Modular Sequential Circuits

COE 202

Digital Logic Design

Dr. Muhamed Mudawar

King Fahd University of Petroleum and Minerals

Presentation Outline

- ❖ Registers
- ❖ Shift Registers and their Applications
- ❖ Ripple Counters
- ❖ Synchronous Counters
- ❖ Memory and ROM

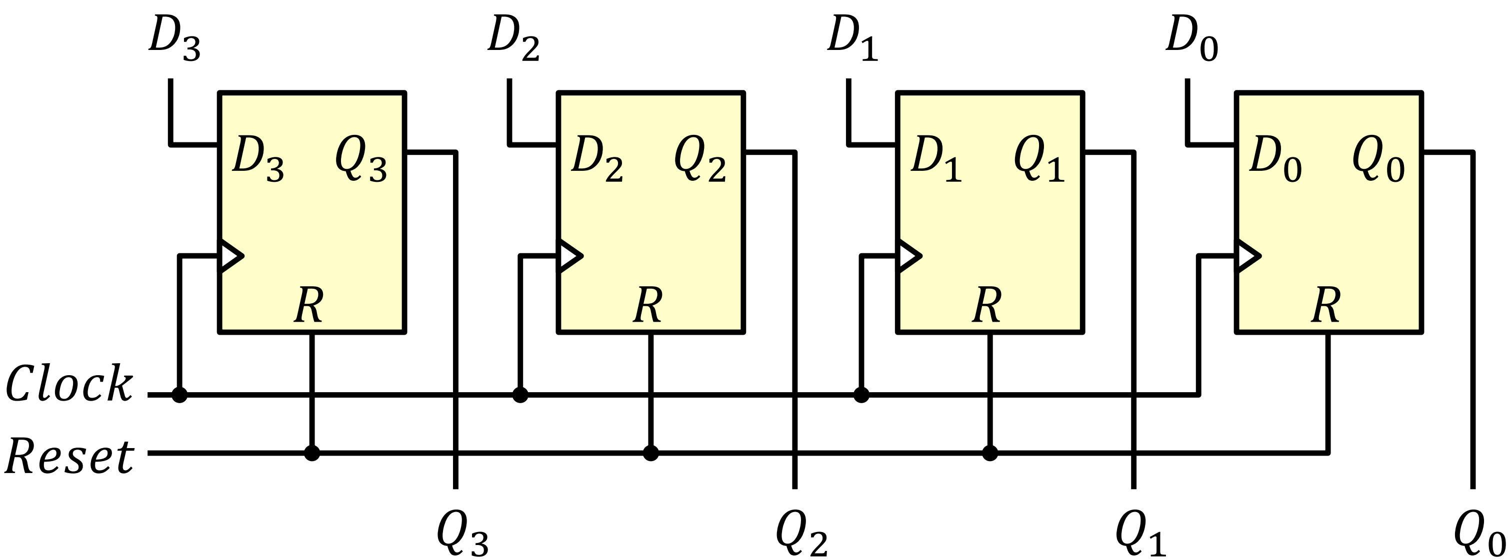
Register

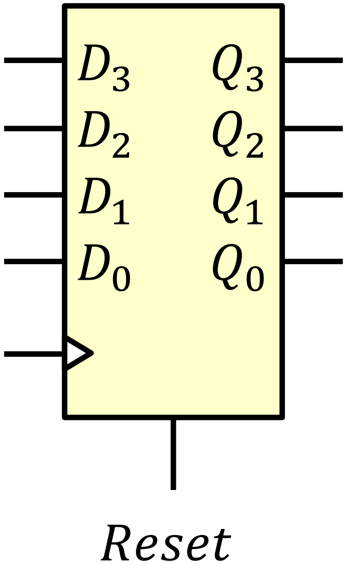
❖ A register is a circuit capable of storing data

❖ An n-bit register consists of n Flip-Flops and stores n bits

❖ Common clock: data is loaded in parallel at the same clock edge

❖ Common reset: All Flip-Flops are reset in parallel



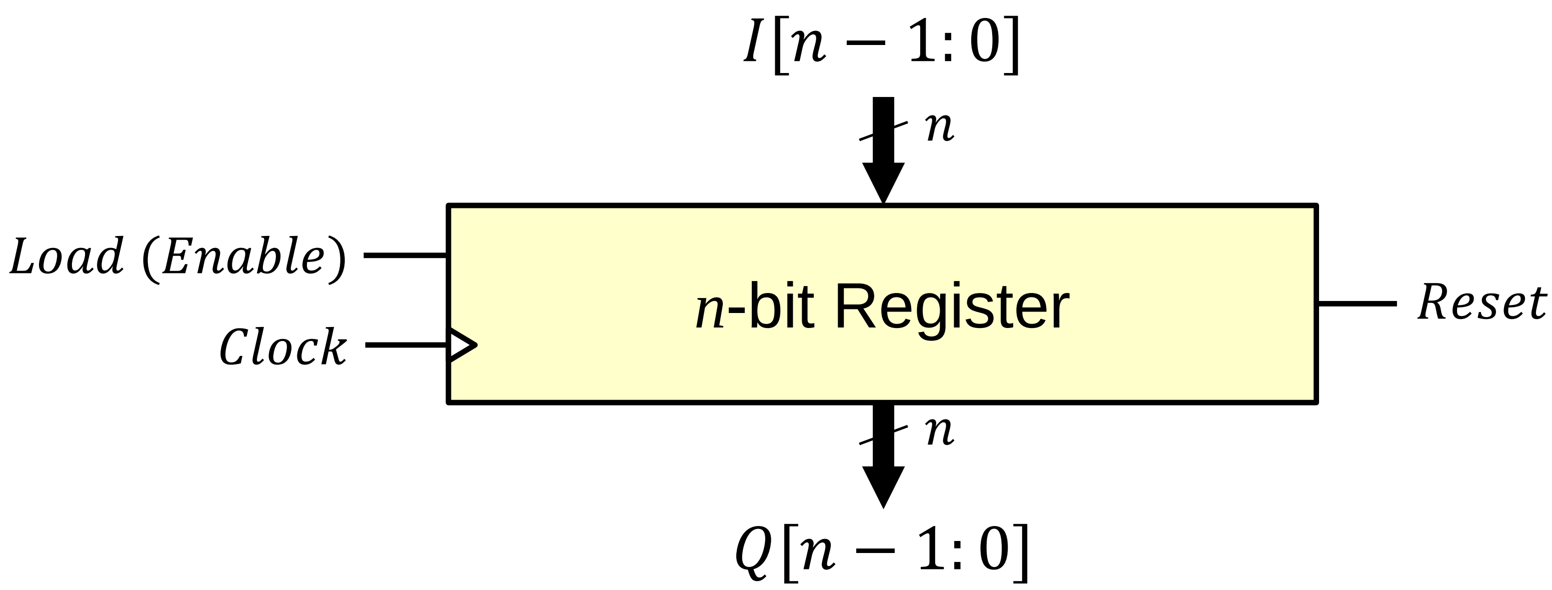


Register Load (or Enable)

- **Question:** How to control the loading of data into a register?
- **Solution:** Introduce a register Load (or Enable) signal

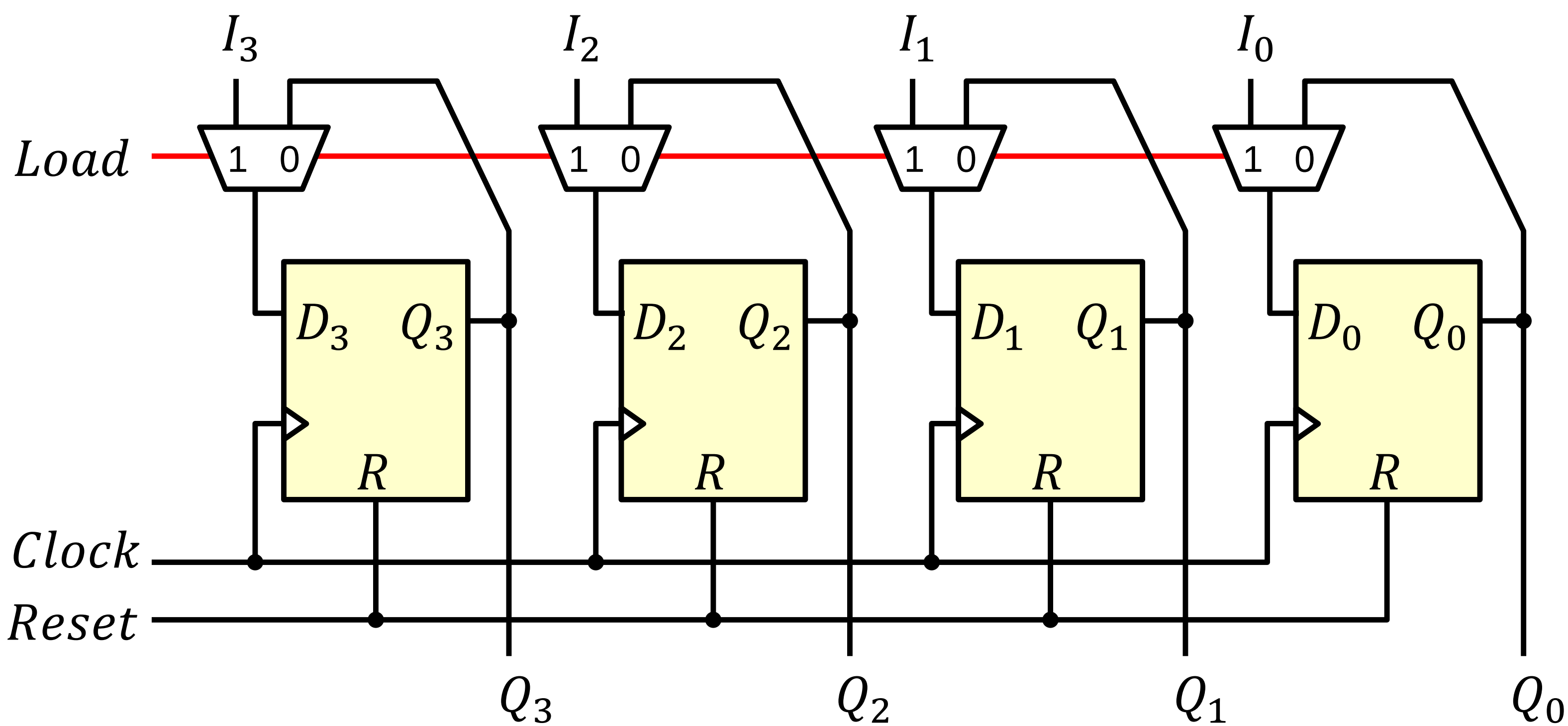
  If the register is enabled, load the data into the register

  Otherwise, do not change the value of the register
- **Question:** How to implement register Load?



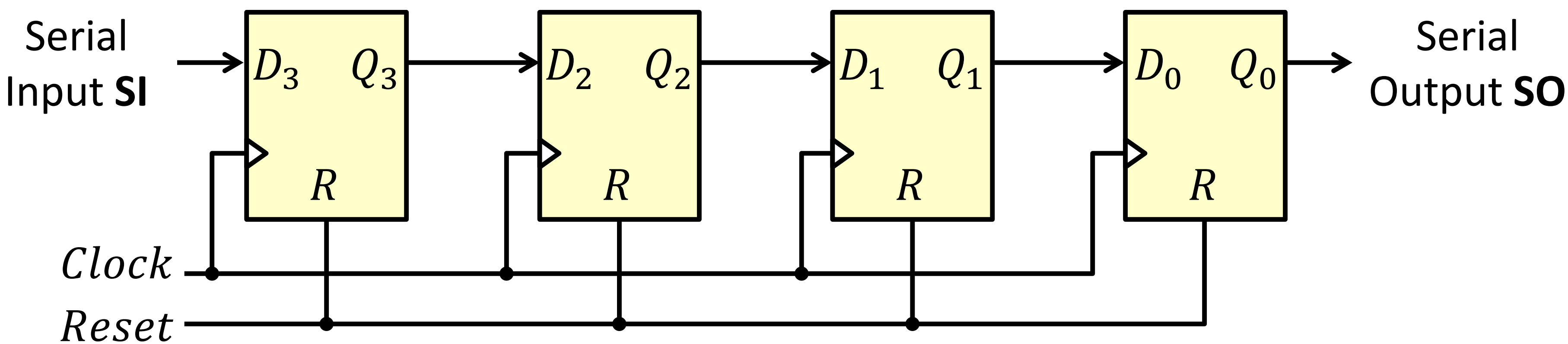
Register with Parallel Load

- **Solution:** Add a mux at the D input of the register
- $D_i = Load \cdot I_i + \overline{Load} \cdot Q_i$
- If $Load$ is **1** then $D_i = I_i$ If $Load$ is **0** then $D_i = Q_i$



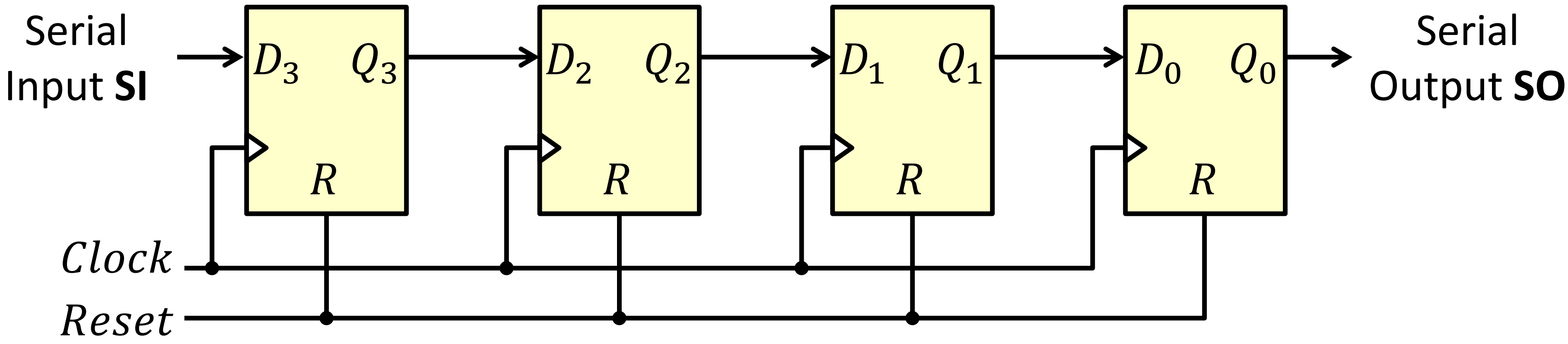
Shift Registers

- A shift register is a cascade of flip flops sharing the same clock
- Allows the data to be shifted from each flip-flop to its neighbor
- The output of a flip-flop is connected to the input of its neighbor
- Shifting can be done in either direction
- All bits are shifted simultaneously at the active edge of the clock



Right Shift Register

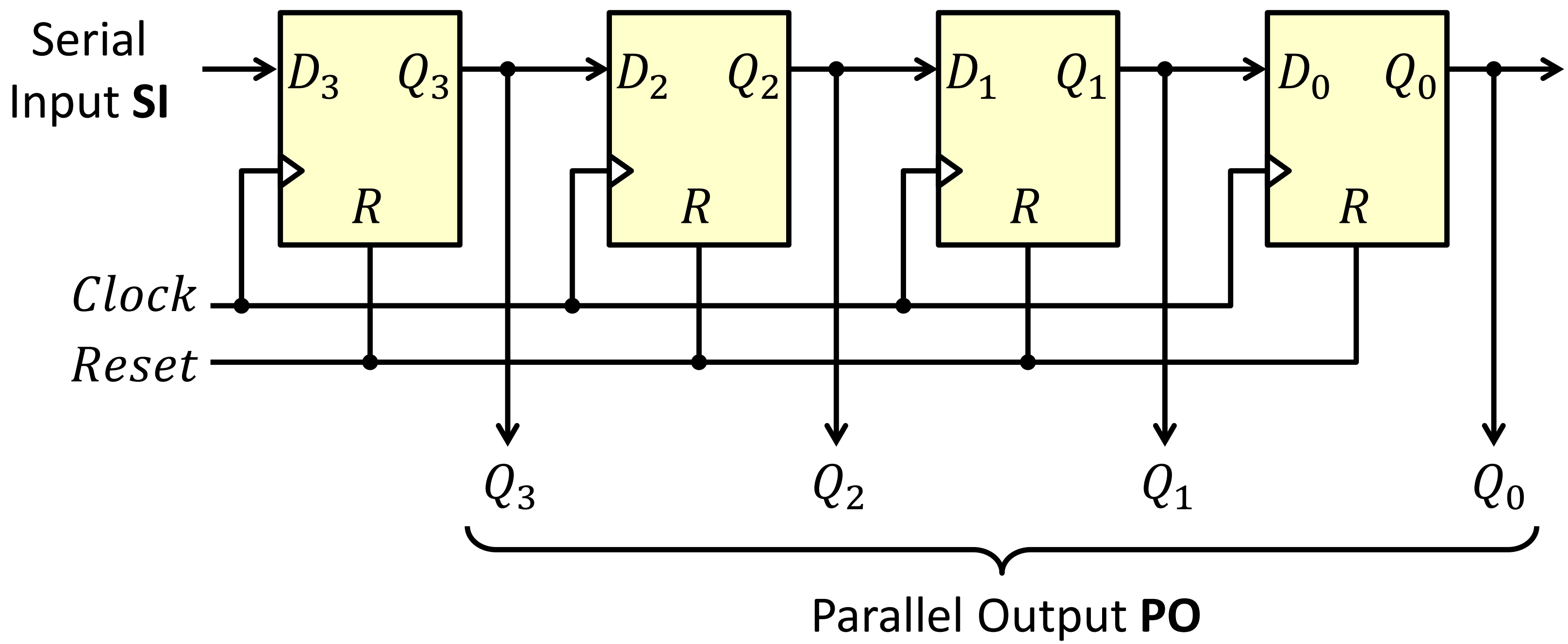
Timing of a Shift Register



Cycle	SI	Q3	Q2	Q1	Q0 = SO
T0	1	1	0	1	0
T1	0	1	1	0	1
T2	1	0	1	1	0
T3	1	1	0	1	1
T4	0	1	1	0	1
T5	1	0	1	1	0
T6	0	1	0	1	1

Shift Register with Parallel Output

- The output of a shift register can be serial or parallel
- A Serial-In Parallel-Out (SIPO) shift register is shown below
- All flip-flop outputs can be read in parallel

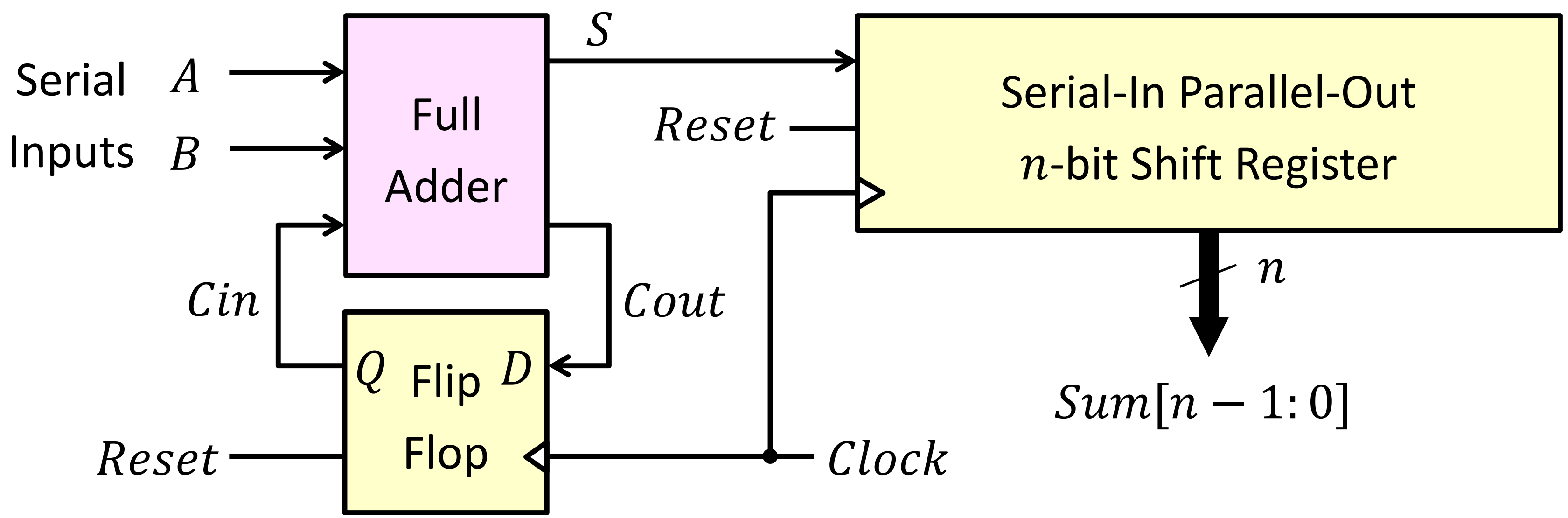


Bit Serial Adder

- Adding two n-bit numbers A and B serially over n clock cycles
- A bit-serial adder can be implemented using

1. A Full Adder
2. A Flip-Flop to store the carry-out
3. A Shift Register to store the n-bit sum

Serial Addition Starts at the Least-significant bit



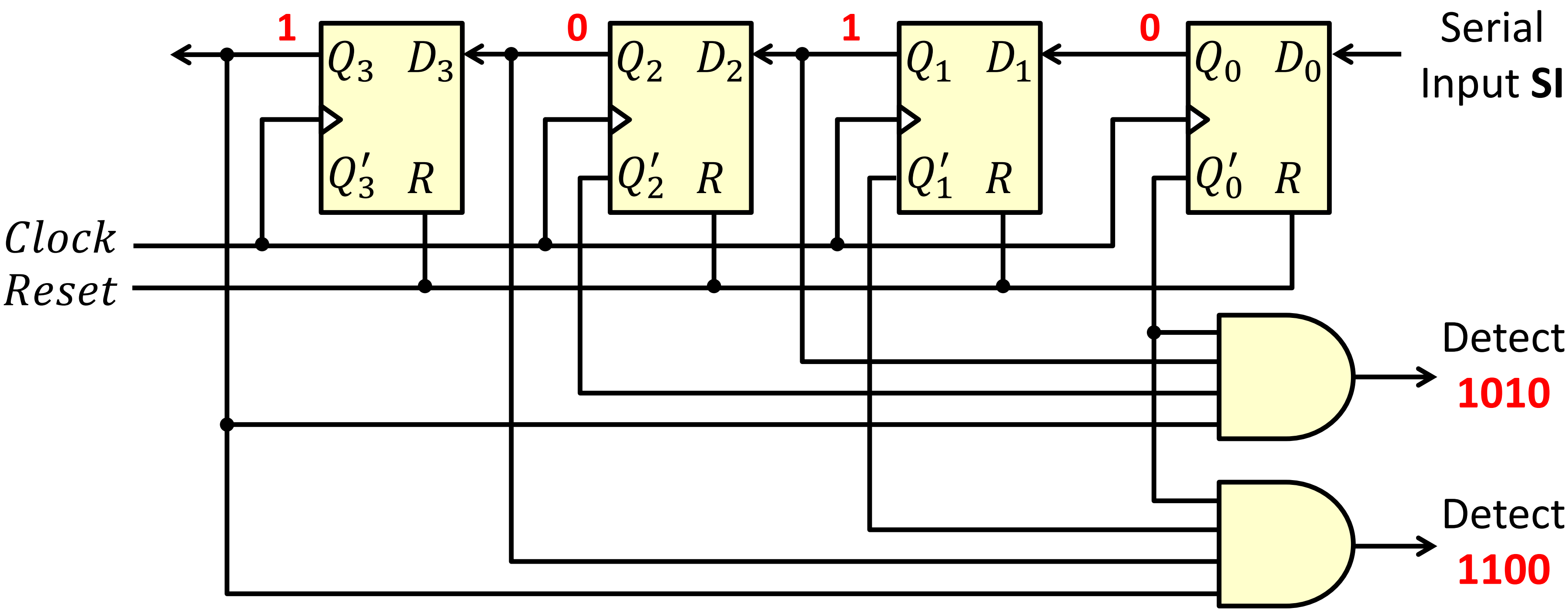
Sequence Detector with a Shift Register

- A sequence detector can be implemented using:

  Left Shift Register (SIPO) + AND Gates

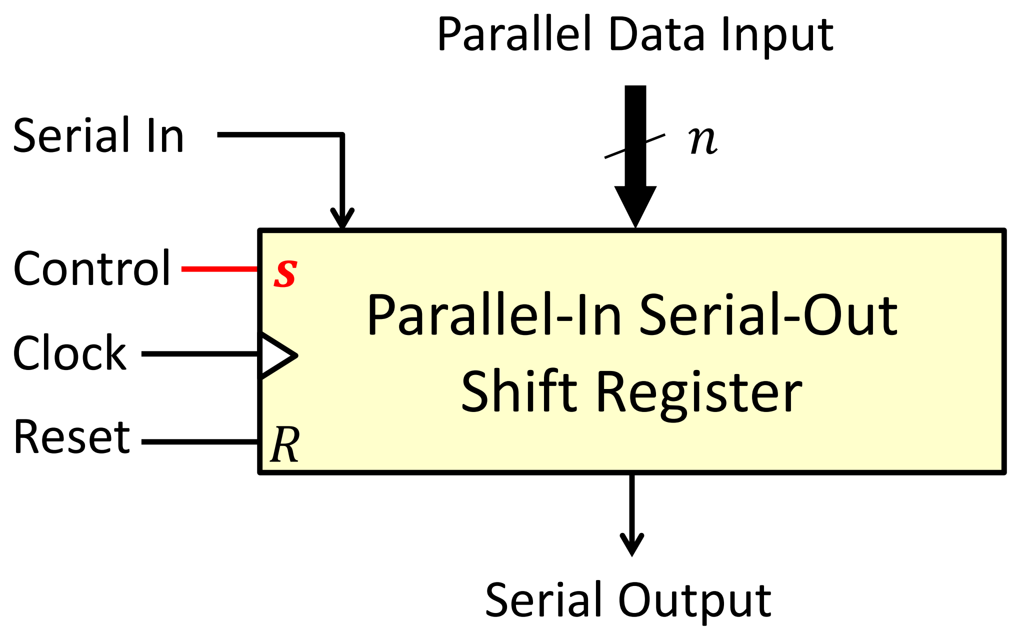
- Example: Detecting the sequences **1010** and **1100**

  Bits are shifted left starting at the most-significant bit



Parallel-In Serial-Out Shift Register

- A Parallel-In Serial-Out (PISO) Shift Register has:
  - n parallel data input lines
  - Serial Input
  - Serial Output
  - Control input ***s***
  - Clock input
  - Reset input

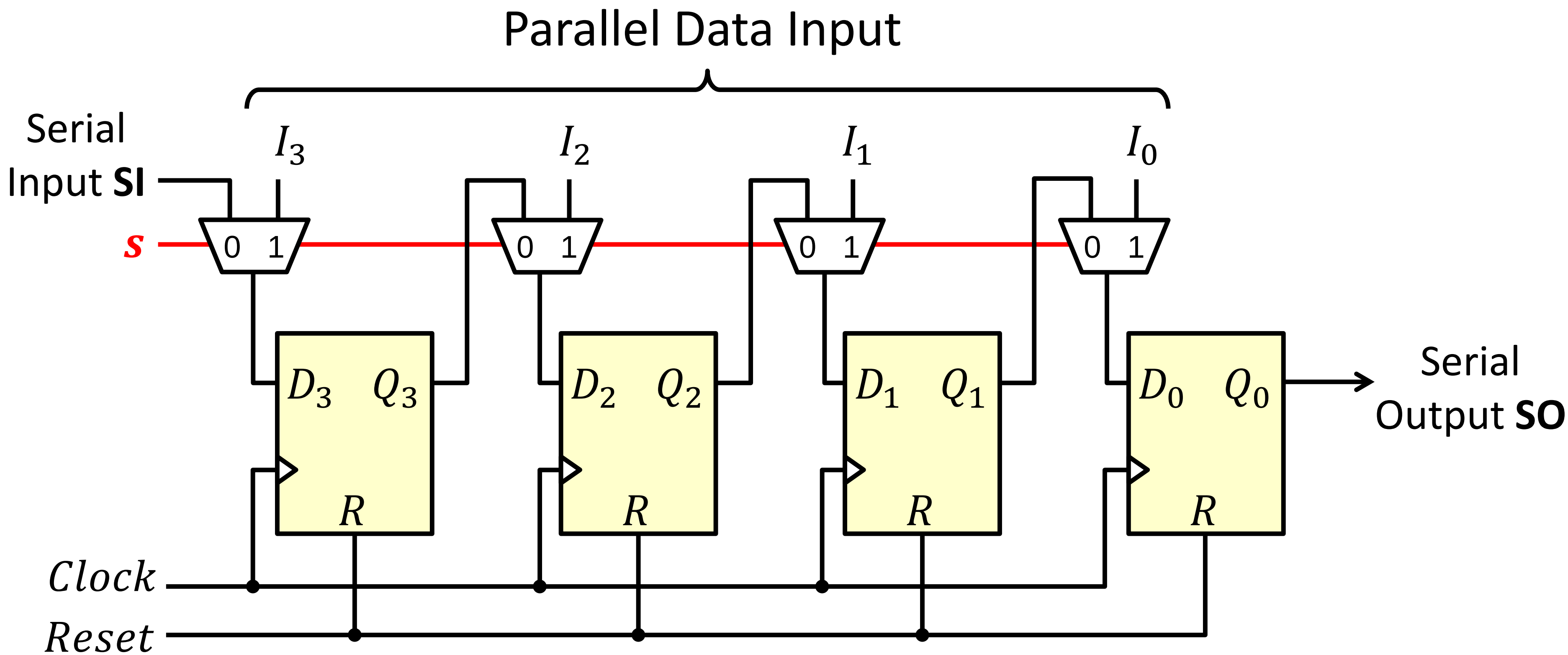


- Two control functions:
  - ***s* = 0** → Shift Data
  - ***s* = 1** → Parallel Load n input bits

Parallel In Serial Out Shift Register

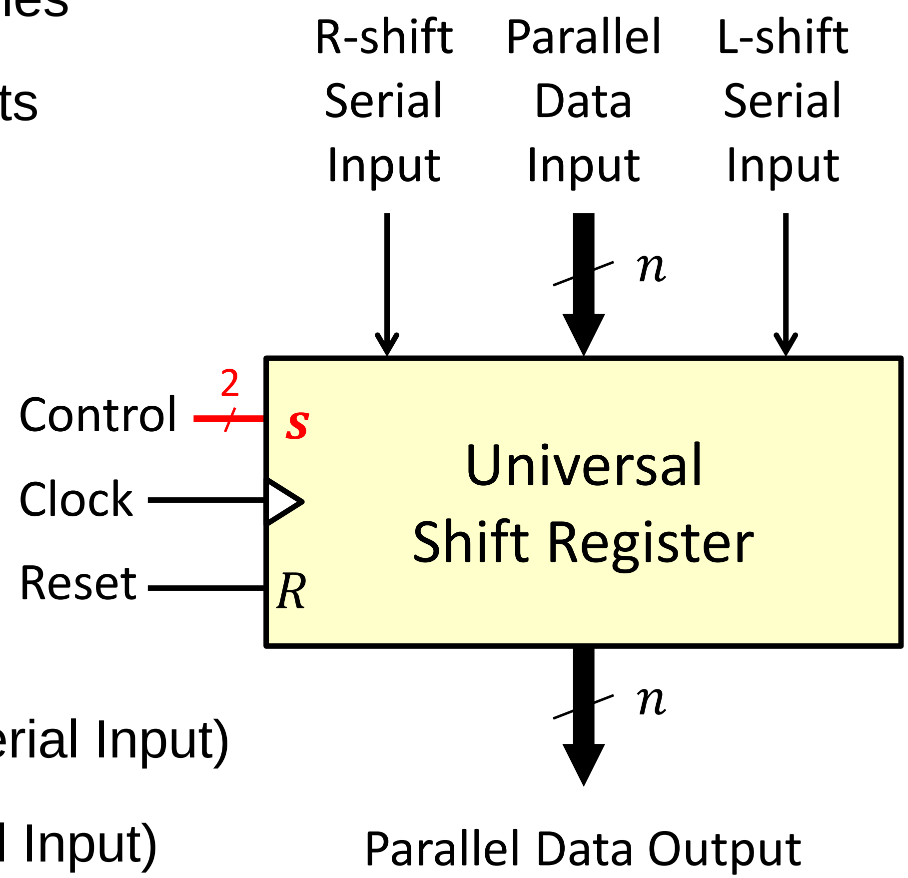
❖ Two control functions:

s = 0 ➔ Shift　　　　**s = 1** ➔ Load data

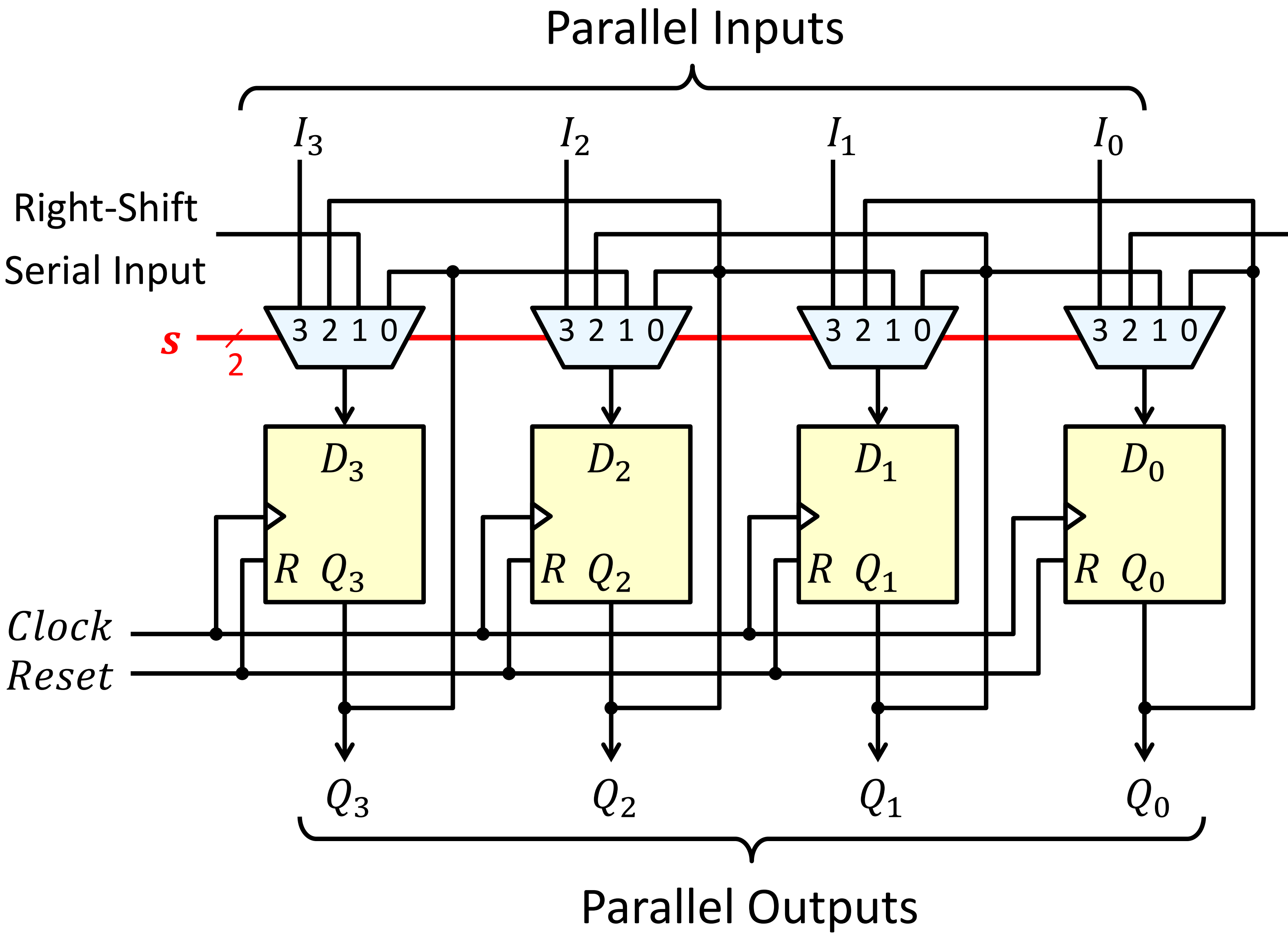


Universal Shift Register

- A Universal Shift Register has the following specification:
  - n parallel data input and n output lines
  - Right-shift and Left-shift Serial Inputs
  - Two control input lines ***s***
  - Clock input
  - Reset input
- Four control functions:
  - ***s* = 00** ➔ No change in value
  - ***s* = 01** ➔ Shift Right (Right-Shift Serial Input)
  - ***s* = 10** ➔ Shift Left (Left-Shift Serial Input)
  - ***s* = 11** ➔ Parallel Load n input bits



Universal Shift Register Design



s = 0 ➔ No change

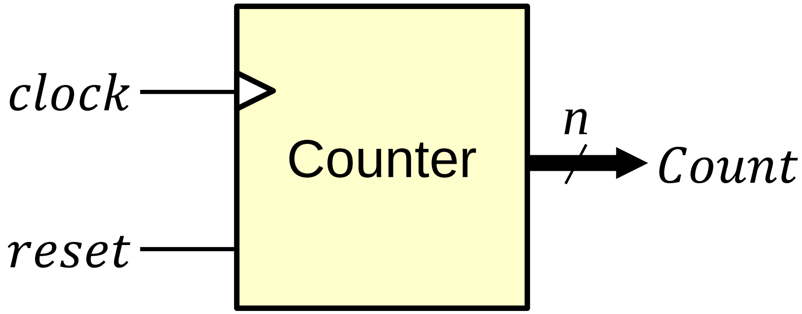
s = 1 ➔ Shift Right

s = 2 ➔ Shift Left

s = 3 ➔ Parallel Load

Counter

- Sequential circuit that goes through a specific sequence of states
- Output of the counter is the **count value**
- Modulo-*N* counter: goes through 0, 1, 2, …, (*N* – 1)
- Modulo-8 binary counter: goes through 0, 1, 2, …, 7
- Modulo-10 (BCD) counter: goes through 0, 1, 2, …, 9
- Counting can be up or down
- Some Applications:
  - Timers
  - Event Counting
  - Frequency Division



Implementing Counters

Two Basic Approaches:

1. Ripple Counters

- The system clock is connected to the clock input of the first flip-flop (LSB)
- Each flip-flop output connects to the clock input of the next flip-flop
- Advantage: simple circuit and low power consumption
- Disadvantage: The counter is not truly synchronous
- No common clock to all flip-flops
- Ripple propagation delay as the clock signal propagates to the MSB

2. Synchronous Counters

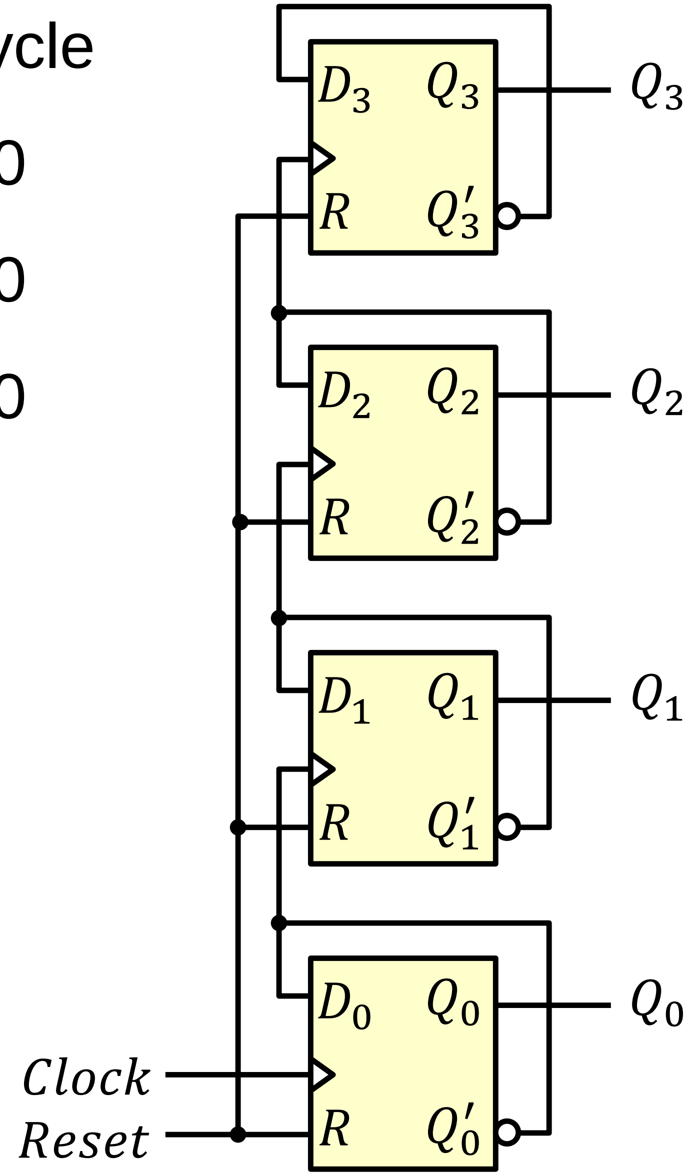
- The system clock is connected to the clock input of ALL flip-flops
- Combinational logic is used to implement the desired state sequence

Ripple Counter

- Q_0 toggles at the positive edge of every cycle
- Q_1 toggles when Q_0 goes from 1 down to 0
- Q_2 toggles when Q_1 goes from 1 down to 0
- Q_3 toggles when Q_2 goes from 1 down to 0

Q3	Q2	Q1	Q0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0

Counts Up
from 0 to 15
then back to 0



Ripple Counter (cont'd)

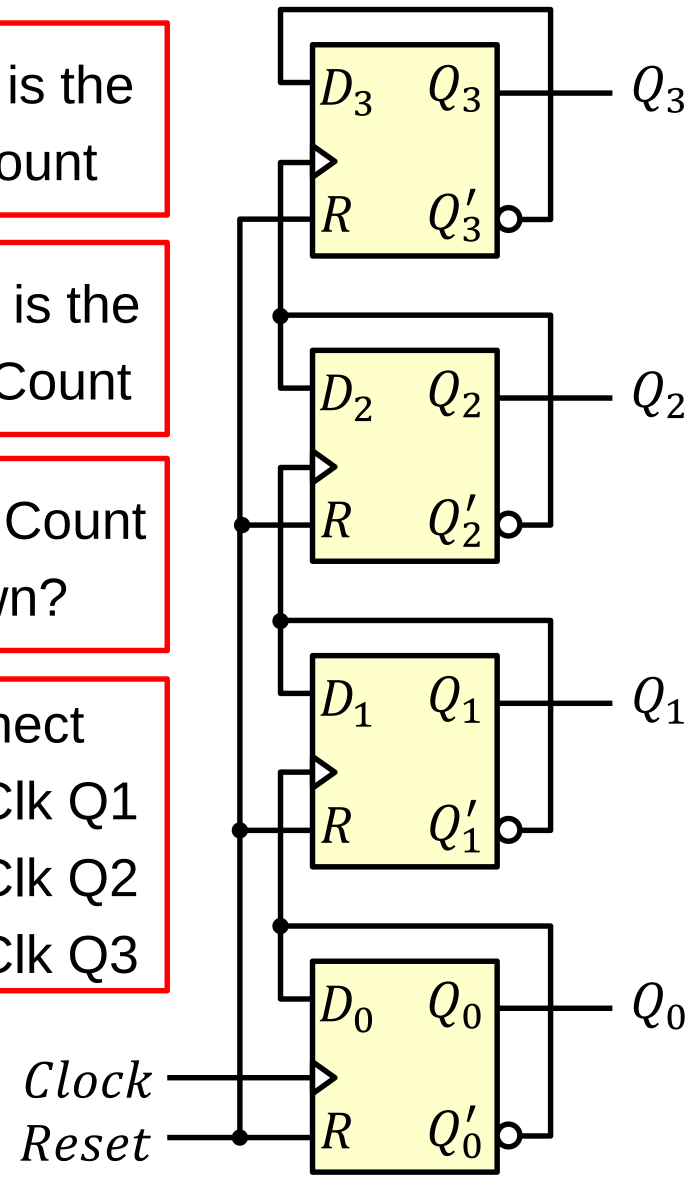
Up Count				Down Count			
Q_3	Q_2	Q_1	Q_0	Q'_3	Q'_2	Q'_1	Q'_0
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	0
0	0	1	0	1	1	0	1
0	0	1	1	1	1	0	0
0	1	0	0	1	0	1	1
0	1	0	1	1	0	1	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	0	0
1	0	0	0	0	1	1	1
1	0	0	1	0	1	1	0
1	0	1	0	0	1	0	1
1	0	1	1	0	1	0	0
1	1	0	0	0	0	1	1
1	1	0	1	0	0	1	0
1	1	1	0	0	0	0	1
1	1	1	1	0	0	0	0

$Q[3:0]$ is the Up Count

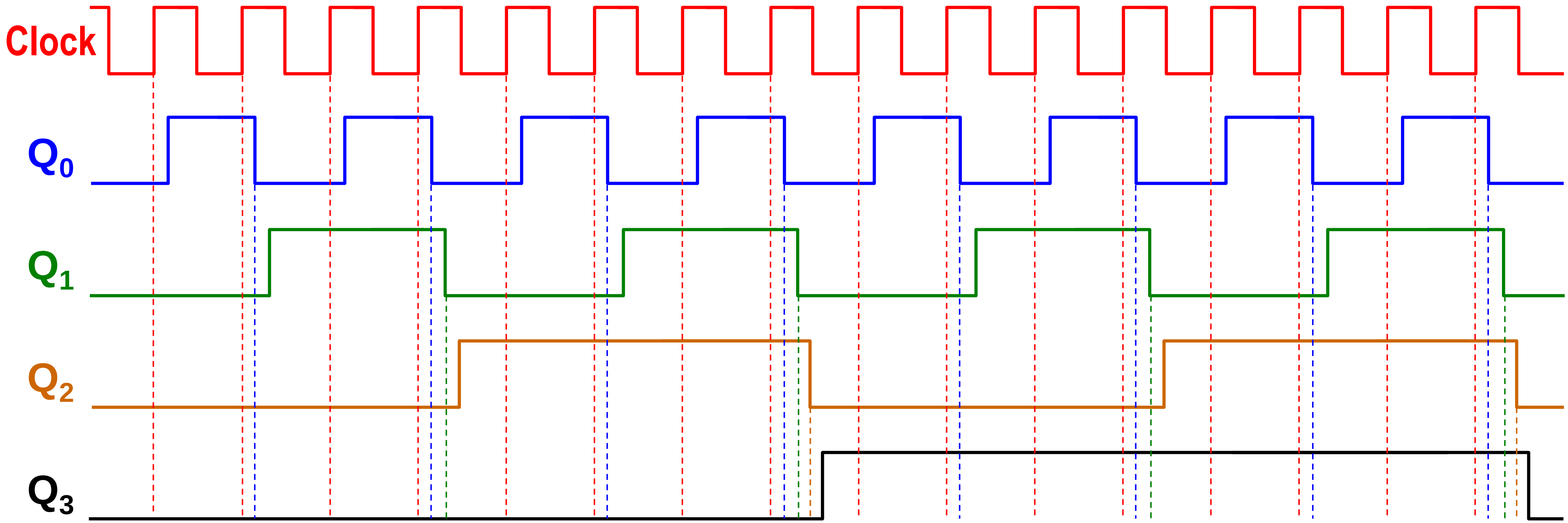
$Q'[3:0]$ is the Down Count

How to Count Down?

Connect
Q0 to Clk Q1
Q1 to Clk Q2
Q2 to Clk Q3



Timing of a Ripple Counter



- **Drawback of ripple counter:**

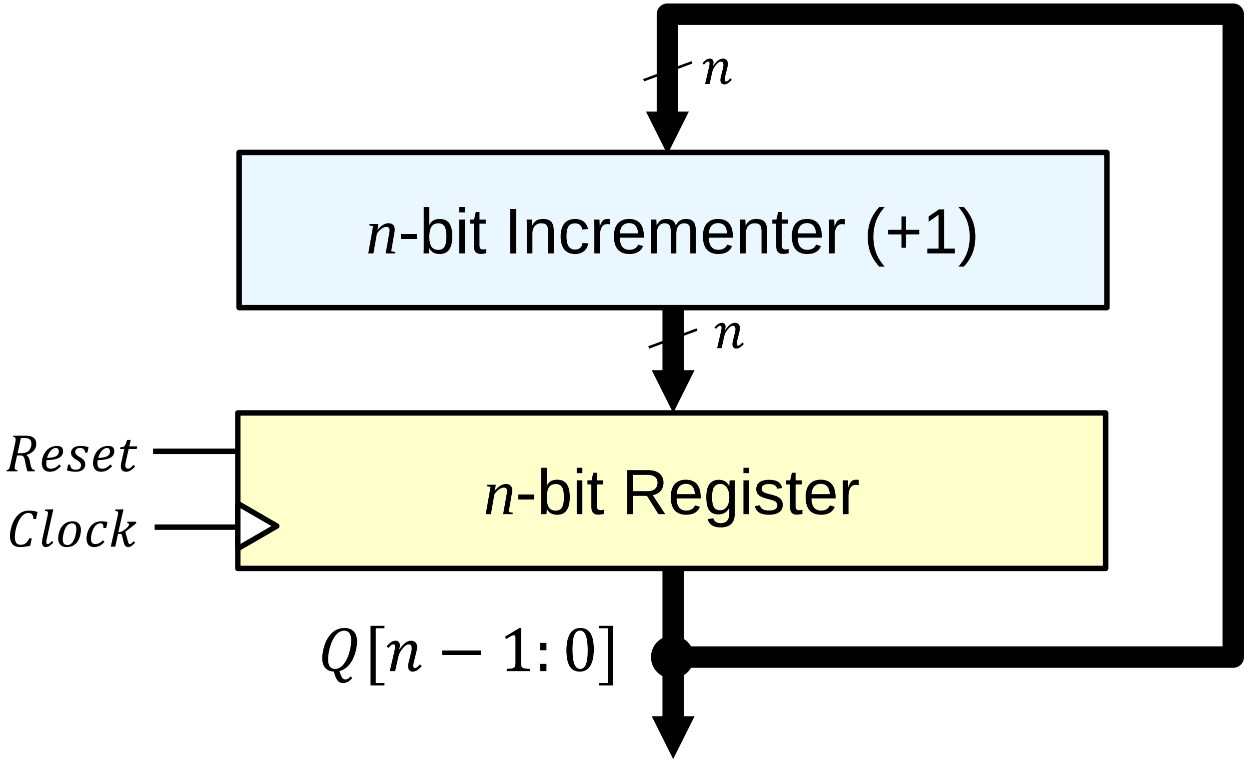
  Flip-flops are NOT driven by the same clock (Not Synchronous)

  Q delay increases as we go from Q_0 to Q_3

  Given Δ = flip-flop delay ➔ Delay of Q_0, Q_1, Q_2, Q_3 = Δ, 2Δ, 3Δ, 4Δ

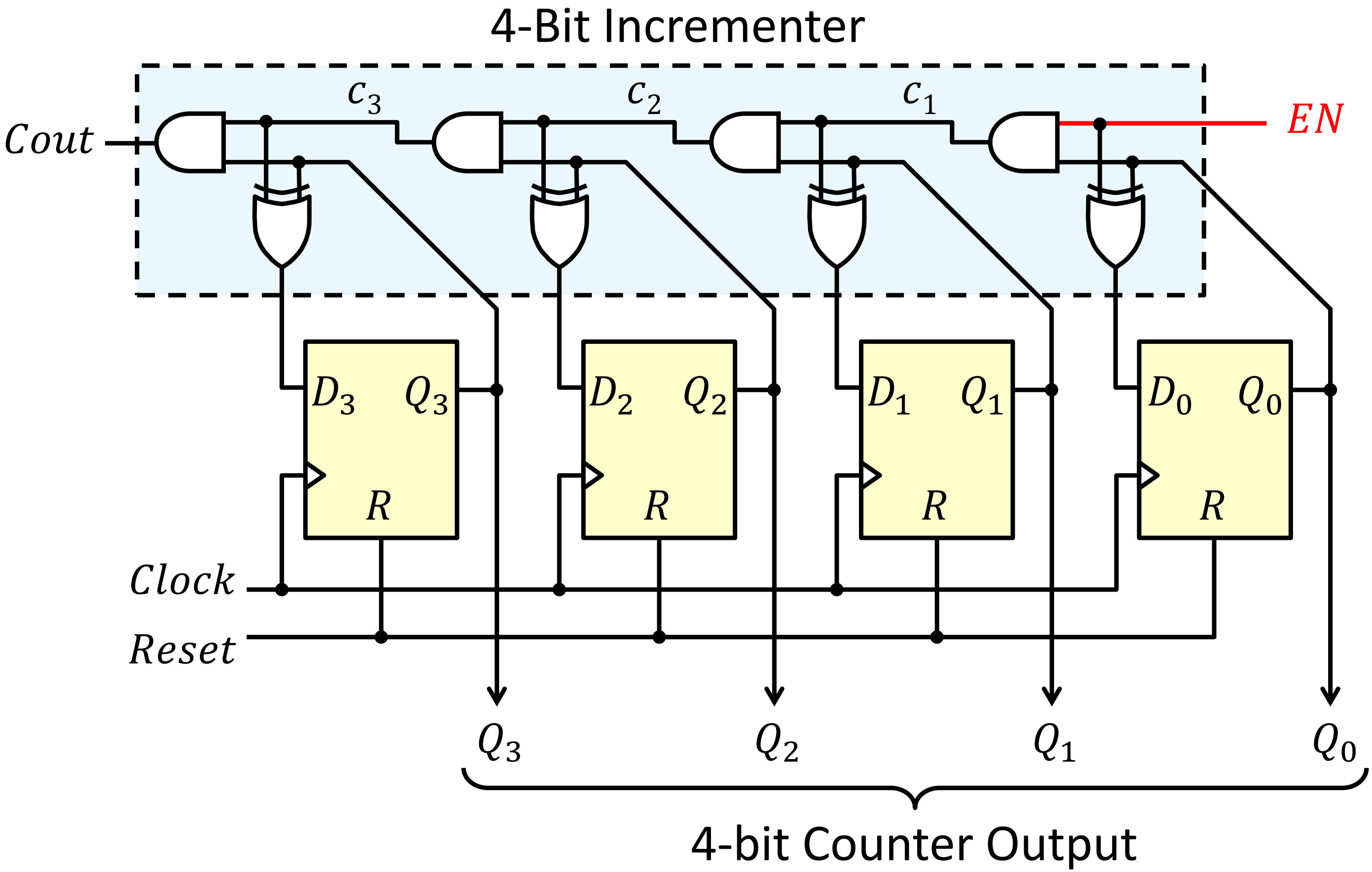
Synchronous Counter

- Avoid clock rippling
- n-bit Register with a **common clock** for all flip-flops
- n-bit Incrementer to generate next state (Up-Counter)



4-Bit Synchronous Counter with Enable

❖ An incrementer is a reduced (contracted) form of an adder



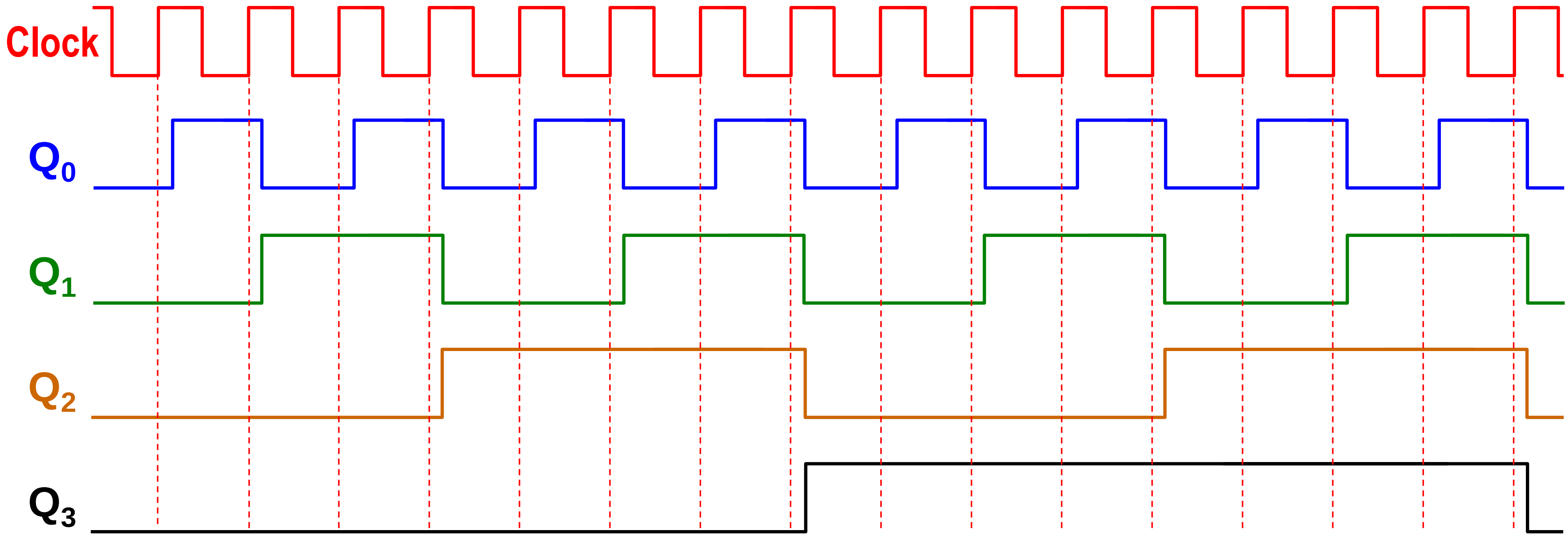
EN = 0 ➔ No Change

EN = 1 ➔ Increment

How to Count Down?

Increment **Q'** (Not Q)

Timing of a Synchronous Counter



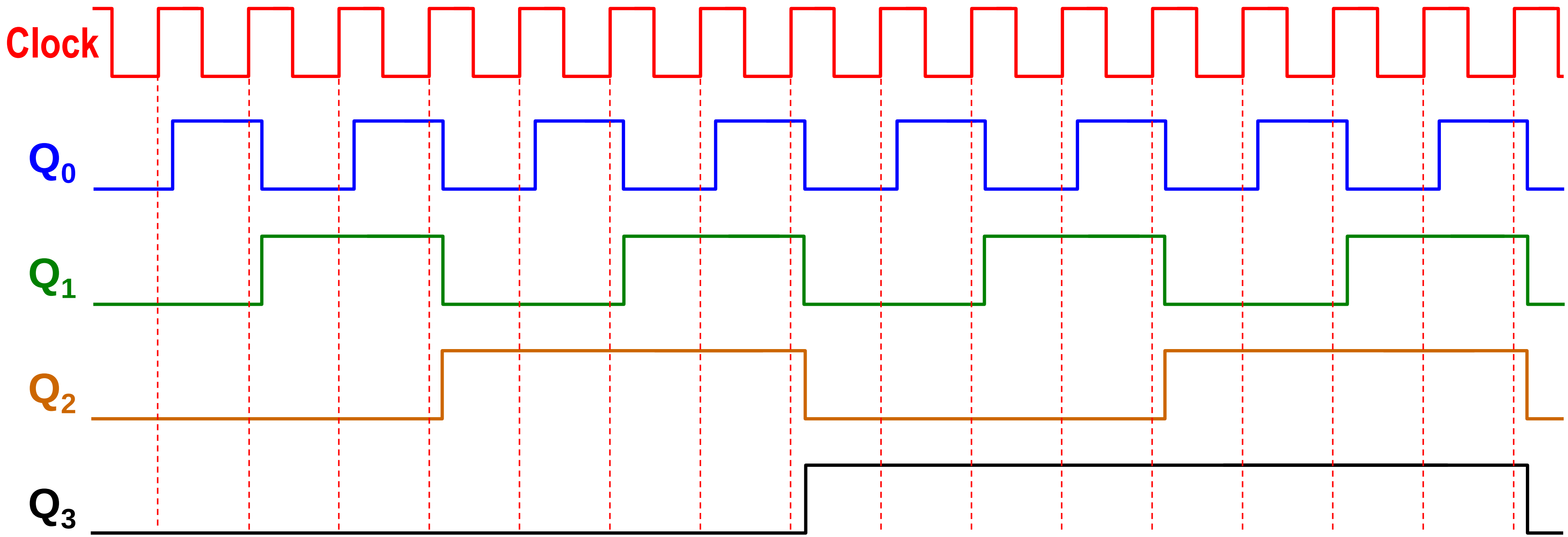
❖ **Advantage of Synchronous counter:**

ALL Flip-flops are driven by the **same clock**

Delay of all outputs is identical ➔ Delay of $Q_0 = Q_1 = Q_2 = Q_3 = \Delta$

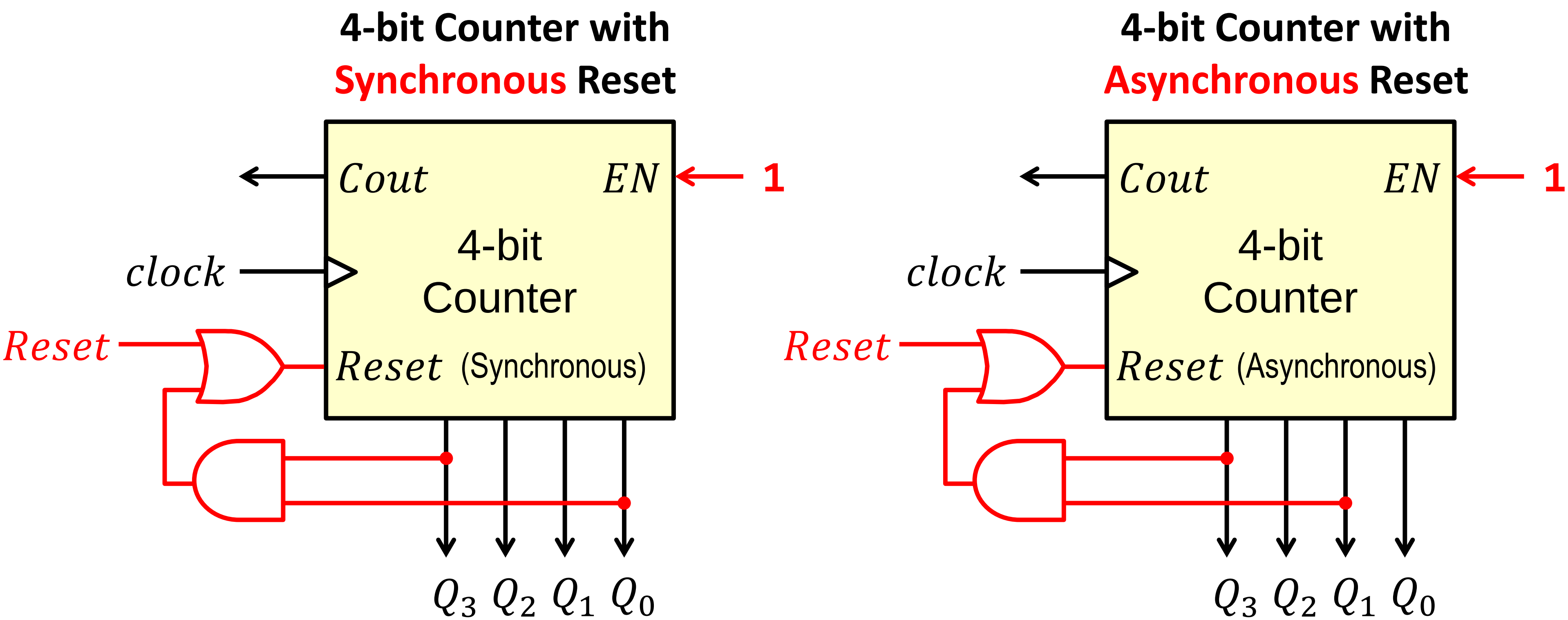
Frequency Division

- A counter can be used as a frequency divider
- Counter is driven by a **Clock** with **frequency F**
- Output Q_0 Frequency = **F/2**, Output Q_1 Frequency = **F/4**
- Output Q_2 Frequency = **F/8**, Output Q_3 Frequency = **F/16**



BCD Counter

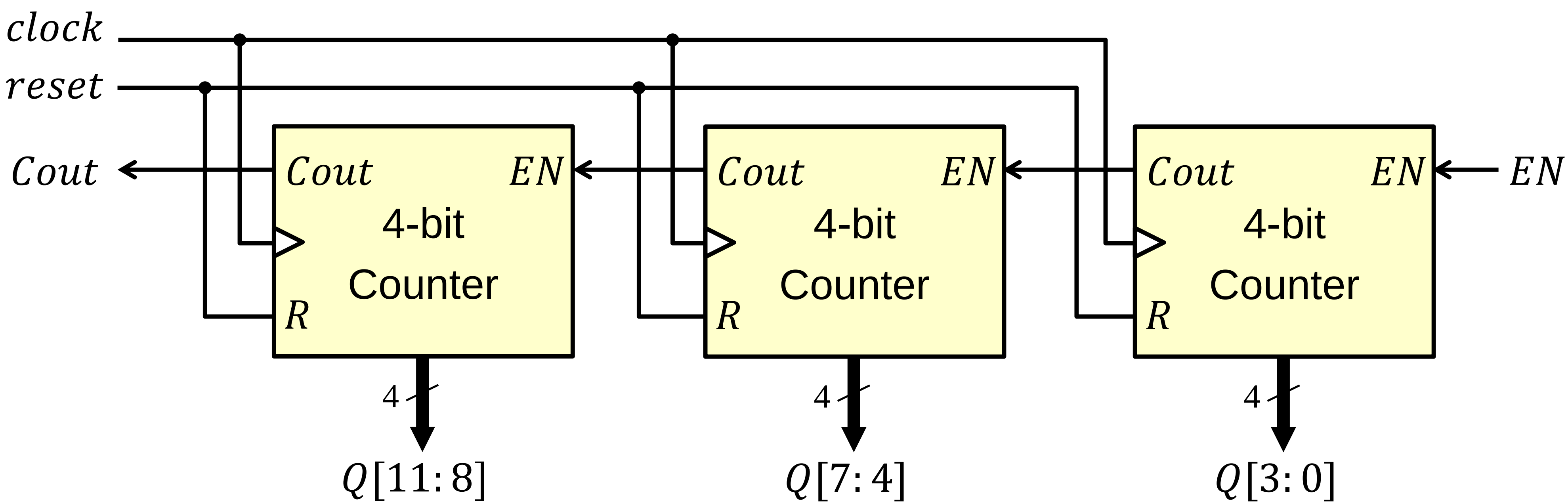
- **Problem:** Convert a 4-bit binary counter into a BCD counter
- **Solution:** When output reaches **9** then reset back to **0**
- **Asynchronous Reset:** Count to **10** and reset immediately



Building Larger Synchronous Counters

- Smaller counters can be used to build a larger counter
- Example: 12-bit counter designed using three 4-bit counters

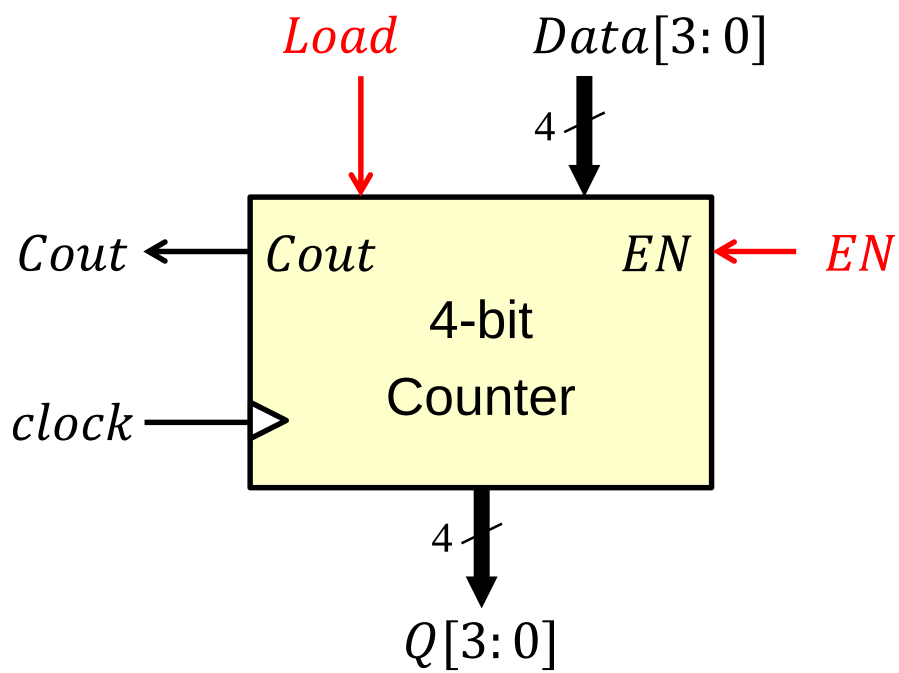
  Counts from **0** to **4095** ($2^{12} - 1$), then back to **0**
- The $Cout$ of a 4-bit counter is used to enable the next counter



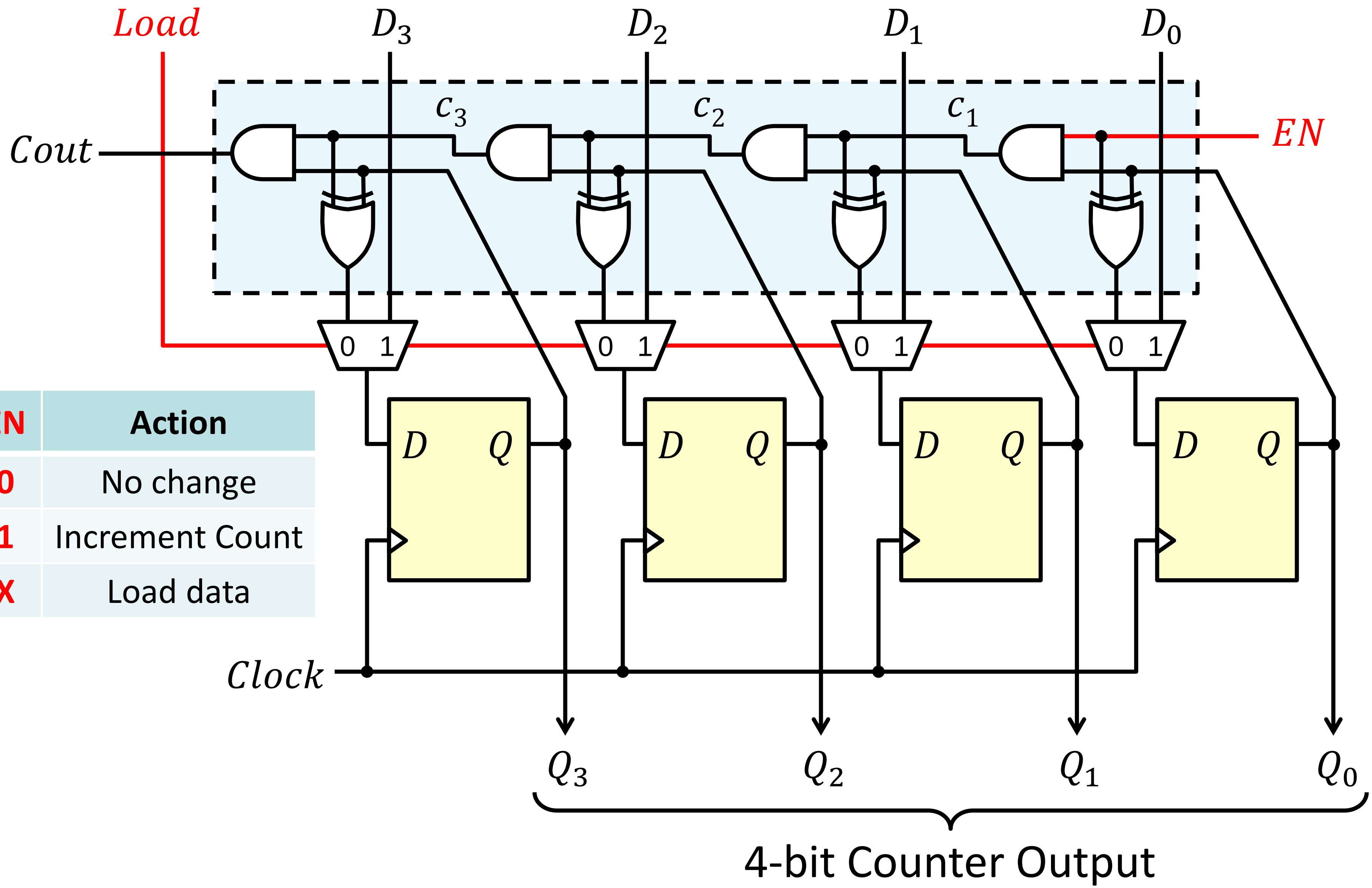
Synchronous Counter with Parallel Load

- Ability to load an initial binary number into the counter
  - Prior to the count operation
- Two control inputs:
  - **Load:** Initialize counter with input Data
  - **EN:** enables the counting

Very useful in implementing different counting sequences



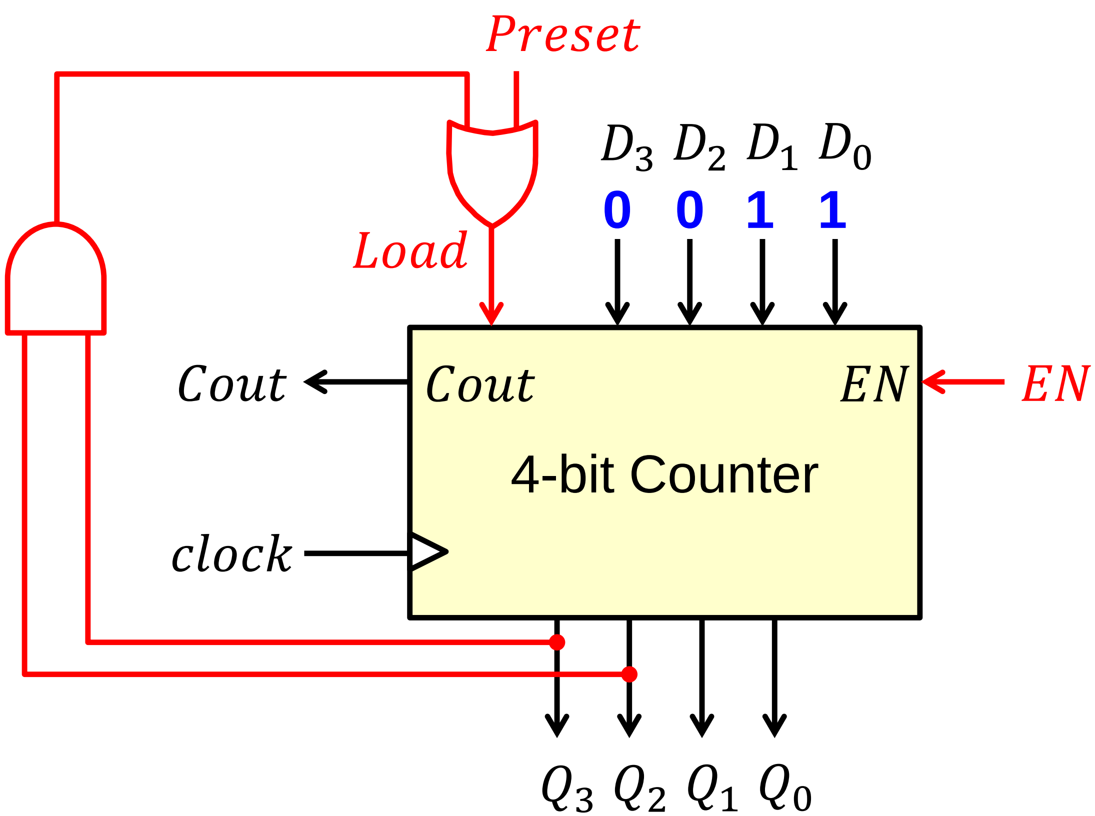
Implementing a Counter with Parallel Load



Load	EN	Action
0	0	No change
0	1	Increment Count
1	X	Load data

3-to-12 Counter

- Convert a 4-bit binary counter with load into 3-to-12 counter
- **Solution:** Detect binary count **12** and then load **3**
- **Detect 12:** Binary count with $Q_3 = Q_2 = 1$



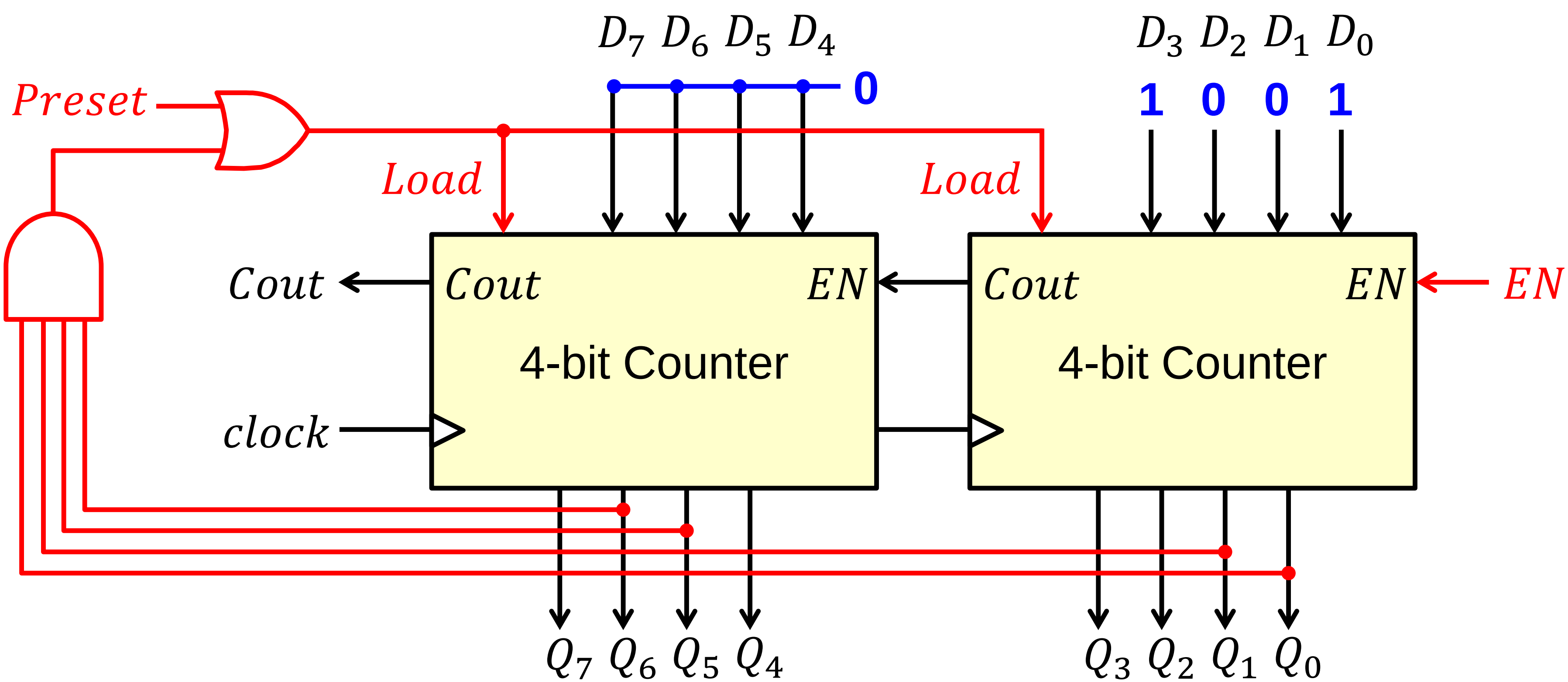
Synchronous Preset

Load: 0011

9-to-99 Counter

Problem: Use two 4-bit binary counters with parallel load and logic gates to build a counter that counts from **9** to **99 = 'b01100011**

Add a synchronous **Preset** input to initialize the counter to value **9**



Memory

- Large array or storage cells, capable of storing many 0's and 1's
- Random Access Memory: bits can be accessed randomly
- Memory is addressable

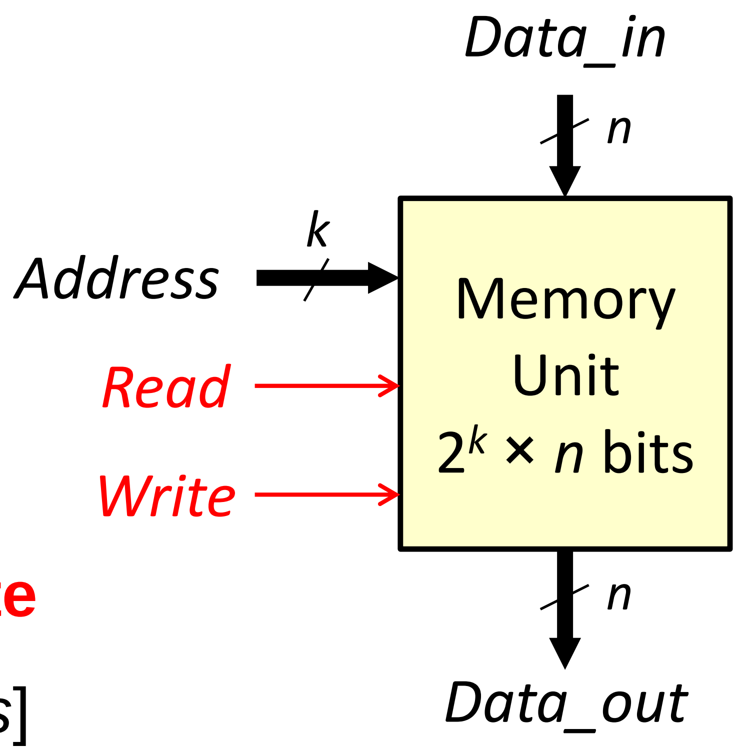
  Memory address consists of *k* bits

  Can address 2^k words in memory

  Each word consists of *n* bits
- Memory capacity = $2^k \times n$ bits
- Two control functions: **Read** and **Write**

  Read: *Data_out* ← **Memory** [*Address*]

  Write: **Memory** [*Address*] ← *Data_in*



RAM, ROM, EEPROM, and Flash

- **RAM:** Random Access Memory

  Can be read and written using Read/Write operations

  Volatile: data is lost when power is turned off

- **ROM:** Read Only Memory (No Write operation)

  Mask programming by the circuit manufacturer (not by the user)

  Non-Volatile Memory (NVM): data is permanent

- **EEPROM:** Electrically Erasable Programmable ROM

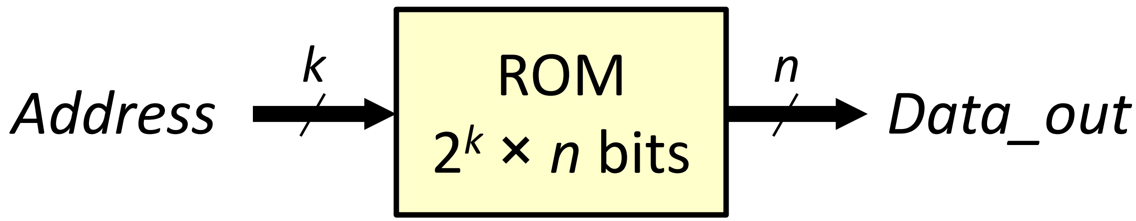
  Can be erased and reprogrammed by the user (special write)

  EEPROM Programmer: Device that writes the EEPROM

- **Flash:** Non-Volatile Memory that can be read and written

ROM Memory

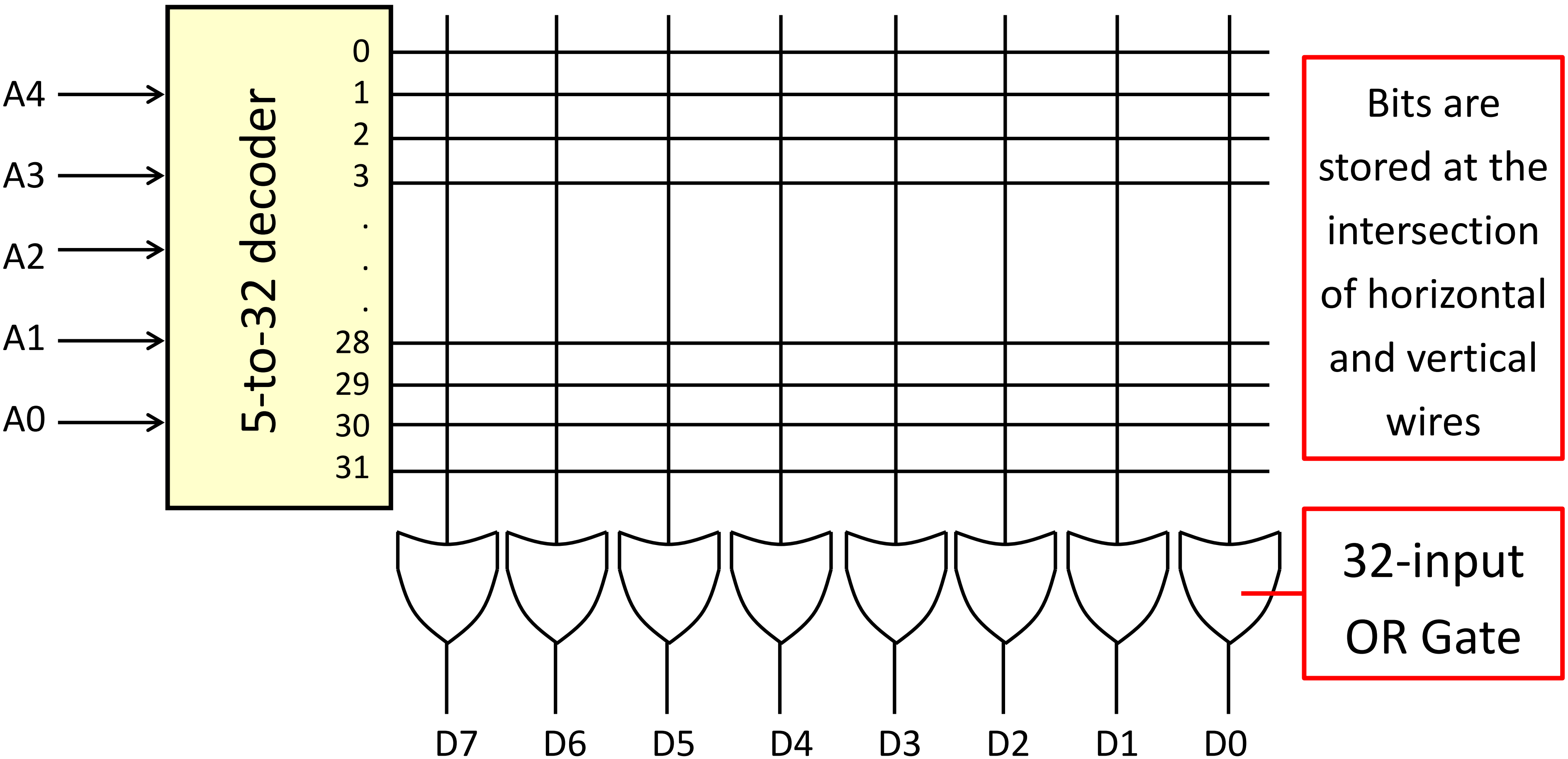
- Address consists of k bits → 2^k memory addresses
- At each memory address, there is a word consisting of n bits
- The n-bit word appears at the data output of the ROM
- ROM does not have data inputs or a write operation



- ROM memory is useful for implementing Boolean Functions
- Also useful for storing permanent data

ROM Internal Structure (32 x 8-bit)

- 5-bit Address ➔ 5-to-32 line decoder (Only one line is selected)
- Each line = 8 bits ➔ 8-bit Data output



Implementing a Combinational Circuit

- Implementing a Combinational Circuit with a ROM is easy
- Store the truth table of the circuit by programming the ROM

Truth Table with Five Inputs and Eight output functions

I4	I3	I2	I1	I0	F7	F6	F5	F4	F3	F2	F1	F0
0	0	0	0	0	0	0	1	1	1	0	0	0
0	0	0	0	1	1	0	0	0	0	1	1	0
0	0	0	1	0	0	1	0	1	1	0	0	1
0	0	0	1	1	1	0	1	1	0	0	1	0
	.	.	.				.	.	.			
1	1	1	1	1	0	0	1	0	0	0	1	0

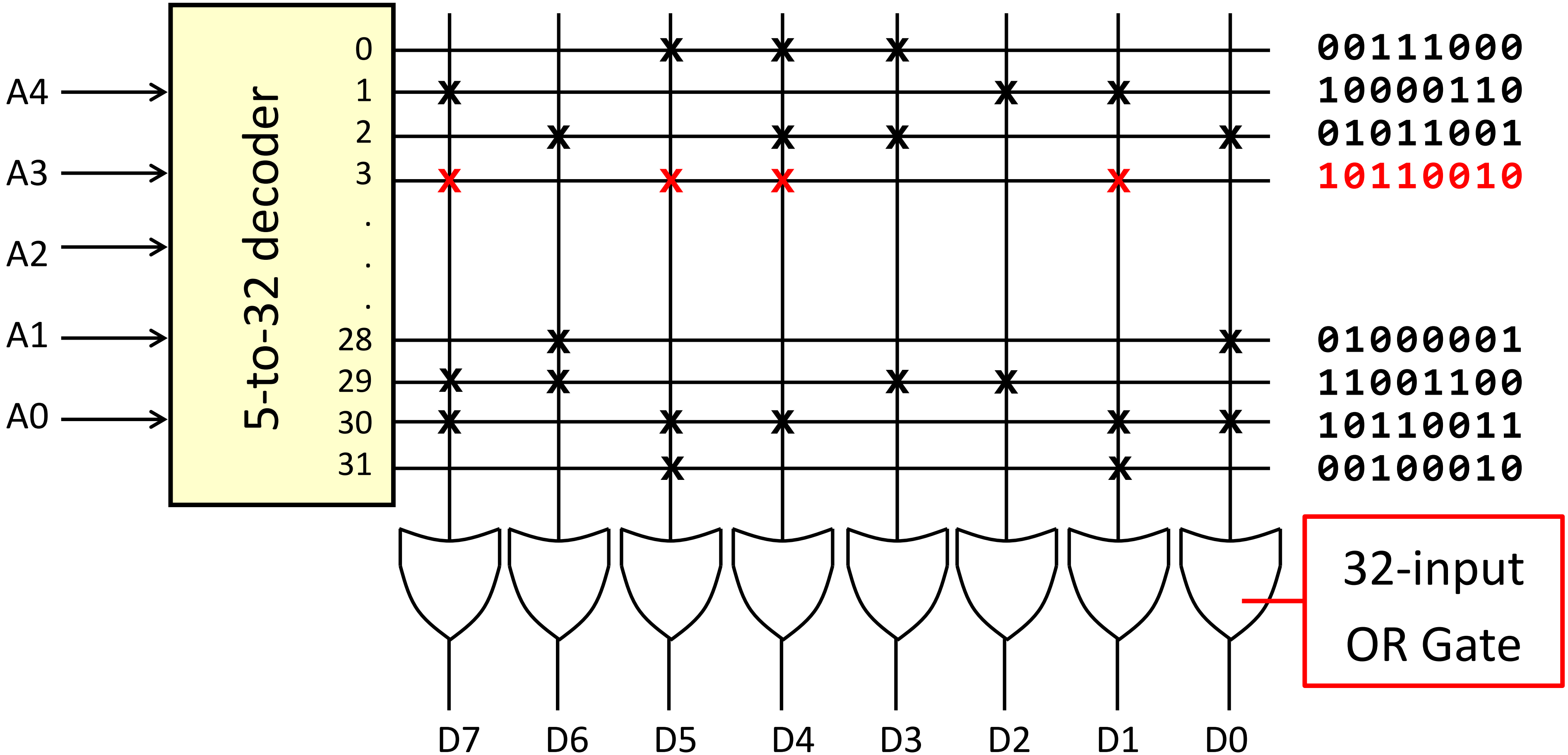
Inputs are used as Address lines to the ROM

Programming a ROM

Every **1** in the truth table ➔ **X** (CLOSED) connection

Every **0** in the truth table ➔ NO connection

Example: At address **00011** = (decimal **3**), the word **10110010** is stored



Example: Square Function

- Design a square function with a ROM
- Input **X** = 3-bit number, Output **Y = X²**
- **Solution:** Derive the Truth Table

X_2 X_1 X_0	Square	Y_5 Y_4 Y_3 Y_2 Y_1 Y_0
0 0 0	0	0 0 0 0 0 0
0 0 1	1	0 0 0 0 0 1
0 1 0	4	0 0 0 1 0 0
0 1 1	9	0 0 1 0 0 1
1 0 0	16	0 1 0 0 0 0
1 0 1	25	0 1 1 0 0 1
1 1 0	36	1 0 0 1 0 0
1 1 1	49	1 1 0 0 0 1

ROM Table

- Output Y_0 is identical to input X_0 ➔ No need to store in ROM
- Similarly, Output Y_1 is always **0** ➔ No need to store in ROM
- ROM table ➔ Only need to store Y_5, Y_4, Y_3, and Y_2 in ROM

X_2	X_1	X_0	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1
0	1	0	0	0	0	1	0	0
0	1	1	0	0	1	0	0	1
1	0	0	0	1	0	0	0	0
1	0	1	0	1	1	0	0	1
1	1	0	1	0	0	1	0	0
1	1	1	1	1	0	0	0	1

Minimal ROM

Size = $2^3 \times 4$ bits