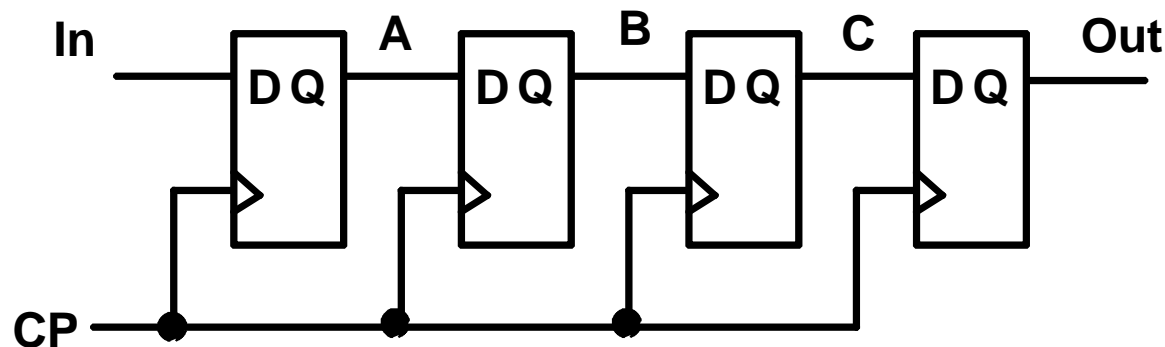


Shift Registers

- Shift Registers move data laterally within the register toward its MSB or LSB position



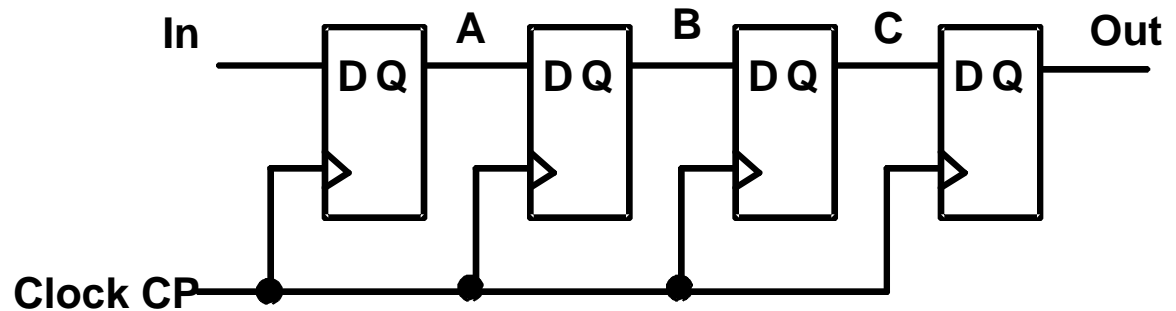
- In the simplest case, the shift register is simply a set of D flip-flops connected in a row like this:



- 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)

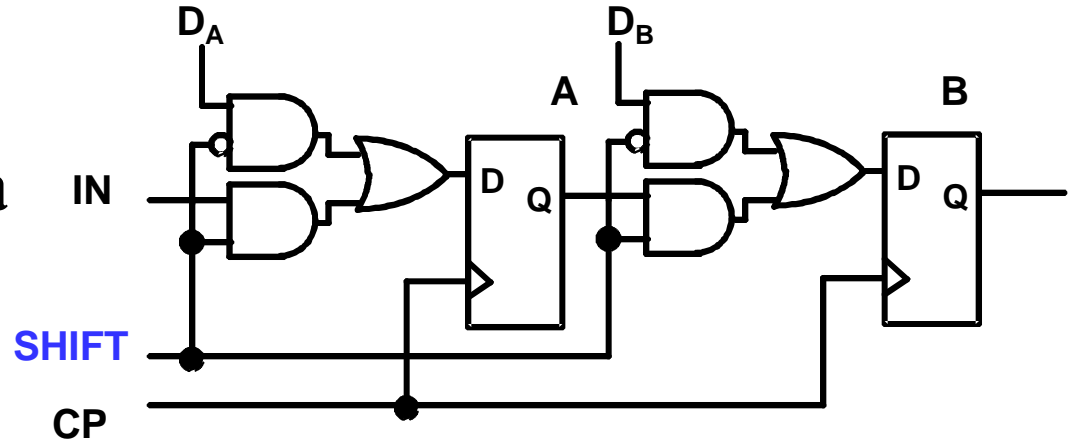
- The behavior of the serial shift register is given in the listing on the lower right
- T0 is the register state just before the first clock pulse occurs
- T1 is after the first pulse and before the second.
- Initially unknown states are denoted by “?”
- Complete the last three rows of the table



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

Parallel Load Shift Registers

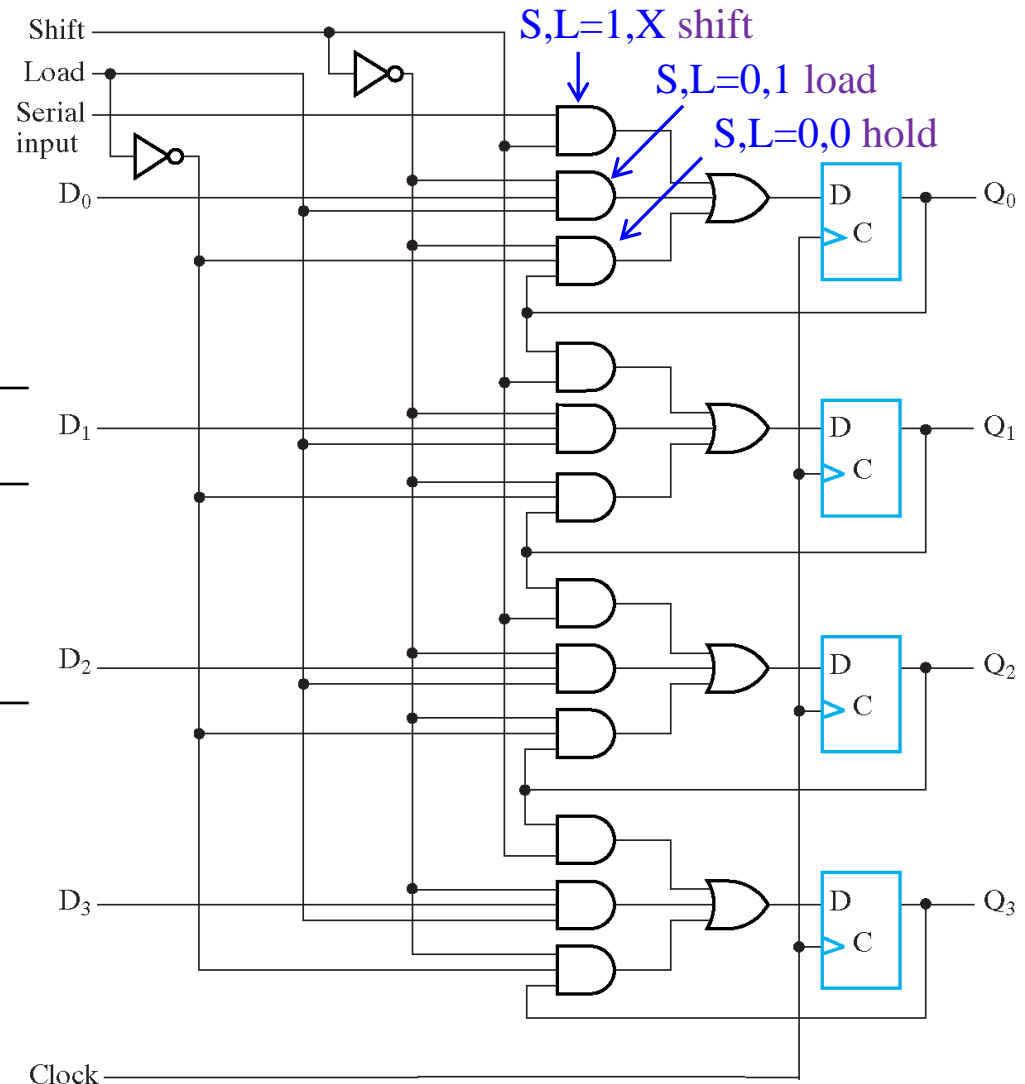
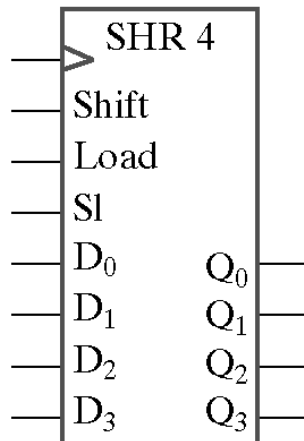
- By adding a MUX between each shift register stage, data can be shifted or loaded
- If SHIFT is low, A and B are replaced by the data on D_A and D_B lines, else data shifts right on each clock.
- By adding more bits, we can make n -bit parallel load shift registers.



Parallel Load Shift Registers (continued)

- A parallel load shift register with an added “**hold**” operation that stores data:

Shift	Load	Operation
0	0	No Change
0	1	Load
1	x	Shift down



- 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.**

The diagram illustrates a 3-bit shift register implemented using a 4-to-1 Multiplexer (MUX) and three D flip-flops. The MUX has four data inputs: S_1 , S_0 , 0, and 1, and a select input D_i . The MUX output is connected to the D input of the top flip-flop, which produces output Q_{i-1} . The output of the top flip-flop is connected to the D input of the middle flip-flop, which produces output Q_i . The output of the middle flip-flop is connected to the D input of the bottom flip-flop, which produces output Q_{i+1} . All three flip-flops share a common clock input labeled "Clock".

Shift Registers with Additional Functions (continued)

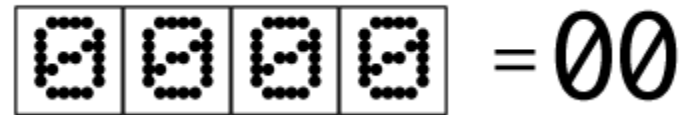
- Shift registers can also be designed to shift more than a single bit position right or left.
- Shift registers can be designed to shift a variable number of bit positions specified by a variable called a *shift amount*.

Overview

- **Part 1 – Registers, Microoperations and Implementations**
- **Part 2 – Counters, register cells, buses, & serial operations**
 - **Microoperations on single register (continued)**
 - **Counters**
 - **Register cell design**
 - **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:

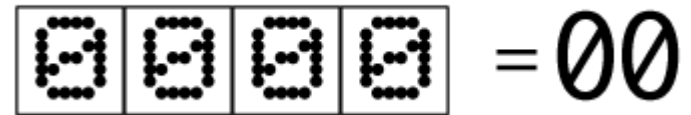


- **Ripple Counters**
 - Clock connected to the flip-flop clock input on the LSB bit flip-flop
 - For all other bits, a flip-flop output is connected to the clock input, thus circuit is not truly synchronous!
 - Output change is delayed more for each bit toward the MSB.
 - Resurgent because of low power consumption
- **Synchronous Counters**
 - Clock is directly connected to the flip-flop clock inputs
 - Logic is used to implement the desired state sequencing

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:



- Synchronous Counters

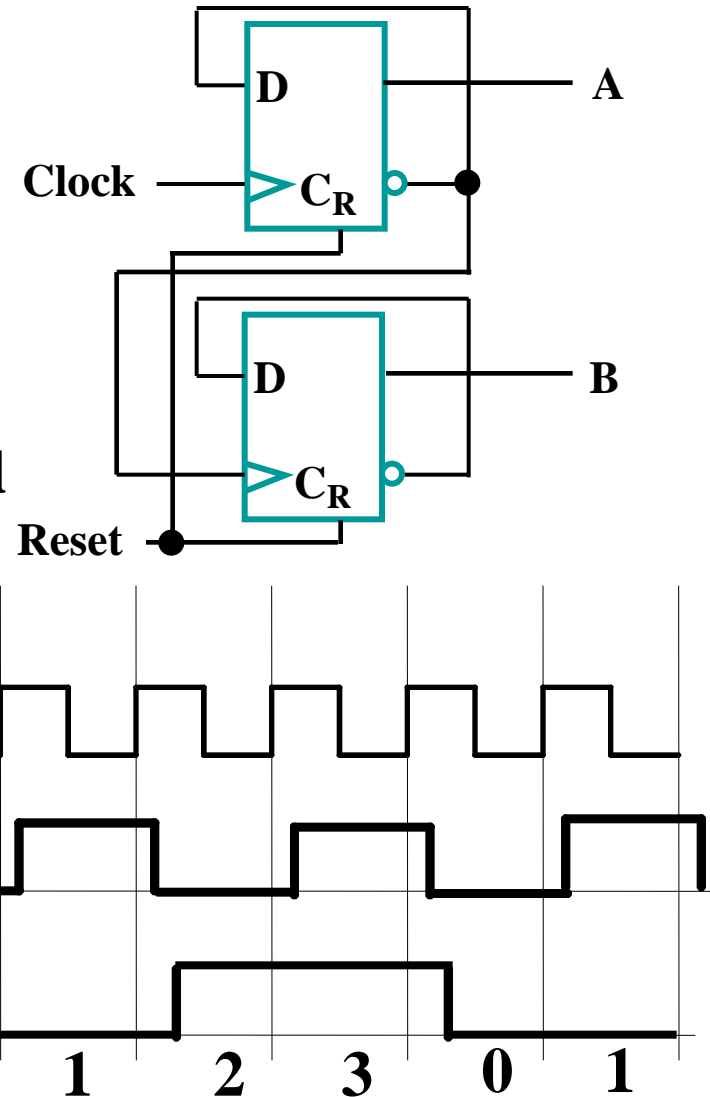
- Clock is directly connected to the flip-flop clock inputs
- Logic is used to implement the desired state sequencing

- 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

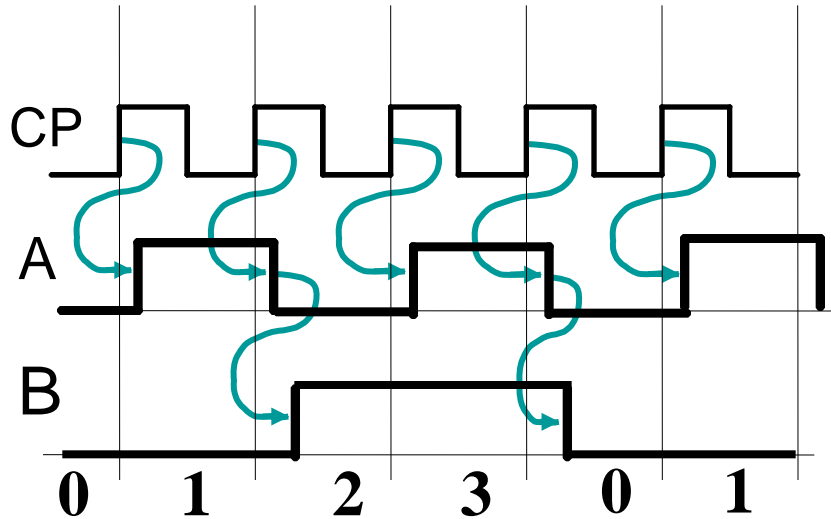
Ripple Counter

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

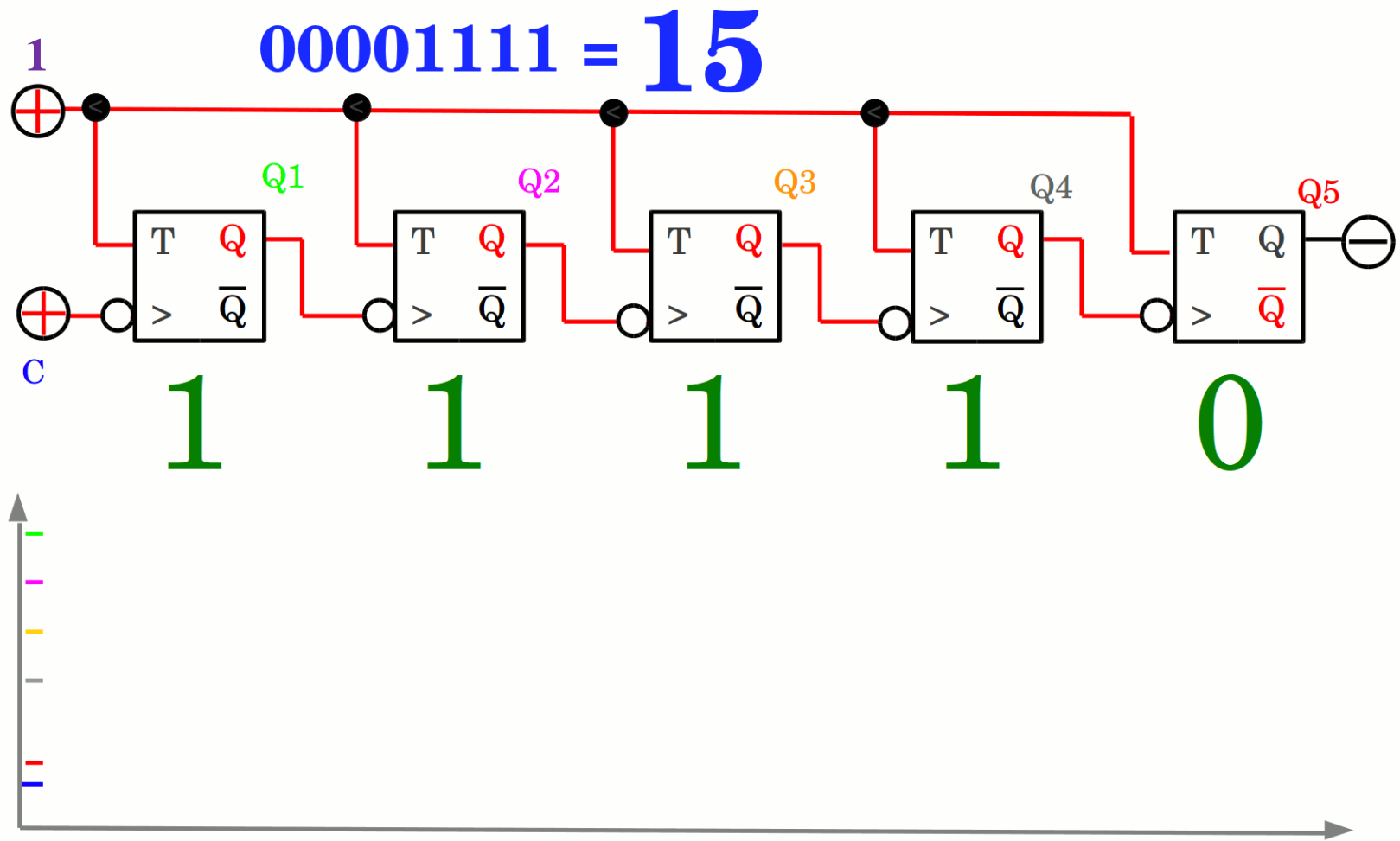


Ripple Counter (continued)

- 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)

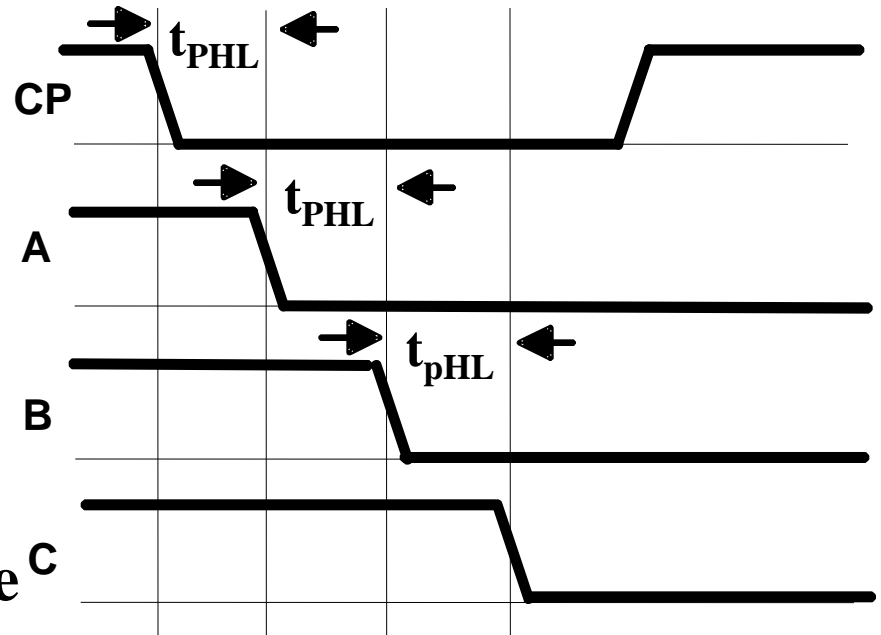
- ripple counters: 01111 (15) \rightarrow 10000 (16)



Ripple Counter (continued)

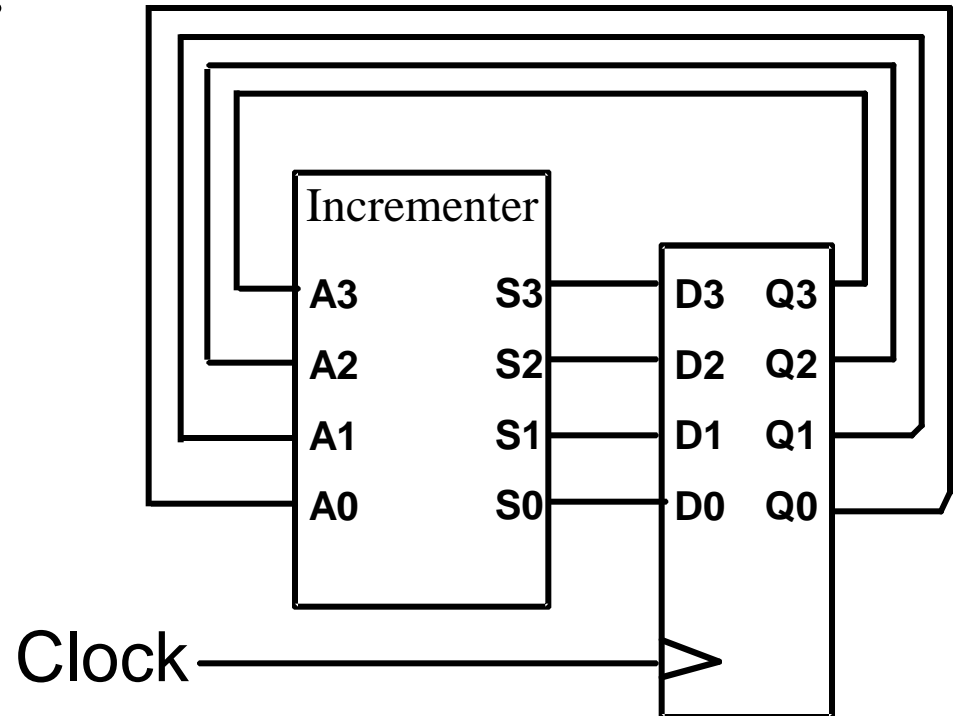
- Starting with $C = B = A = 1$, equivalent to $(C,B,A) = 7$ base 10, the next clock increments the count to $(C,B,A) = 0$ base 10. In fine timing detail:

- 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 $n t_{PHL}$.



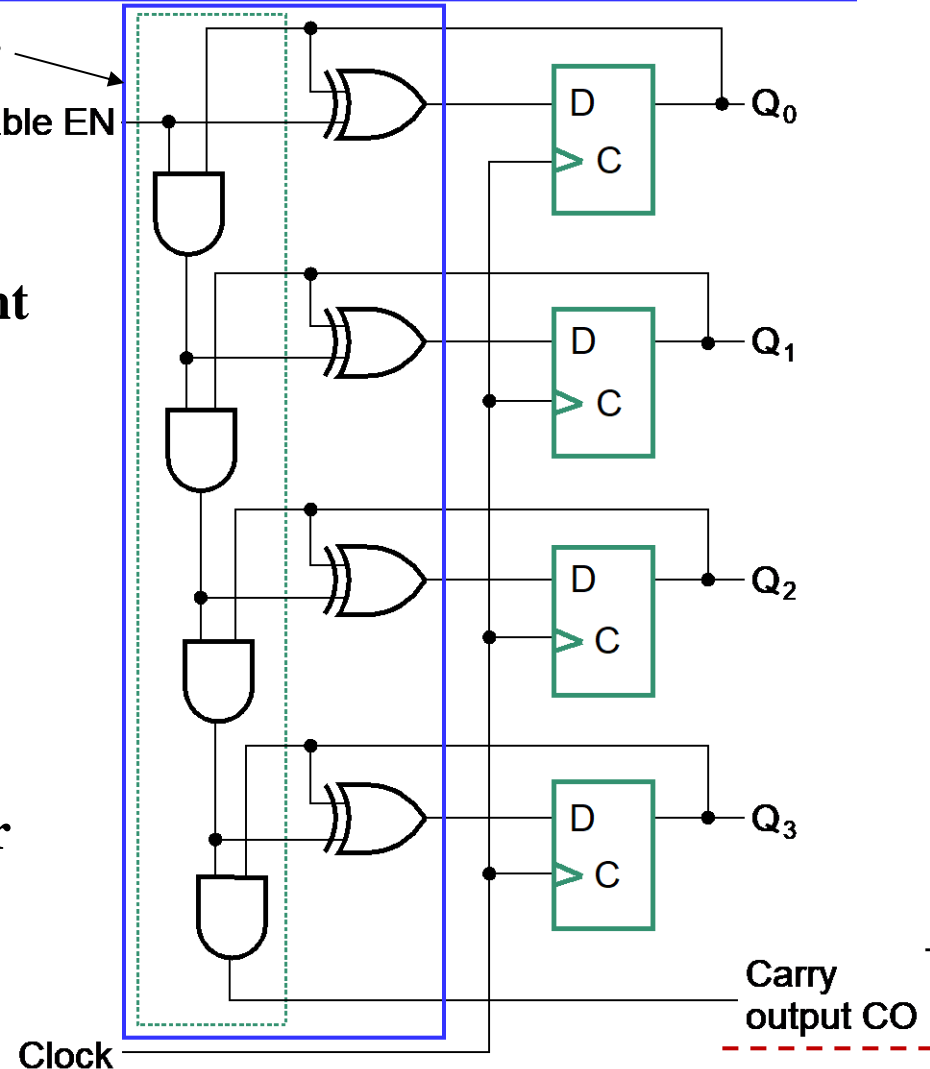
Synchronous Counters

- 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)

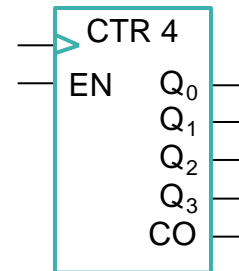
- **Internal details** => Incrementer – Count enable
- **Internal Logic**
 - **XOR** complements each bit
 - **AND** chain causes complement of a bit if all bits toward LSB from it equal 1
- **Count Enable**
 - Forces all outputs of **AND** chain to 0 to “hold” the state
- **Carry Out**
 - Added as part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



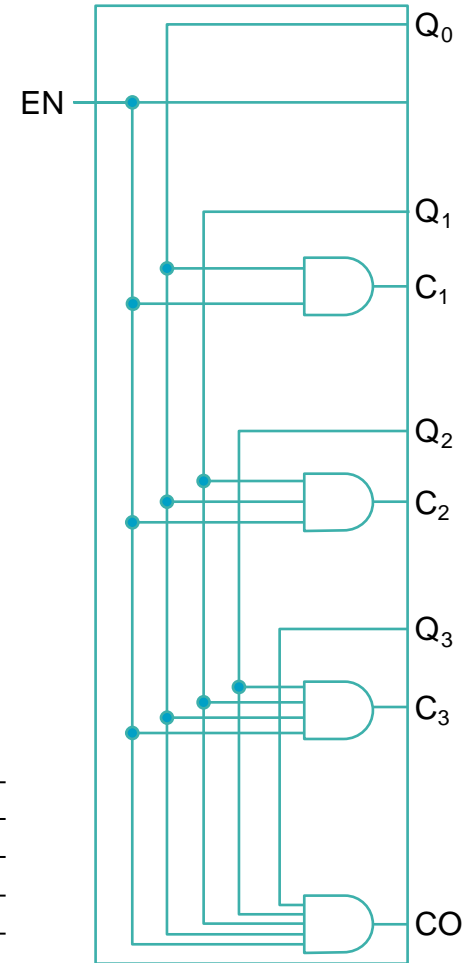
(a) Logic Diagram-Serial Gating

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**



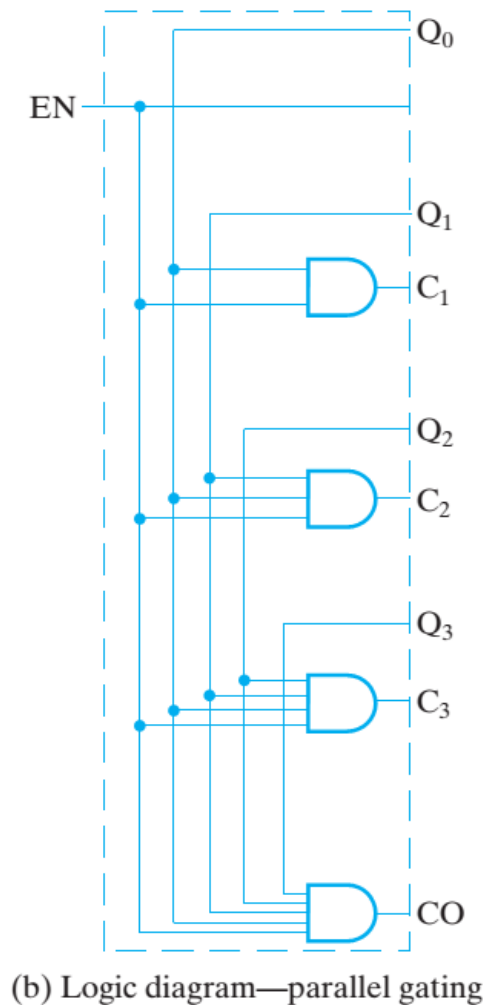
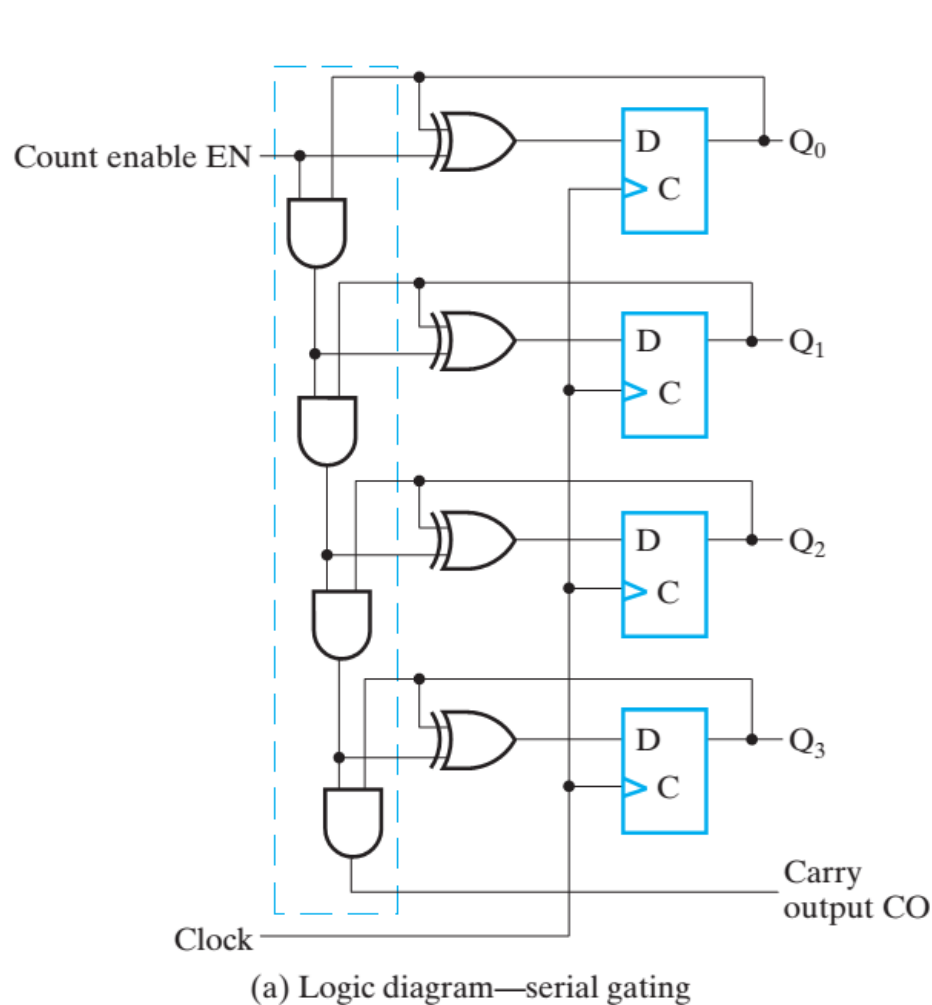
Symbol



Logic Diagram-Parallel Gating

Synchronous Counters (continued)

- Serial counter versus parallel counter



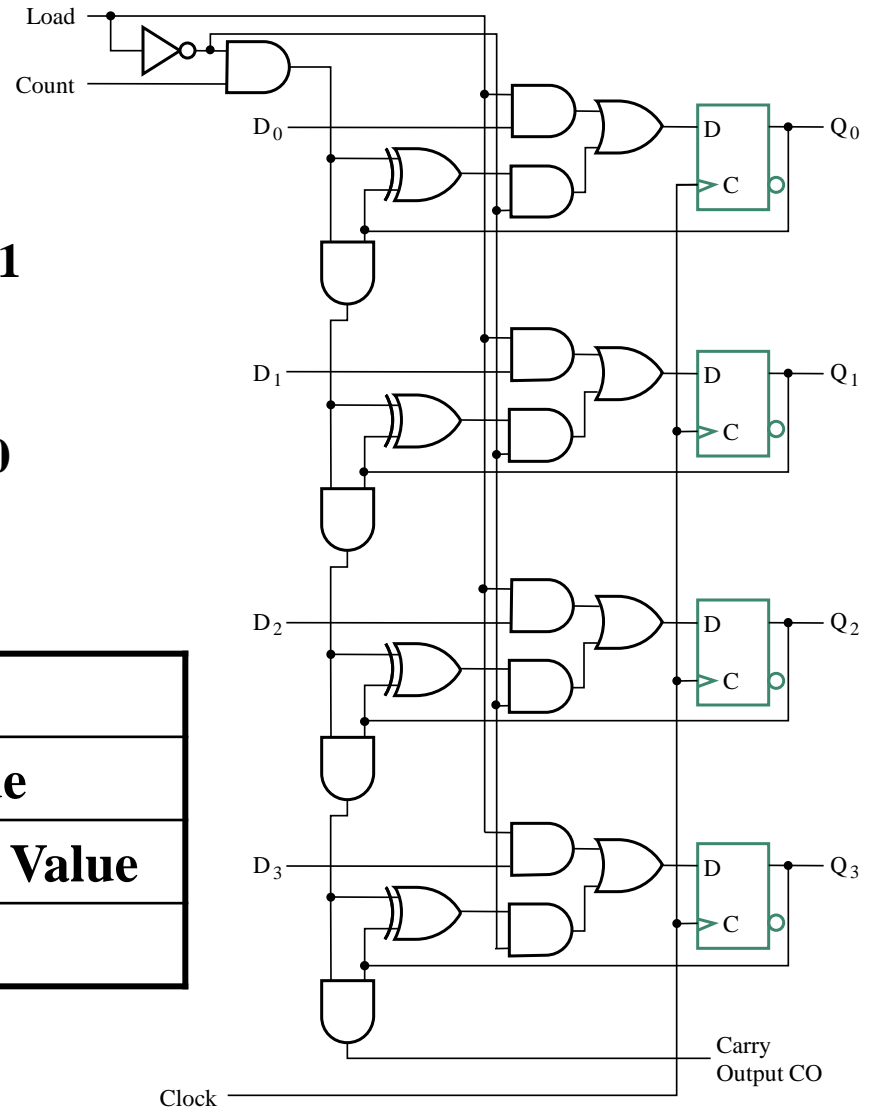
Other Counters

- 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/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
- State Table =>
- Input combinations 1010 through 1111 are don't cares

Current State				Next State			
Q8	Q4	Q2	Q1	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

Synchronous BCD (continued)

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

$$D1 = \overline{Q1}$$

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

$$D4 = Q4 \oplus Q1 Q2$$

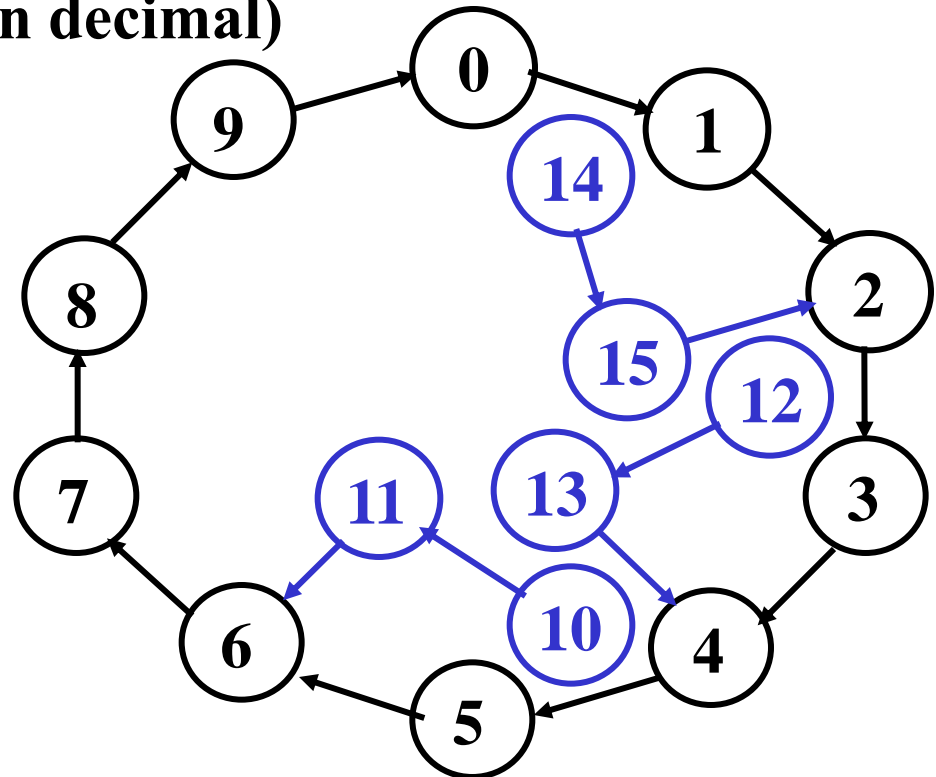
$$D8 = Q8 \oplus (Q1 Q8 + Q1 Q2 Q4)$$

- 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

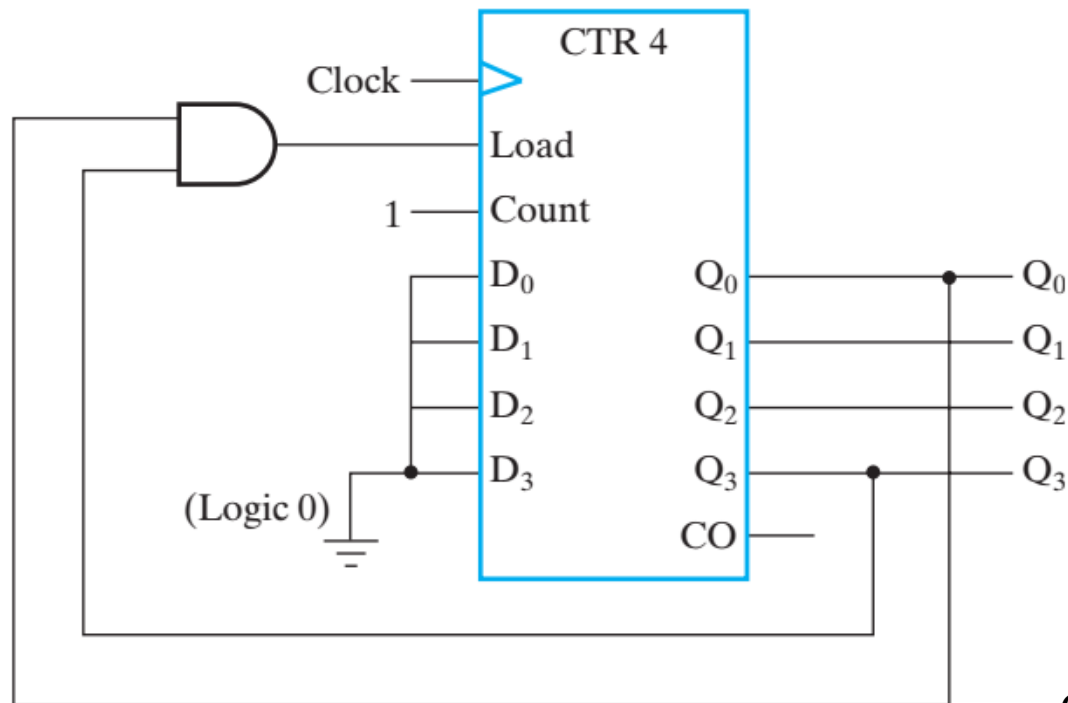


Synchronous BCD (continued)

- 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? **Error = $Q_8Q_4 + Q_8Q_2$**
 - 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

Synchronous BCD (continued)

- A binary counter with parallel load can be converted into a synchronous BCD counter with an external AND gate.
- When the output reaches the count of 1001, both Q0 and Q3 become 1, making the output of the AND gate equal to 1. This condition makes Load active.

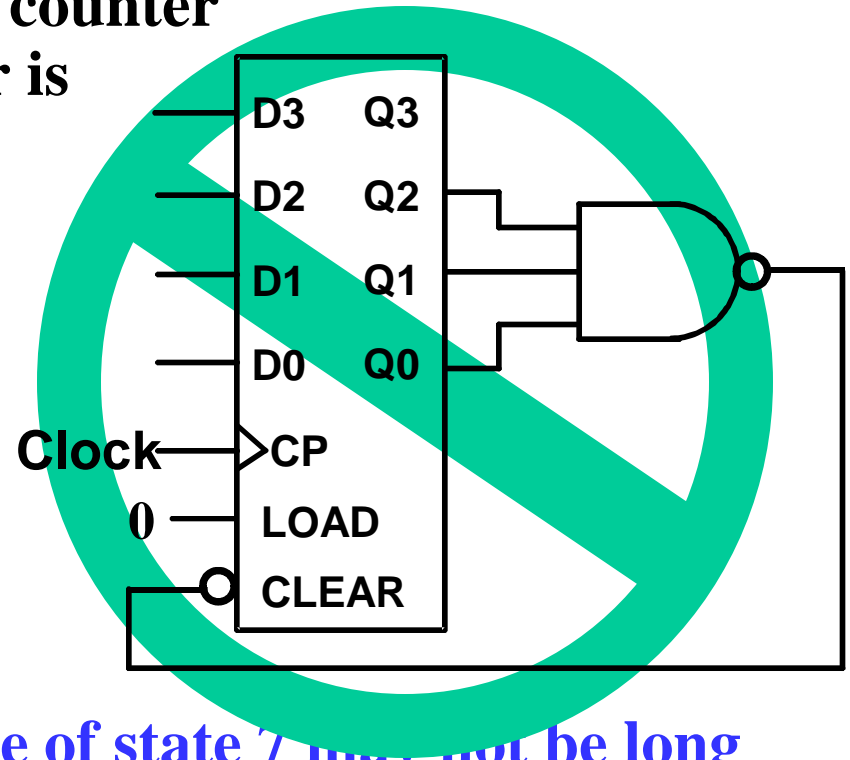


Counting Modulo N

- 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

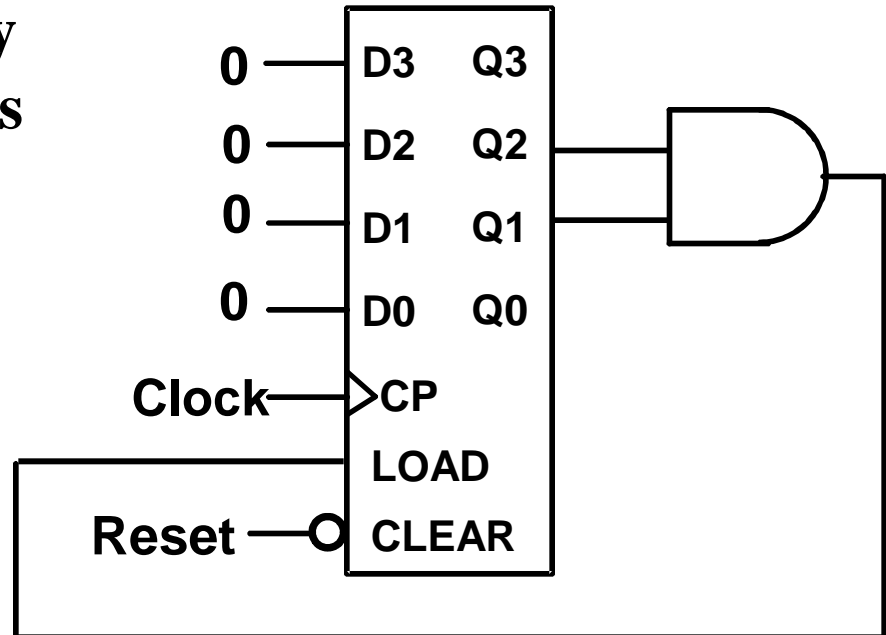
Counting Modulo 7: Detect 7 and Asynchronously Clear

- 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!)



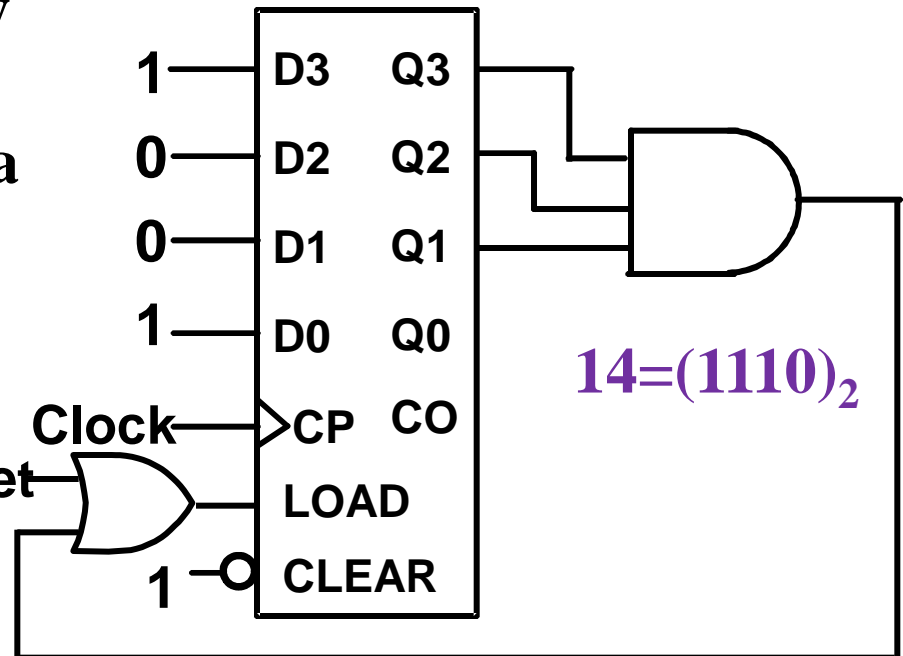
Counting Modulo 7: Synchronously Load on Terminal Count of 6

- 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 = Q2 Q1**



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)

Assignments

Reading:

- 6.1-6.3; 6.5-6.6; 4.13

Problem assignment:

- 6-6、 6-13、 6-16、 6-17、 6-19、 6-27、 6-34