Digital Components

uOttawa

L'Université canadienne
Canada's university

Dr. Voicu Groza

SITE Hall, Room 5017
562 5800 ext. 2159
VGroza@uOttawa.ca

Université d'Ottawa | University of Ottawa

www.uOttawa.ca

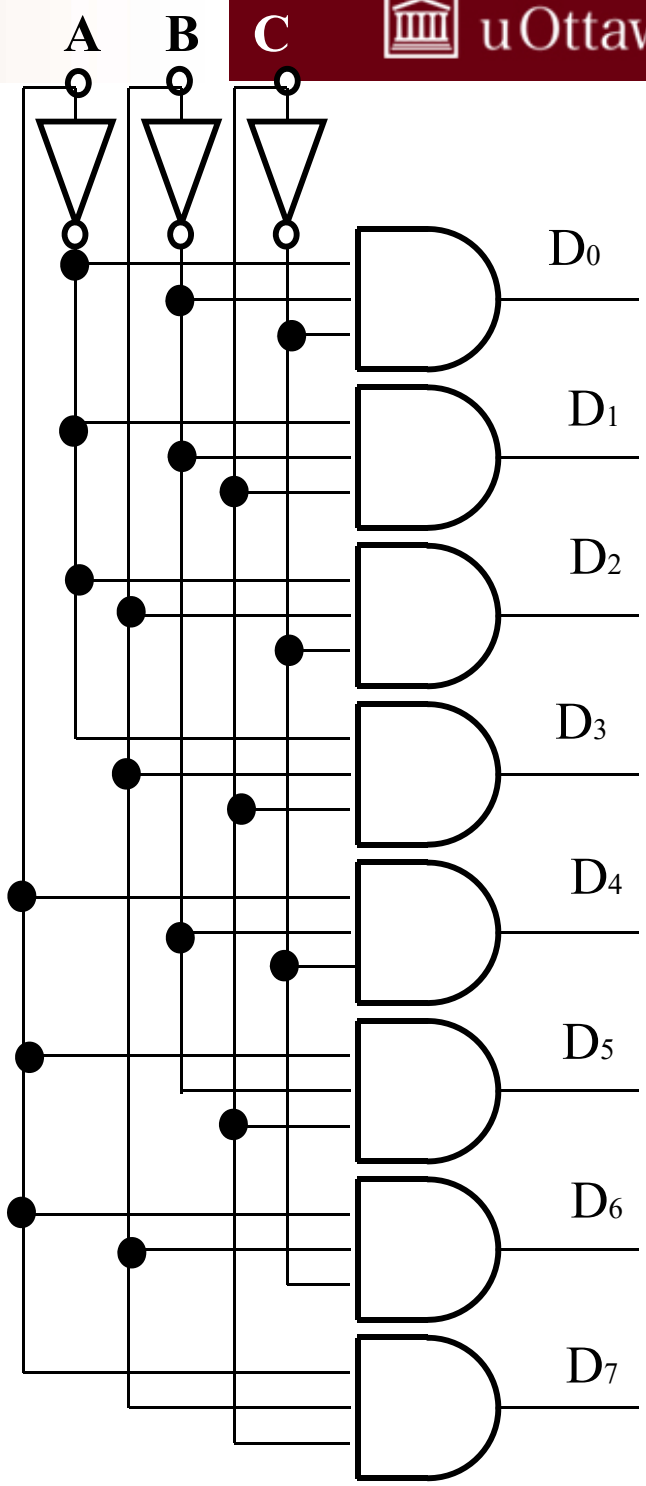
Outline

- **Decoders, Multiplexers**
- Registers + Common sense design strategies
  - Register with Parallel Load
  - Shift Registers
  - Bidirectional Shift Register with Parallel Load
- Binary Counters
  - Binary Counter with Parallel Load
- Multi-function Registers

3-to-8 Line Decoder

- A decoder is a combinational circuit that converts binary information from an n-bit input to a maximum of 2^n-bit output.
- An n-input m-output decoder is called an n-to-m line decoder, where $m \leq 2^n$.

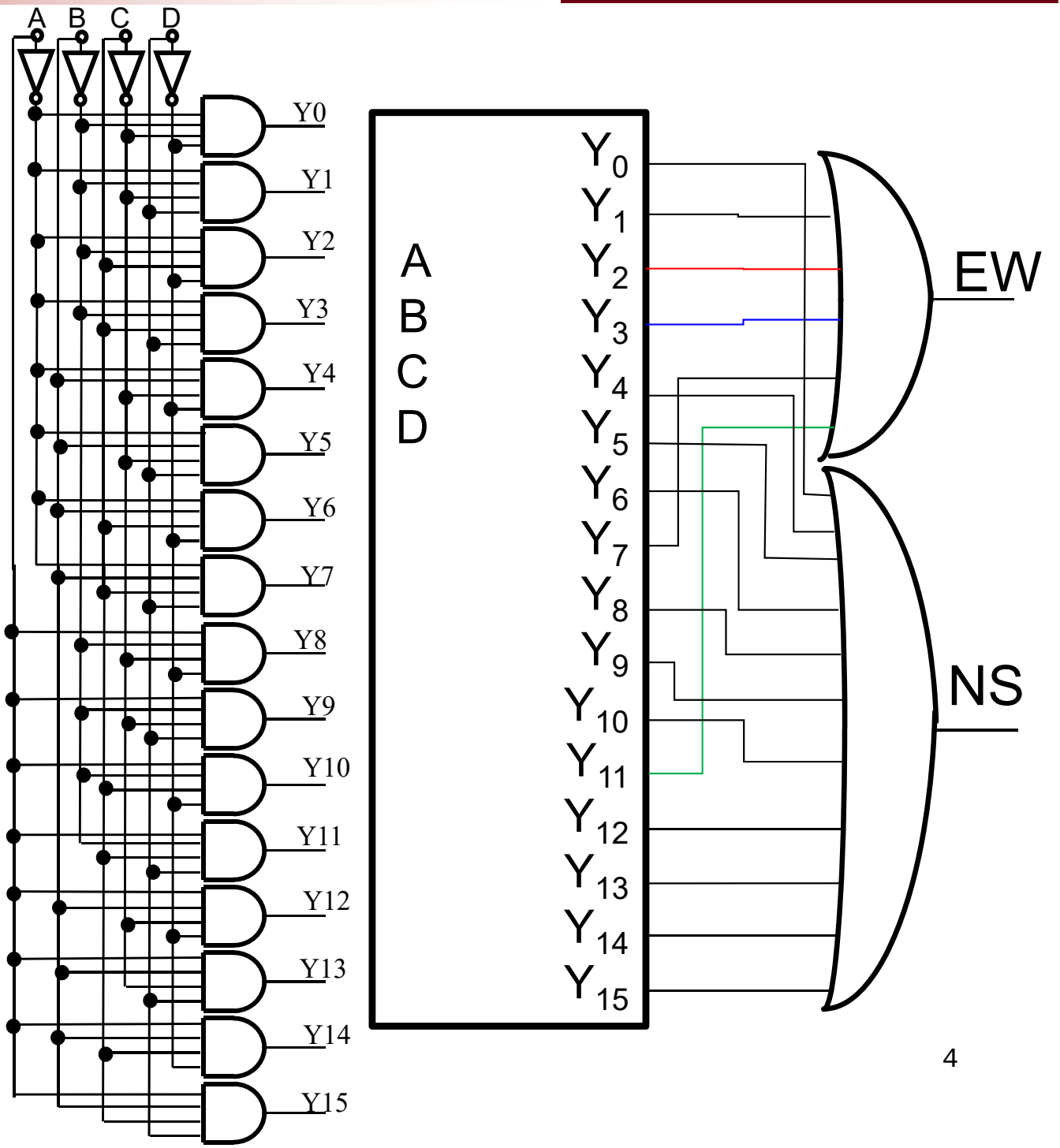
	A	B	C	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
(0)	0	0	0	1	0	0	0	0	0	0	0
(1)	0	0	1	0	1	0	0	0	0	0	0
(2)	0	1	0	0	0	1	0	0	0	0	0
(3)	0	1	1	0	0	0	1	0	0	0	0
(4)	1	0	0	0	0	0	0	1	0	0	0
(5)	1	0	1	0	0	0	0	0	1	0	0
(6)	1	1	0	0	0	0	0	0	0	1	0
(7)	1	1	1	0	0	0	0	0	0	0	1



Implement the following functions using a decoder and OR gates:

$$EW = \Sigma\,(1, 2, 3, 7, 11)$$

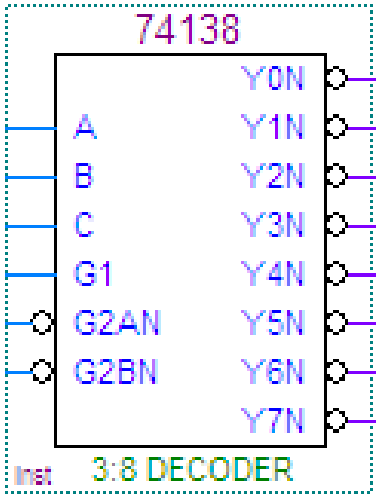
$$NS = \Sigma\,(0, 4, 5, 6, 9, 10, 12, 13, 14, 15)$$



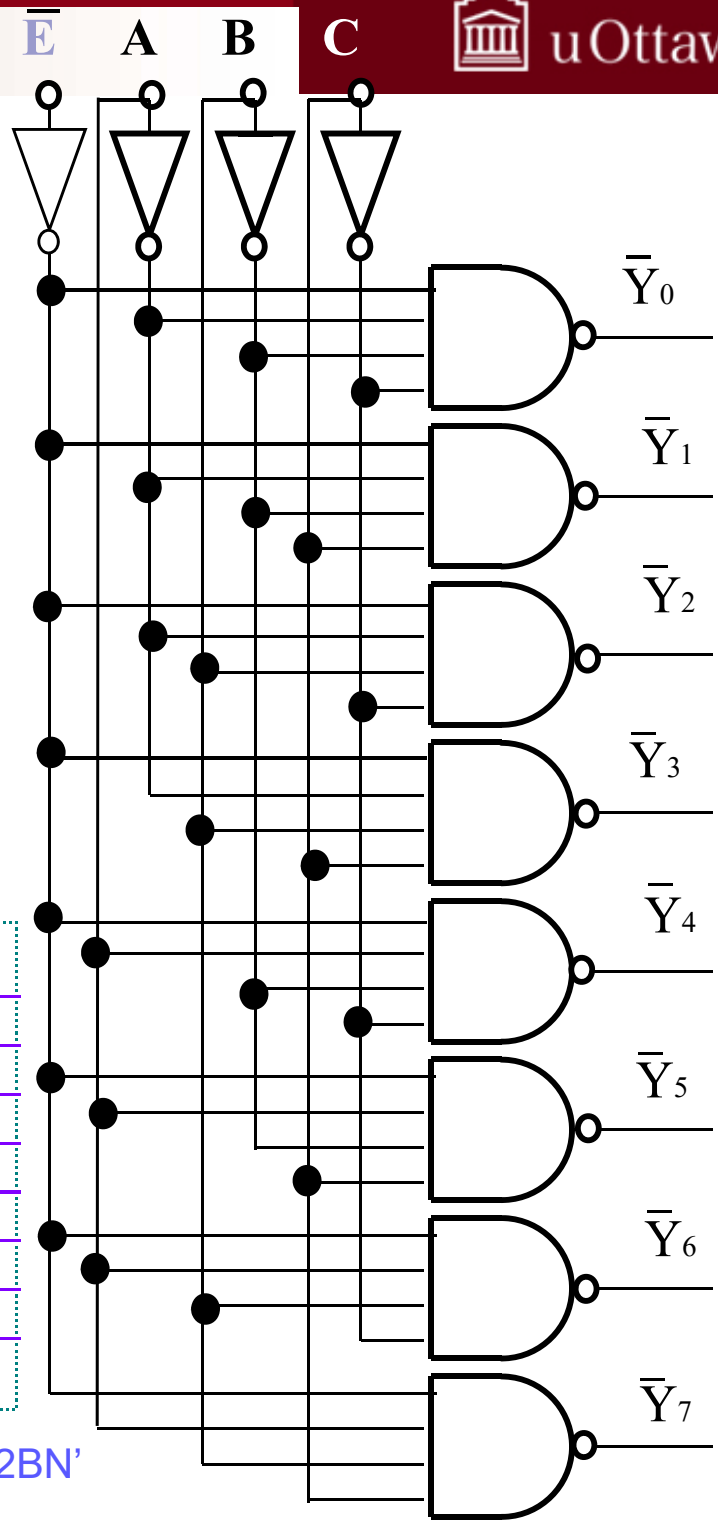
3-to-8 Decoder (74 138)

- The Enable bit, E, enables (when E = 1, i.e. $\overline{E}$=0) or disables (when E = 0, i.e. $\overline{E}$=1) the functionality of the decoder.
- When the decoder is disabled (E = 0), all the output pins of the decoder are reset to 0:

	A	B	C	$\overline{E}$	$\overline{Y}_0$	$\overline{Y}_1$	$\overline{Y}_2$	$\overline{Y}_3$	$\overline{Y}_4$	$\overline{Y}_5$	$\overline{Y}_6$	$\overline{Y}_7$
	x	x	x	1	1	1	1	1	1	1	1	1
(0)	0	0	0	0	0	1	1	1	1	1	1	1
(1)	0	0	1	0	1	0	1	1	1	1	1	1
(2)	0	1	0	0	1	1	0	1	1	1	1	1
(3)	0	1	1	0	1	1	1	0	1	1	1	1
(4)	1	0	0	0	1	1	1	1	0	1	1	1
(5)	1	0	1	0	1	1	1	1	1	0	1	1
(6)	1	1	0	0	1	1	1	1	1	1	0	1
(7)	1	1	1	0	1	1	1	1	1	1	1	0



E=G1 • G2AN' • G2BN'



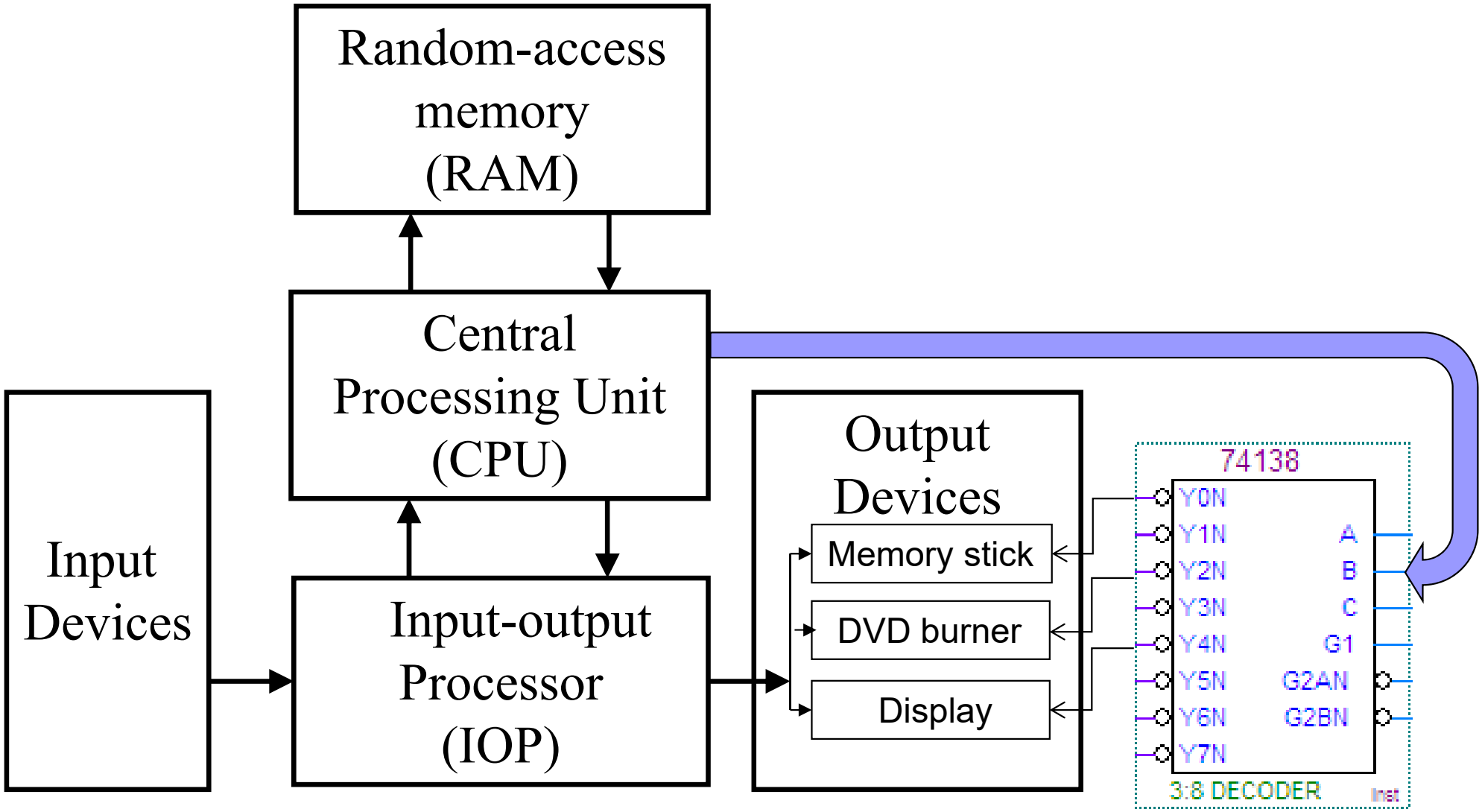
Encoders

- An **encoder** is a combinational circuit that performs the opposite operation of a decoder.
- A 2^n -bit input n-bit output encoder generates the binary code corresponding to the input value.

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

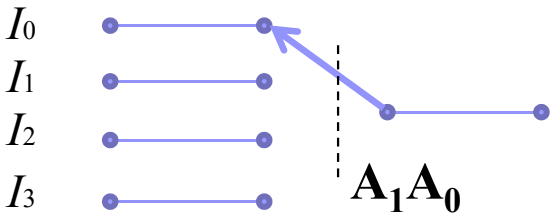
Truth Table for 8-to-3 Encoder
When all inputs are 0's but $D_3 = 1$, for instance, the output = $A_2\ A_1\ A_0 = 011$, which is the binary code for 3.

$$A_0 = D_1 + D_3 + D_5 + D_7$$
$$A_1 = D_2 + D_3 + D_6 + D_7$$
$$A_2 = D_4 + D_5 + D_6 + D_7$$

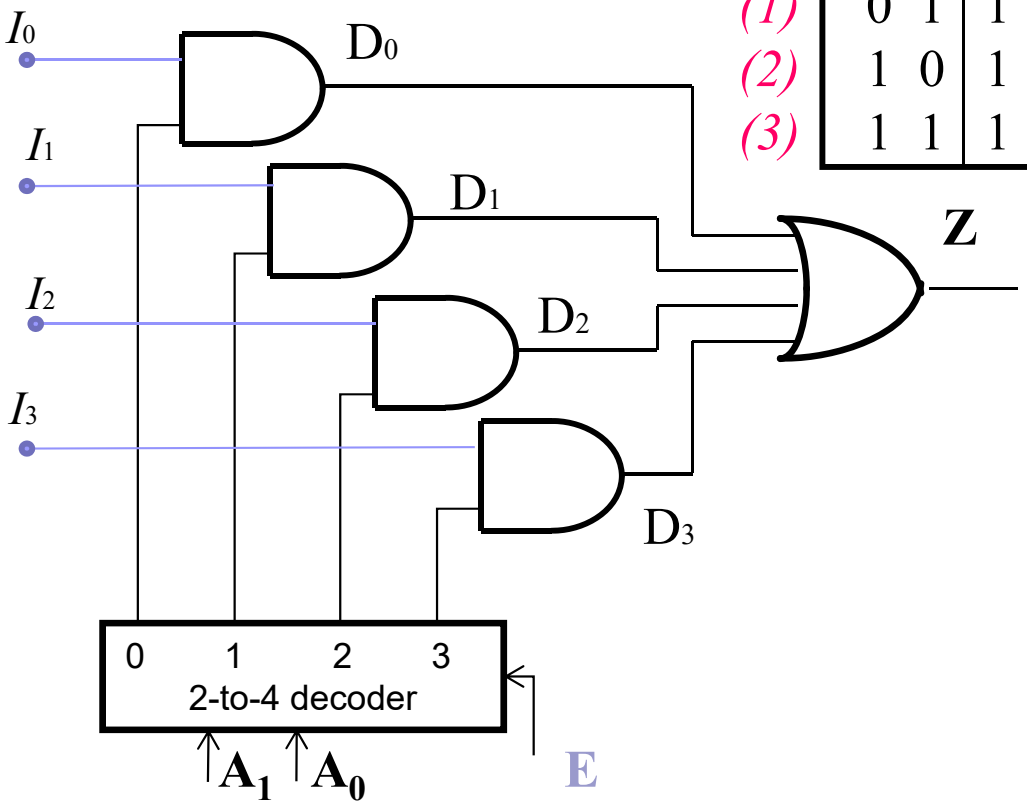

Random-access memory (RAM)
Central Processing Unit (CPU)
Input Devices
Input-output Processor (IOP)
Output Devices
Memory stick
DVD burner
Display
74138
Y0N
Y1N
Y2N
Y3N
Y4N
Y5N
Y6N
Y7N
A
B
C
G1
G2AN
G2BN
3:8 DECODER

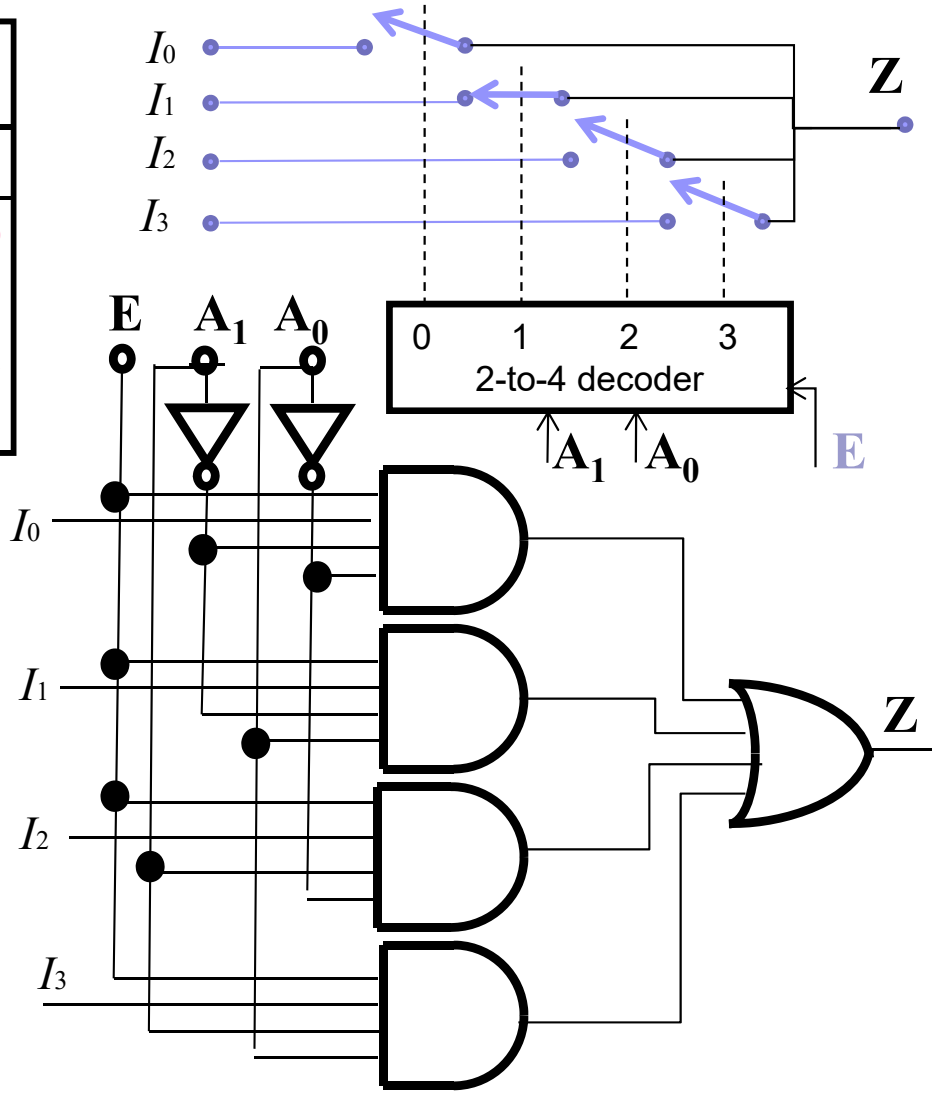
4-to-1-Line Multiplexer

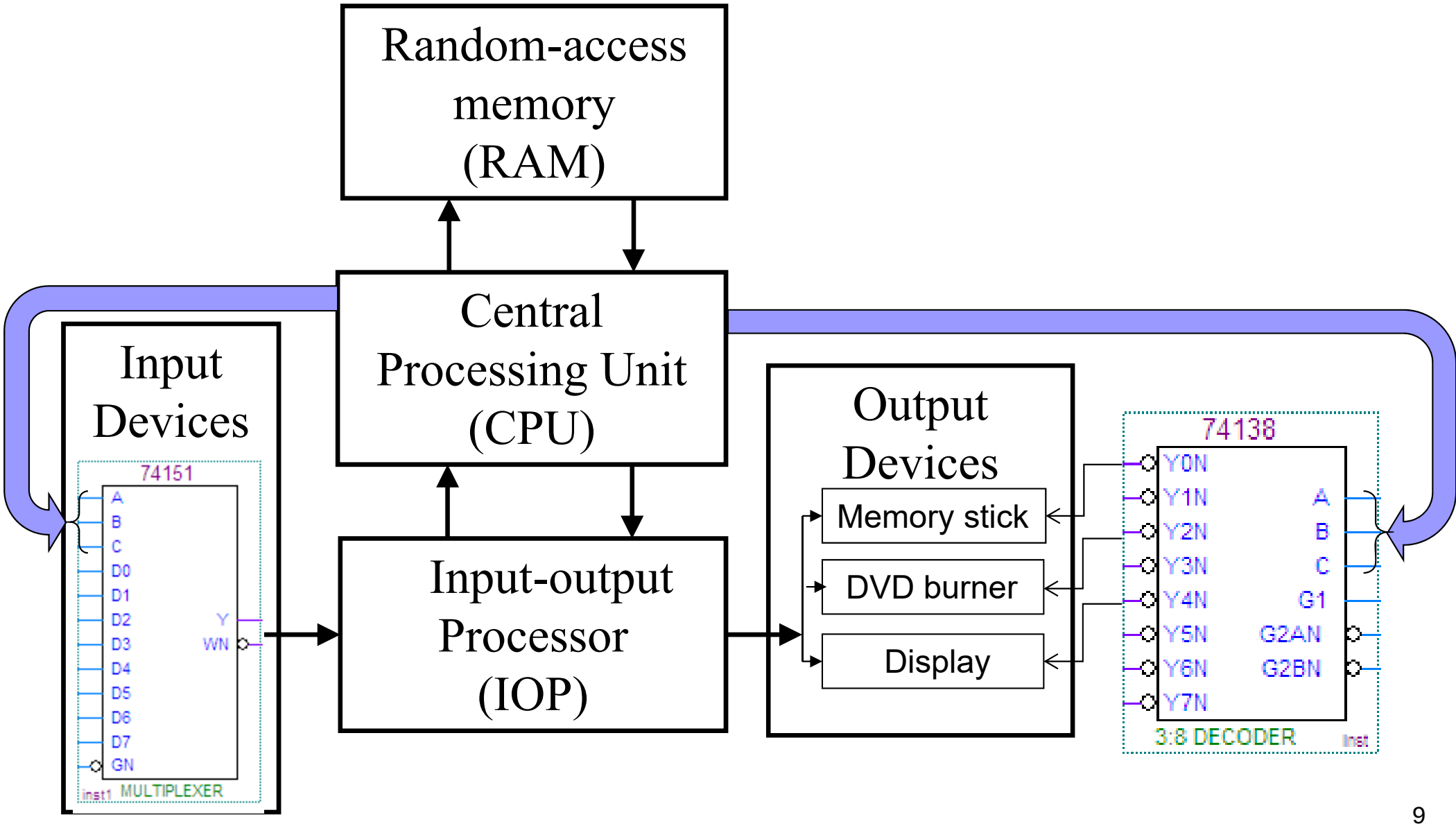
- A **multiplexer** is a combinational circuit that receives binary information from one of the 2^n input data bits (I_0, I_1, …) and directs it to a single output line (Z).
- The selection of a particular input data bit is determined by a set of n input **selection** bits (A_0, A_1, …) .



	A_1	A_0	E	Z
(x)	x	x	0	0
(0)	0	0	1	I_0
(1)	0	1	1	I_1
(2)	1	0	1	I_2
(3)	1	1	1	I_3





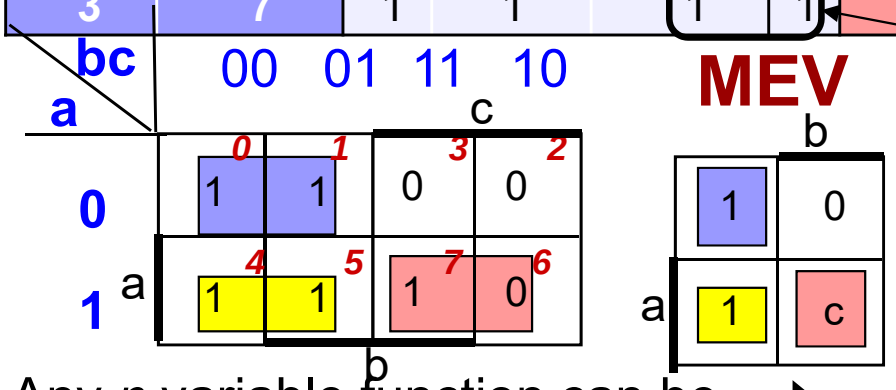

Random-access memory (RAM)
Central Processing Unit (CPU)
Input-output Processor (IOP)
Input Devices
74151
A
B
C
D0
D1
D2
D3
D4
D5
D6
D7
GN
Y
WN
inst1 MULTIPLEXER
Output Devices
Memory stick
DVD burner
Display
74138
Y0N
Y1N
Y2N
Y3N
Y4N
Y5N
Y6N
Y7N
A
B
C
G1
G2AN
G2BN
3:8 DECODER
inst

Logic Function Implementation with MUX

Map-Entered Variables (MEV)

Implement $F = \Sigma\,(0,1,4,5,7)$ with MUX

Decimal MEV	minterm Standard	a	b	c (MEV)	F	Output F(MEV)
0	0	0	0	0	1	1
0	1	0	0	1	1	
1	2	0	1	0	0	0
1	3	0	1	1	0	
2	4	1	0	0	1	1
2	5	1	0	1	1	
3	6	1	1	0	0	c
3	7	1	1	1	1	

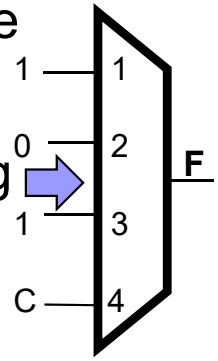


MEV

1. If the output variable is 0 for both standard minterms covered by a MEV square, then a 0 is written in that MEV map square.
2. If the output variable is a 1 for both standard minterms covered by a MEV map square, then a 1 is written in that MEV square.
3. If, for minterms covered by a MEV map square, the output variable (F) has the same value as the MEV, then the MEV is written into MEV map square.
4. If, for standard minterms covered by a MEV map square, the output and ME variables are complements, write the MEV complement into the MEV map square.
5. If, for standard minterms covered by a MEV map square, the output variable is a don't care term, write "x" into the MEV map square.
6. If, for standard minterms covered by a MEV map square, the output variable is a don't care term in one case and a 0 in the other, write 0 in the appropriate square.

Any *n* variable function can be implemented with

1. (2^n)-to-1 MUX directly;
2. (2^{n-1})-to-1 MUX by applying MEV;
3. (2^{n-k})-to-1 MUX + gates by using func. decomposition



MEV K-maps Minimization

Use K-maps to minimize

$$F = \Sigma\,(3,7,11,12,13,14,15,16,18) + d(24,25,26,27,28,29,30,31).$$

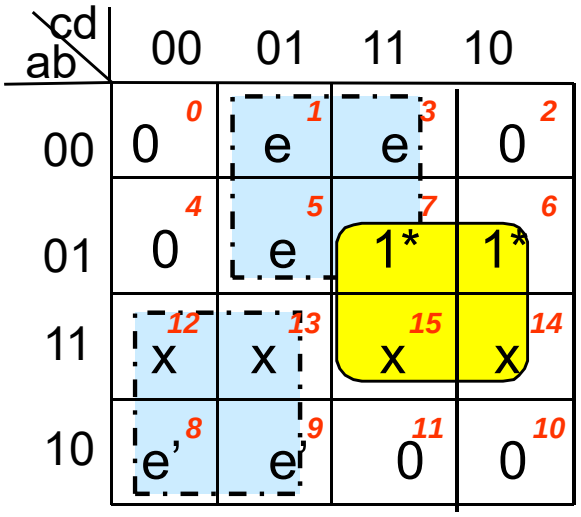
To find the simplified function from a MEV K-map, follow these steps:

1. Determine the EPI's consisting of only 1's along with any *don't care* terms that may exist (i.e., cover the 1's in the K-map).
2. Consider the 1's as don't care terms once step 1 is completed, because all of the 1's have been previously covered.
3. Group all identical MEV terms with 1's or don't care terms to maximize the MEV EPI size, i.e., sequentially take **only one MEV = 1** at a time, why all the others are considered 0, and derive the sum of product terms that cover that MEV=1. Determine the MEV EPI's by reading the K-map in the normal fashion. Then AND the MEV variable or expression with the remaining map variables.

$$F = bc + a'de + ac'e'$$

MEV	m	a	b	c	d	e	F
0	0	0	0	0	0	0	0
	1	0	0	0	0	1	0
1	2	0	0	0	1	0	0
	3	0	0	0	1	1	1
2	4	0	0	1	0	0	0
	5	0	0	1	0	1	0
3	6	0	0	1	1	0	0
	7	0	0	1	1	1	1
4	8	0	1	0	0	0	0
	9	0	1	0	0	1	0
5	10	0	1	0	1	0	0
	11	0	1	0	1	1	1
6	12	0	1	1	0	0	1
	13	0	1	1	0	1	1
7	14	0	1	1	1	0	1
	15	0	1	1	1	1	1

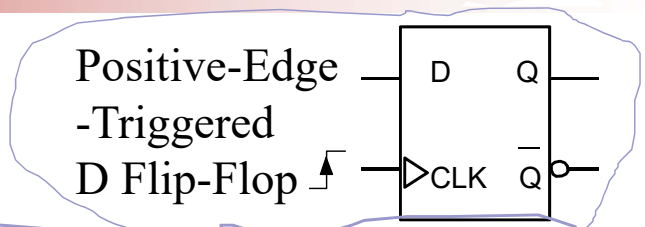
MEV	m	a	b	c	d	e	F
8	16	1	0	0	0	0	1
	17	1	0	0	0	1	0
9	18	1	0	0	1	0	1
	19	1	0	0	1	1	0
10	20	1	0	1	0	0	0
	21	1	0	1	0	1	0
11	22	1	0	1	1	0	0
	23	1	0	1	1	1	0
12	24	1	1	0	0	0	x
	25	1	1	0	0	1	x
13	26	1	1	0	1	0	x
	27	1	1	0	1	1	x
14	28	1	1	1	0	0	x
	29	1	1	1	0	1	x
15	30	1	1	1	1	0	x
	31	1	1	1	1	1	x



Outline

- Decoders, Multiplexers
- **Registers +** Common sense design strategies
  - **Register with Parallel Load**
  - Shift Registers
  - Bidirectional Shift Register with Parallel Load
- Binary Counters
  - Binary Counter with Parallel Load
- Multi-function Registers

Registers

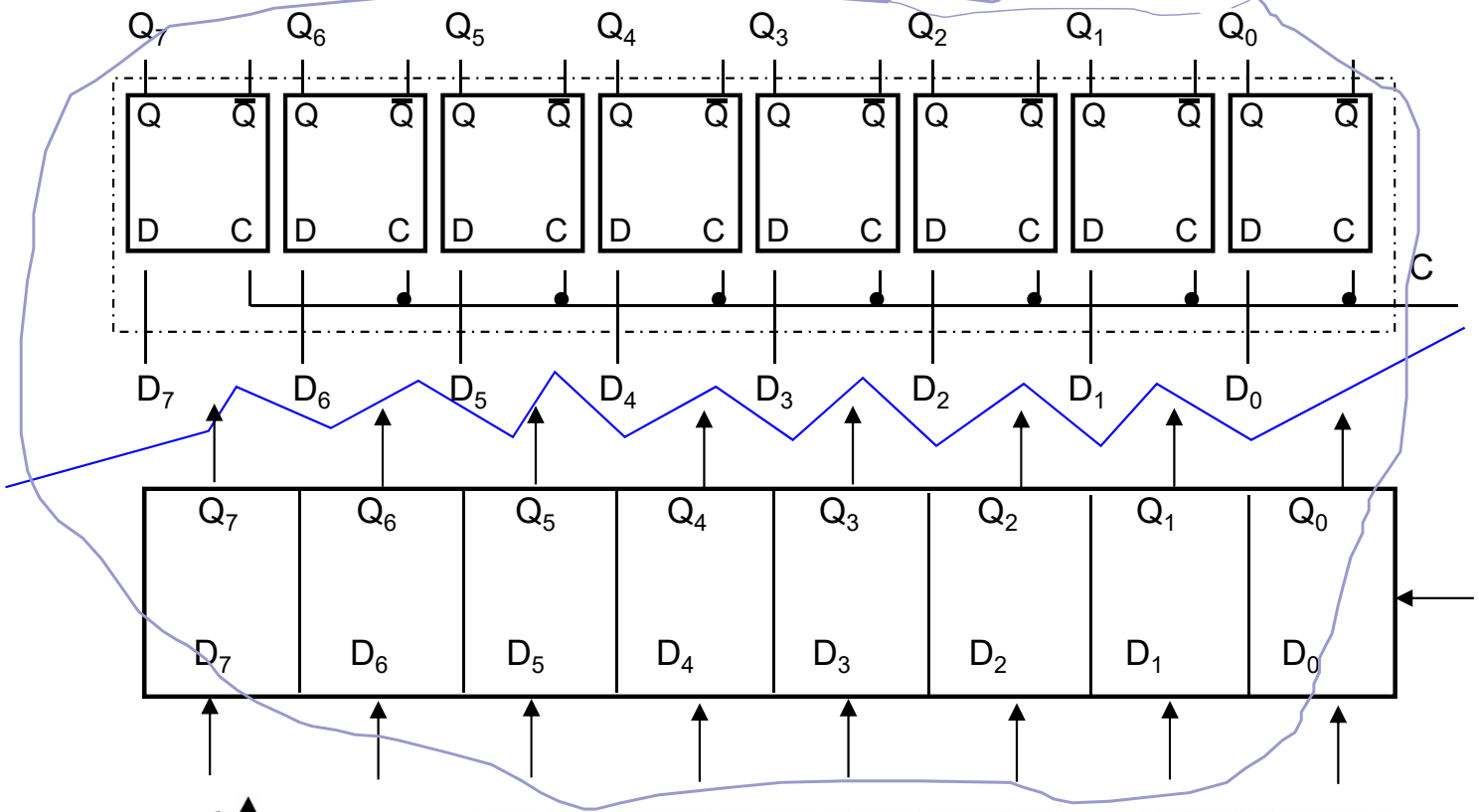


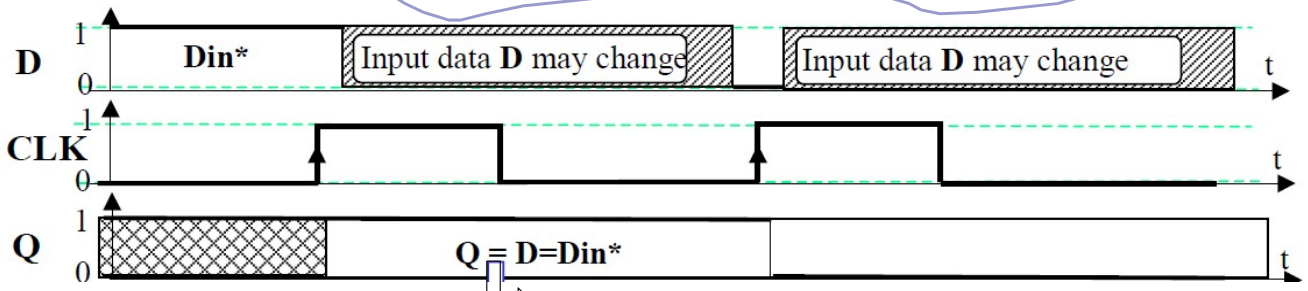
Characteristic Equation:

$Q_{n+1} = D_n$

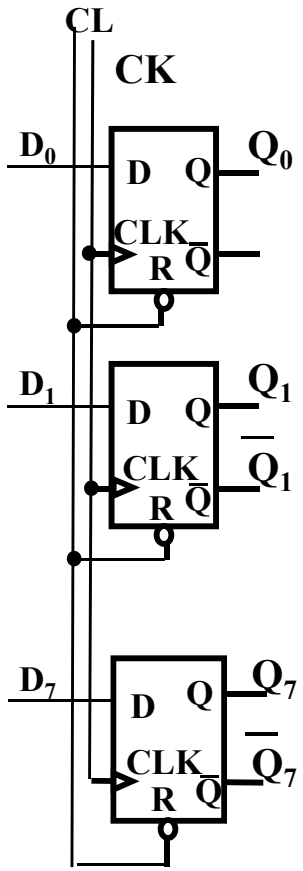
D_n	Q_n	Q_{n+1}
0	0	0
0	1	0
1	0	1
1	1	1

Characteristic Table





The state of the flip-flop's output Q copies input D when the positive edge of the clock CLK occurs

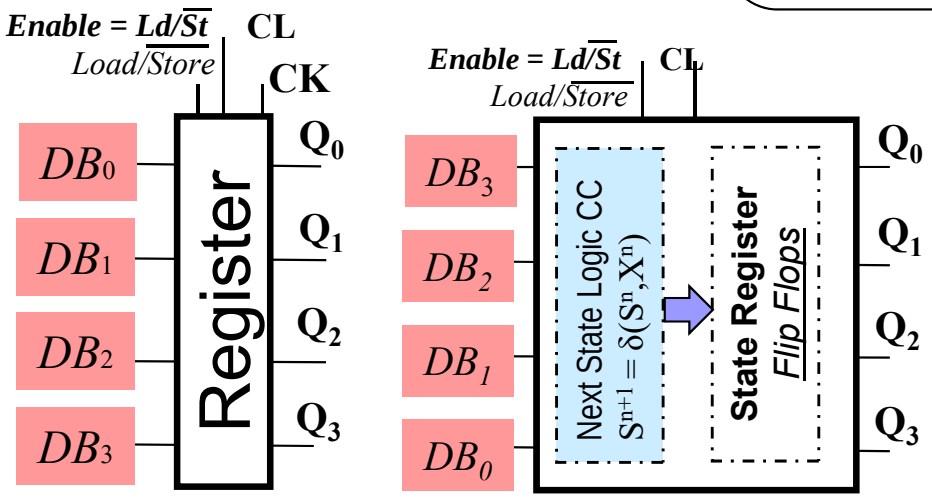


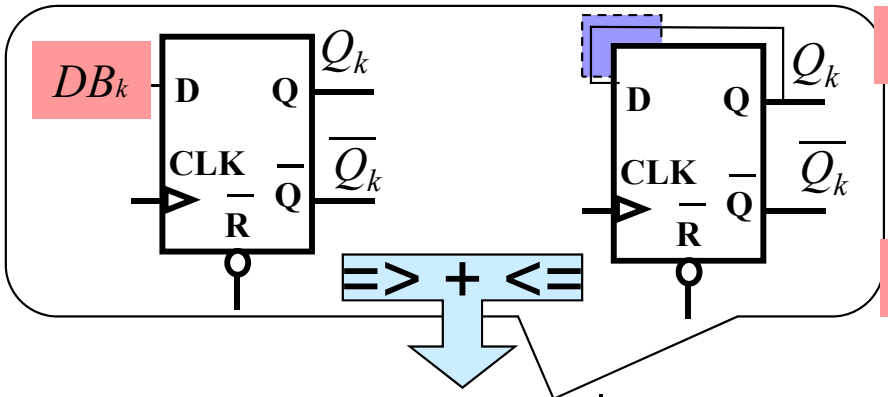
Common sense design strategies

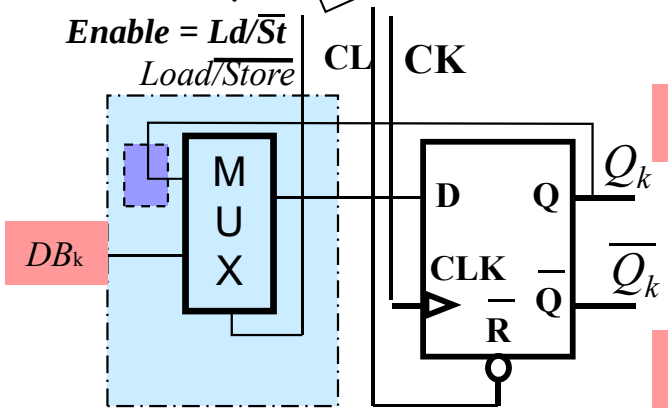
- Focus on one bit first. In many cases the rest would have a similar pattern.
- Try to reason with your own words. Do the computation with “words" first.
- See if you can define the “rules" governing the logic circuit, and put them in your own words.
- The above design strategies may save you a significant amount of time and facilitate your task as a designer. However, they may not be trivial in all the cases.

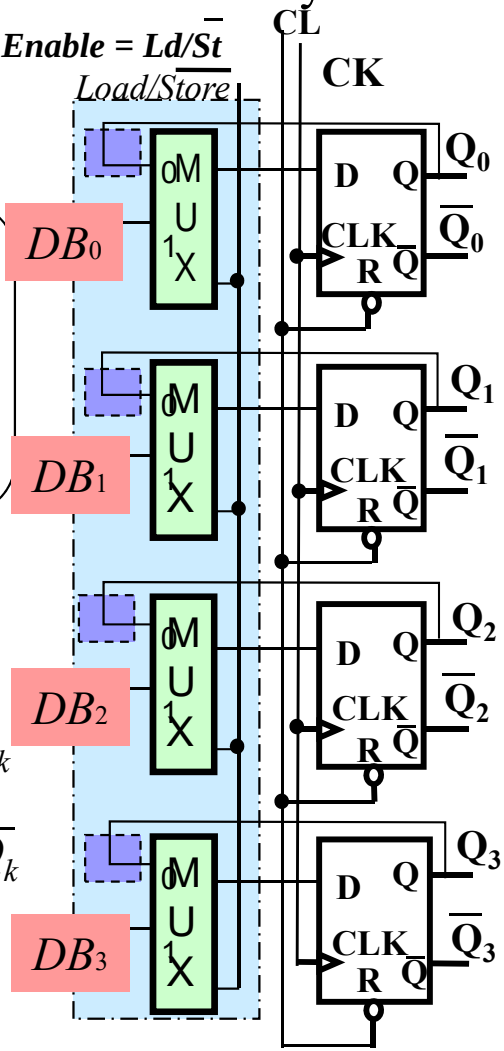
Register with Parallel Load

Enable	Q_k^{n+1}	Function
0	Q_k^n	Store
1	DB_k	Load



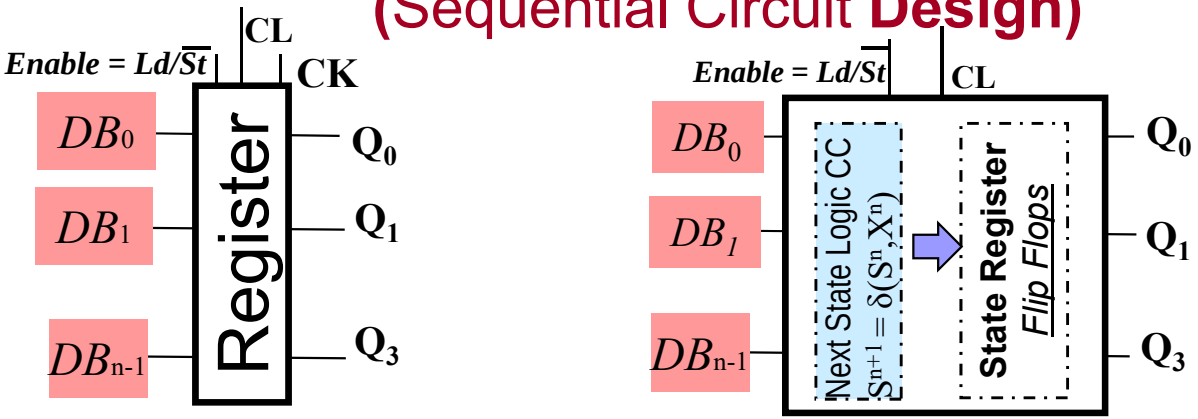






Register with Parallel Load

(Sequential Circuit Design)



- When the *load* input ***Ld / $\overline{St}$*** = 1, **at the positive clock pulse transition time**, the data in the four input bits *DB0* is transferred into the register; otherwise
- when load input ***Ld / $\overline{St}$*** = 0, the register output bits maintain their values.

Steps 1,2

State / Transition Table (MEV format)

Enable = Ld / $\overline{St}$

Enable	Q_k^{n+1}	***Function***
0 ($\overline{St}$)	Q_k^n	*Store*
1 (*Ld*)	DB_k	*Load*

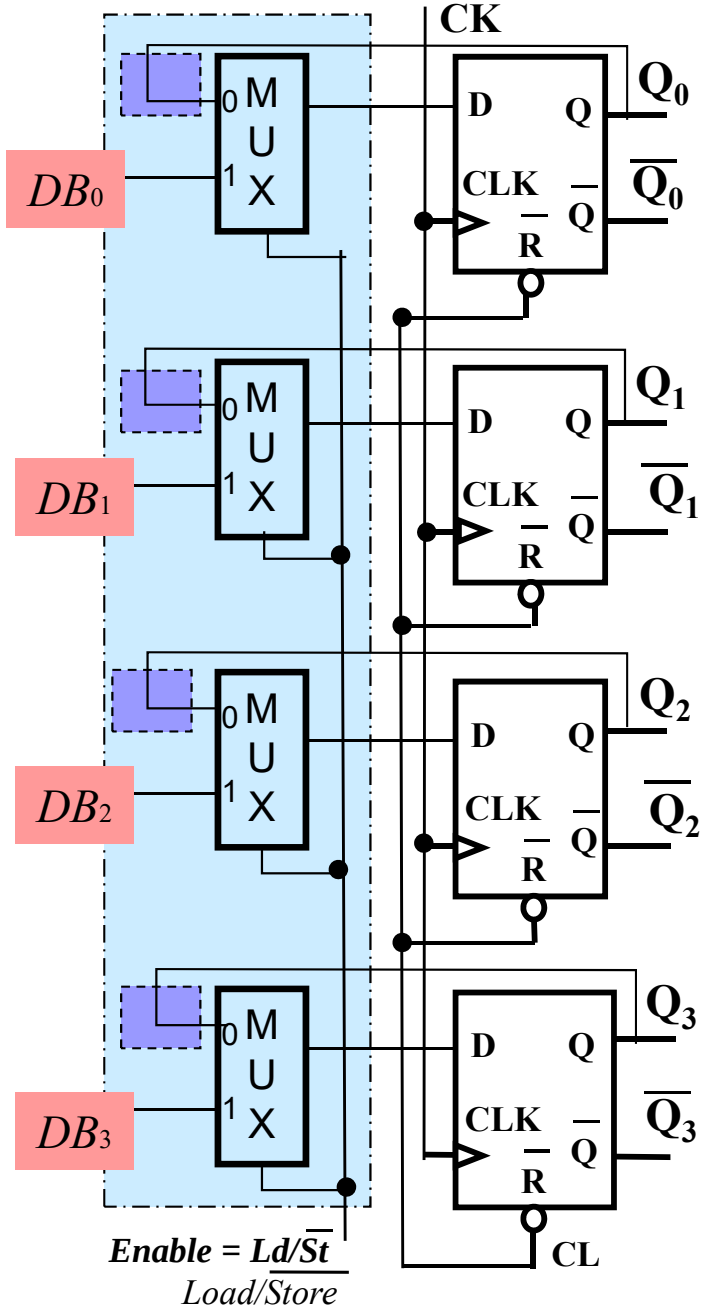
*The **transition function** (δ) of the Parallel Register:*

$$Q_k(n+1) = Ld\ DB_k + \overline{St}\ Q_k(n);\ k=\{0,1,2,3\}$$
$$= Enable\ DB_k + \overline{Enable}\ Q_k;$$

Step 3 *If the Parallel Register is implemented with D-FF's, its excitation equation gives:* $D(n) = Q_k(n+1)$

Step 4 $\Rightarrow D_k = Enable\ DB_k + \overline{Enable}\ Q_k;\ k=\{0,1,2,3\}$

Step 5 Equations D_k can be implemented with gates or MUX-es as shown below:

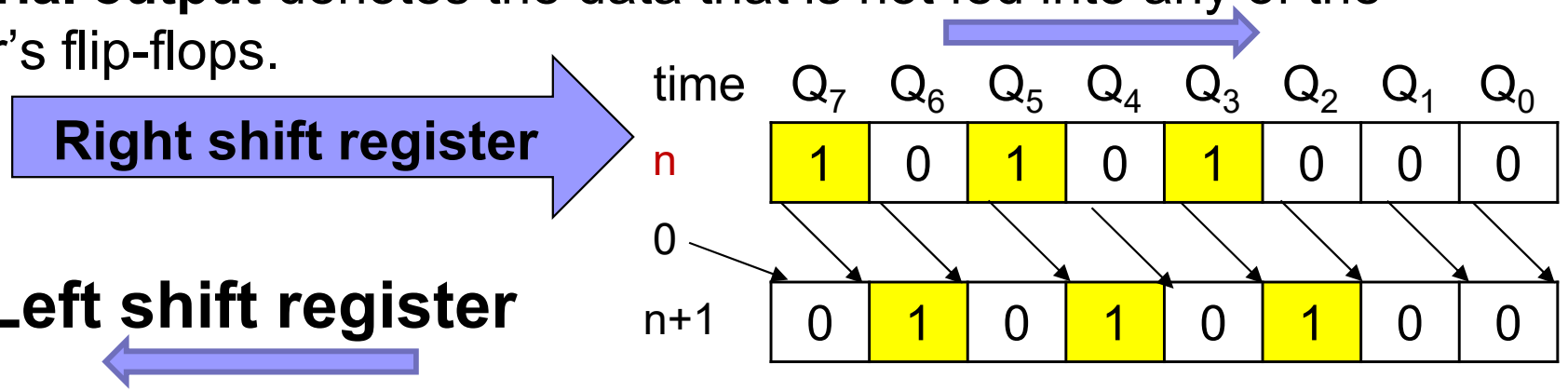


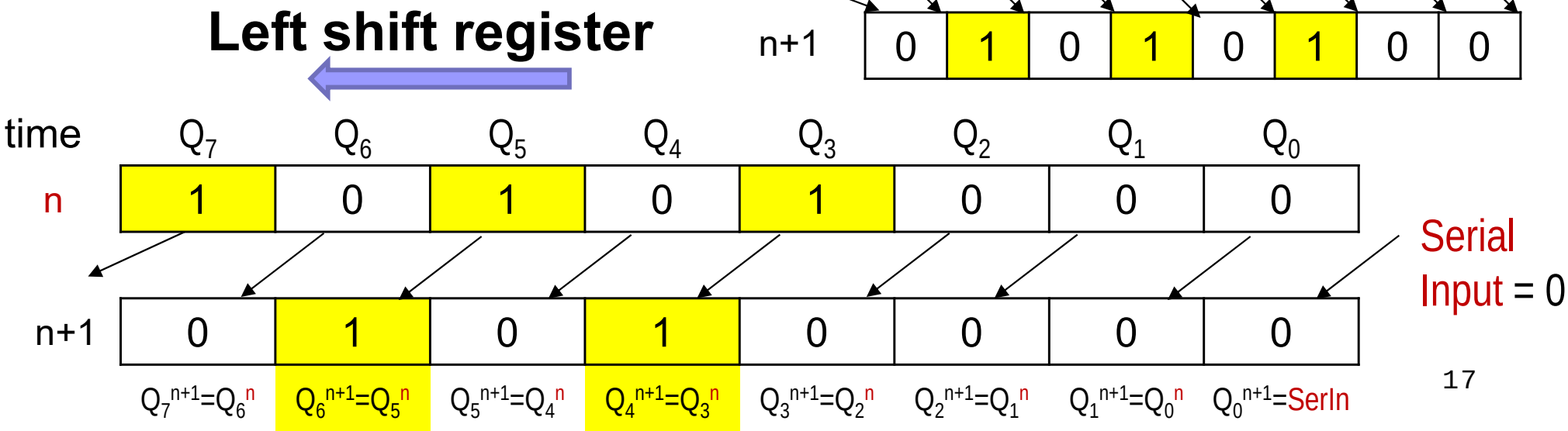
Outline

- Decoders, Multiplexers
- Registers + Common sense design strategies
  - Register with Parallel Load
  - **Shift Registers**
  - Bidirectional Shift Register with Parallel Load
- Binary Counters
  - Binary Counter with Parallel Load

Shift Registers

- A **shift register** is a register that is capable of shifting its binary information in one or both directions.
- The logical configuration of a shift register consists of a chain of cascaded flip-flops.
- The **serial input** denotes the external input fed into the shift register.
- The **serial output** denotes the data that is not fed into any of the register's flip-flops.



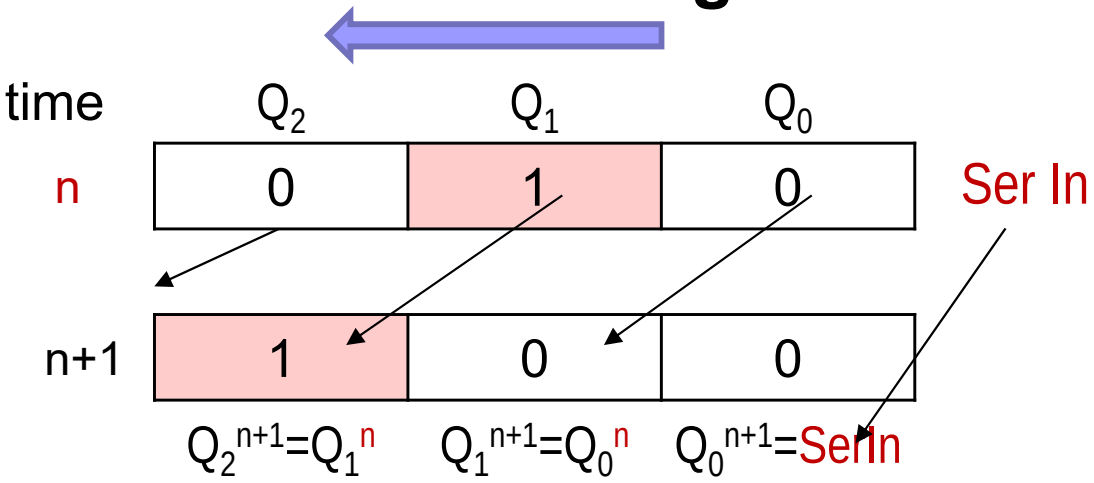


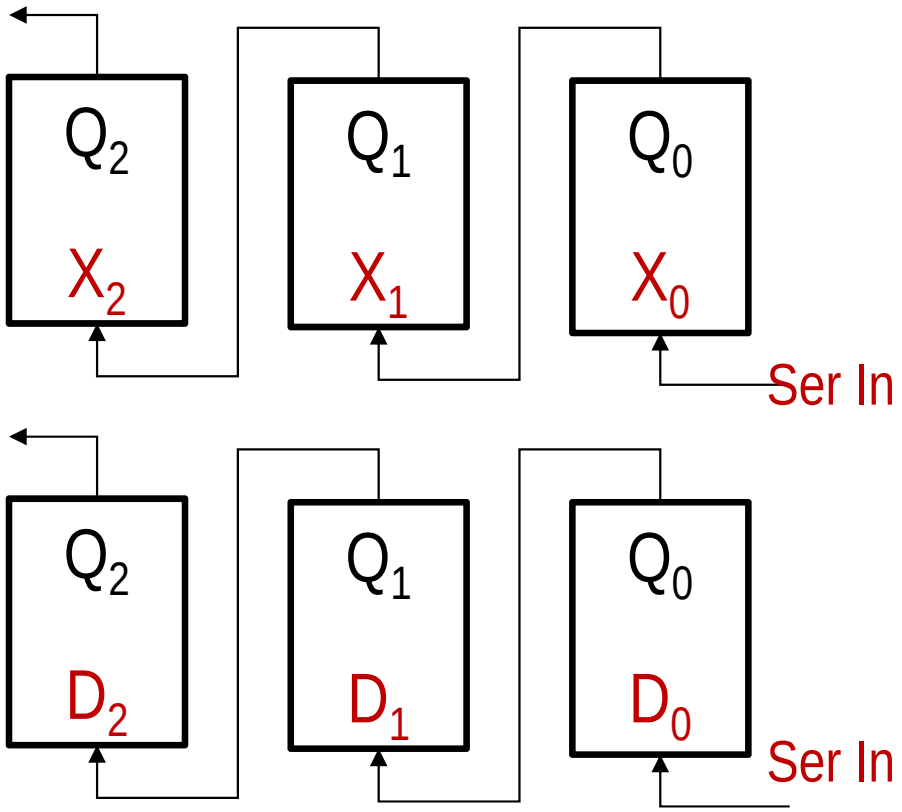
Left Shift Register

The Next State (S^{n+1}) *of a sequential circuit is a function* (δ) *of its input* (X) *and of its present state* (S^n): $S^{n+1} = \delta(X, S^n)$.

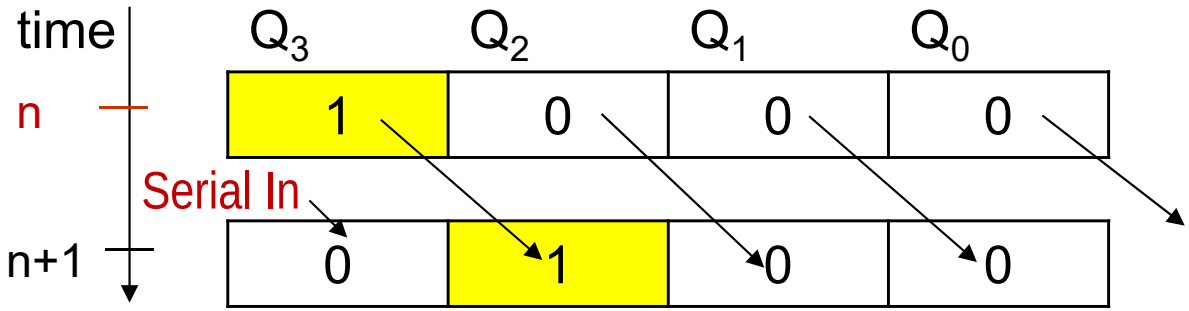
For left shift, the input X_i of each FF_i is connected to the FF_{i-1} output from its right $X_i = Q_{i-1} \Rightarrow Q_i^{n+1} = \delta(Q_{i-1}^n, Q_i^n)$; $i = \{0,1,2,3\}$

Left shift register

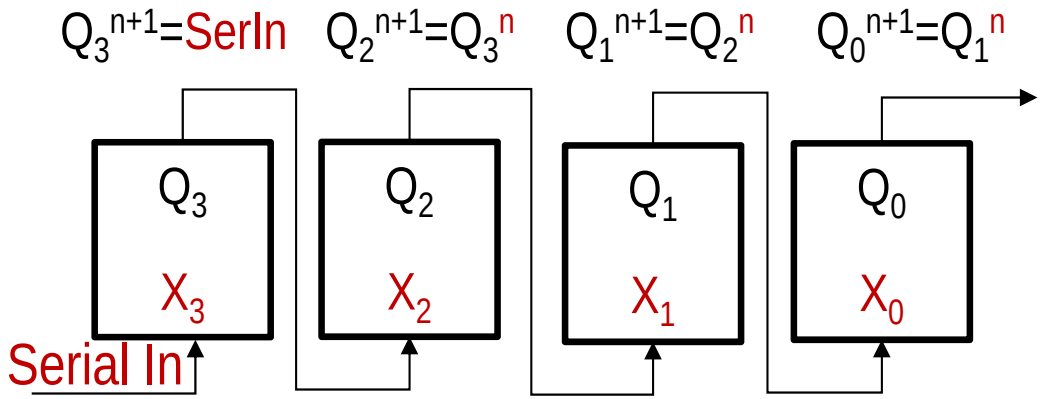


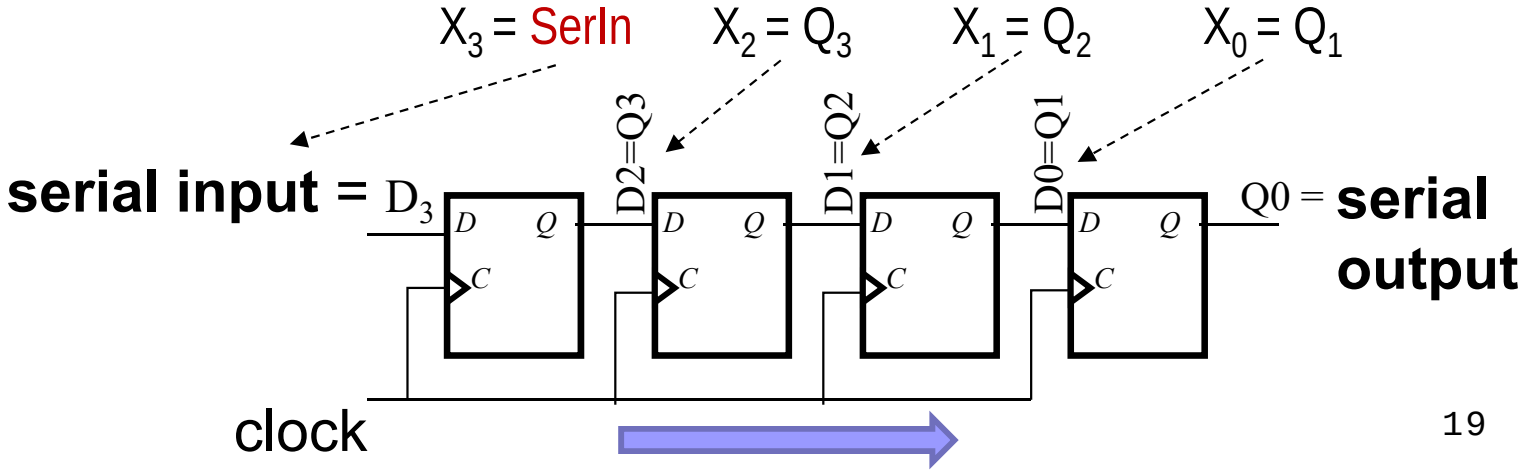


Right Shift Register



- When implementing with D FF's: $Q^{n+1} = D$
- So, the output Q of each flip-flop is connected to the input D of the next flip-flop





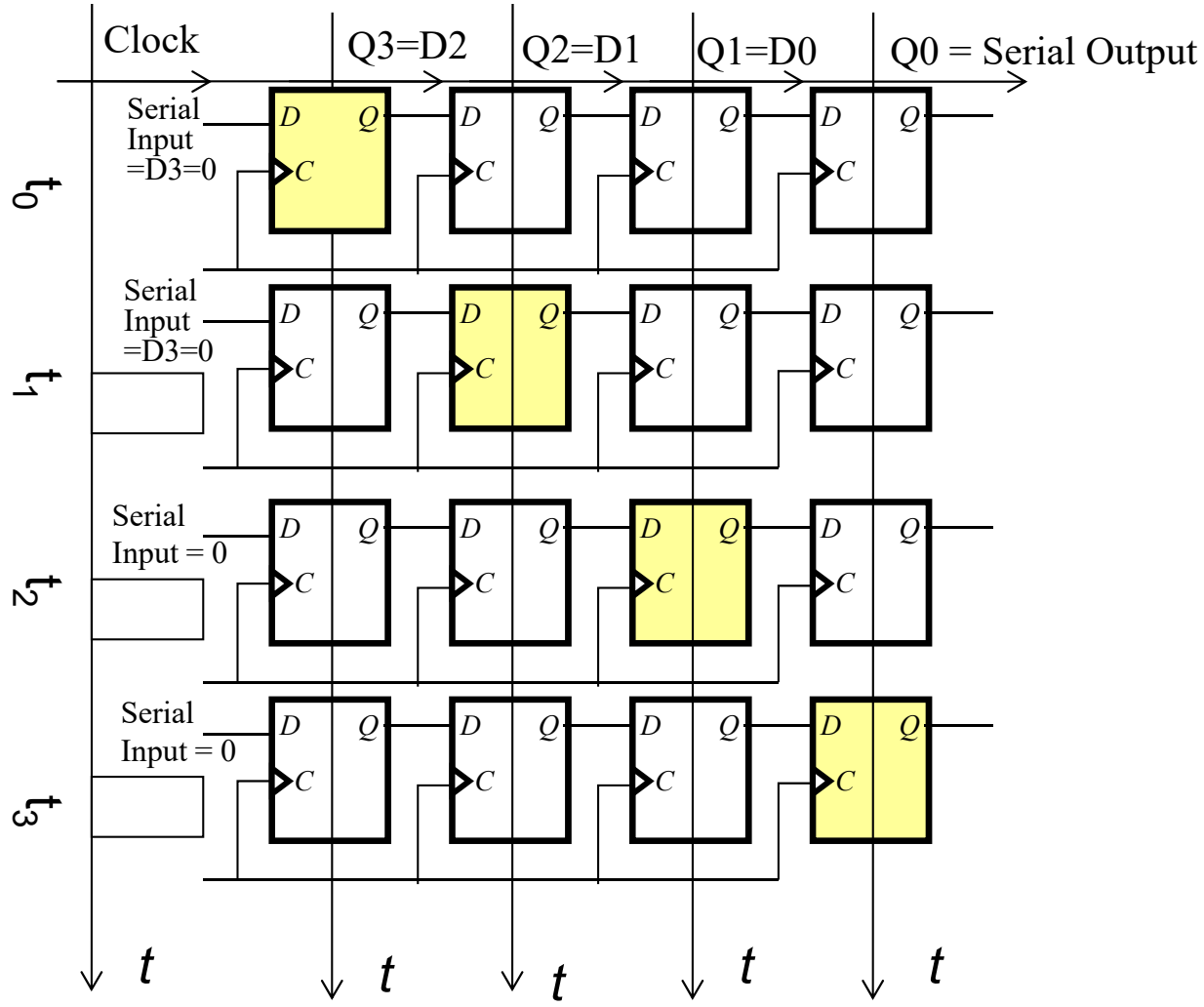
Right Shift Register

Serial Input = 0
Initial State:
$t_0 : Q_3Q_2Q_1Q_0=1000$

$t_1 : Q_3Q_2Q_1Q_0=0100$

$t_2 : Q_3Q_2Q_1Q_0=0010$

$t_3 : Q_3Q_2Q_1Q_0=0001$



Right Shift Register

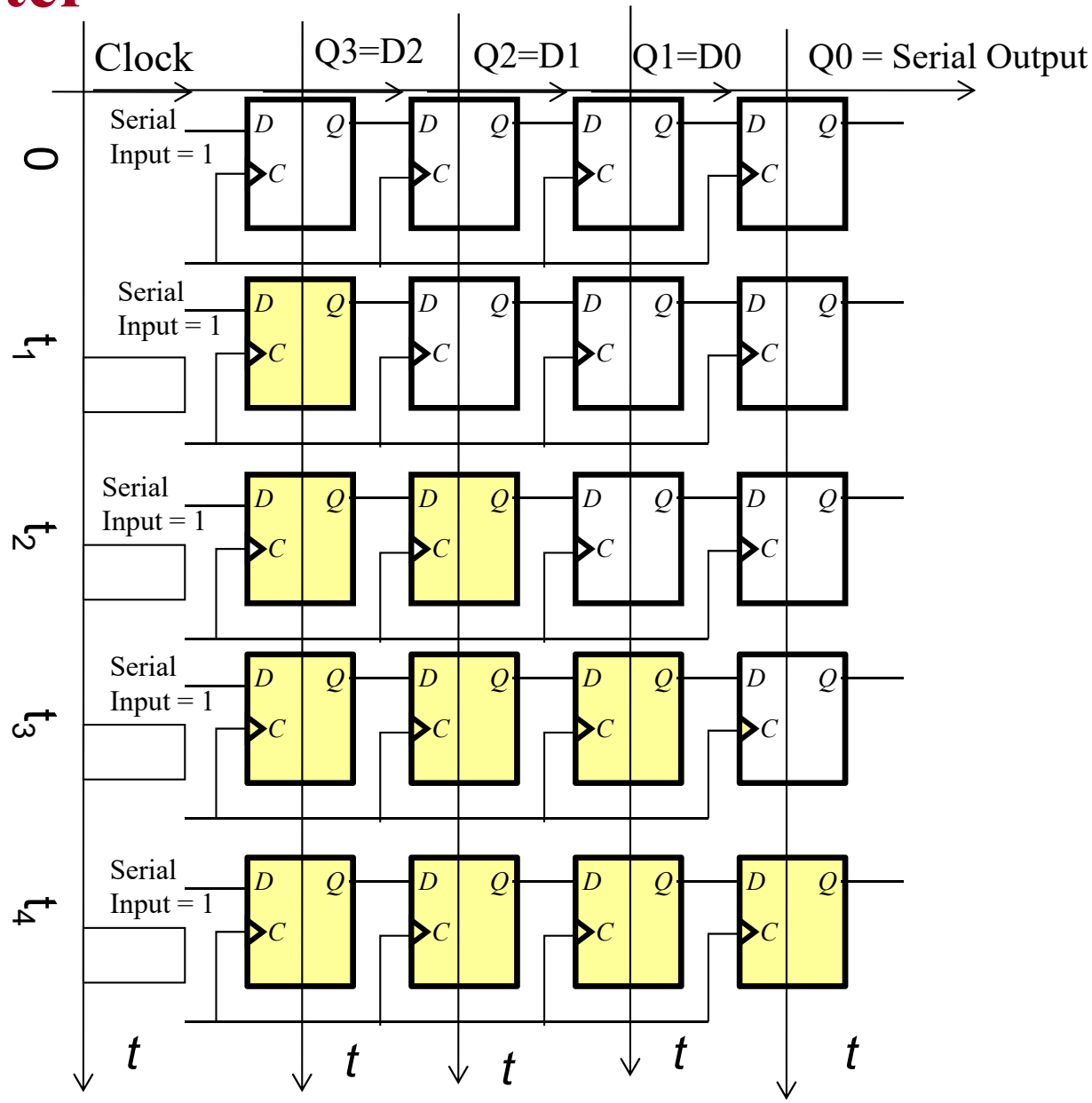
Serial Input = 1
Initial State: 0000

$t_1 : Q_3Q_2Q_1Q_0=1000$

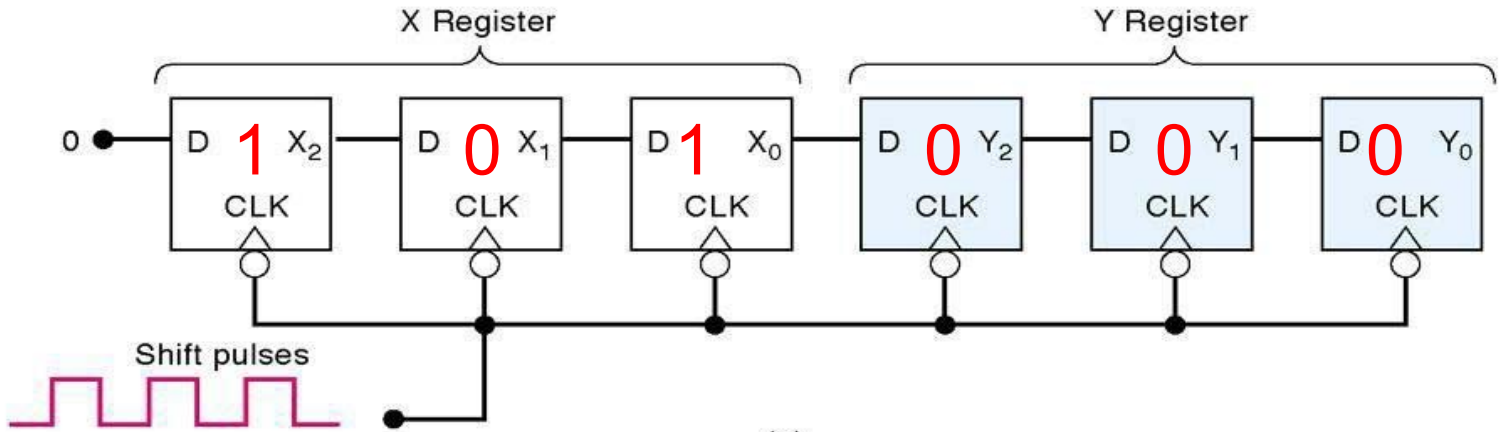
$t_2 : Q_3Q_2Q_1Q_0=1100$

$t_3 : Q_3Q_2Q_1Q_0=1110$

$t_4 : Q_3Q_2Q_1Q_0=1111$



Serial transfer from register X to register Y



(a)

X_2	X_1	X_0	Y_2	Y_1	Y_0	
1	0	1	0	0	0	Before pulses applied
0	1	0	1	0	0	After first pulse
0	0	1	0	1	0	After second pulse
0	0	0	1	0	1	After third pulse

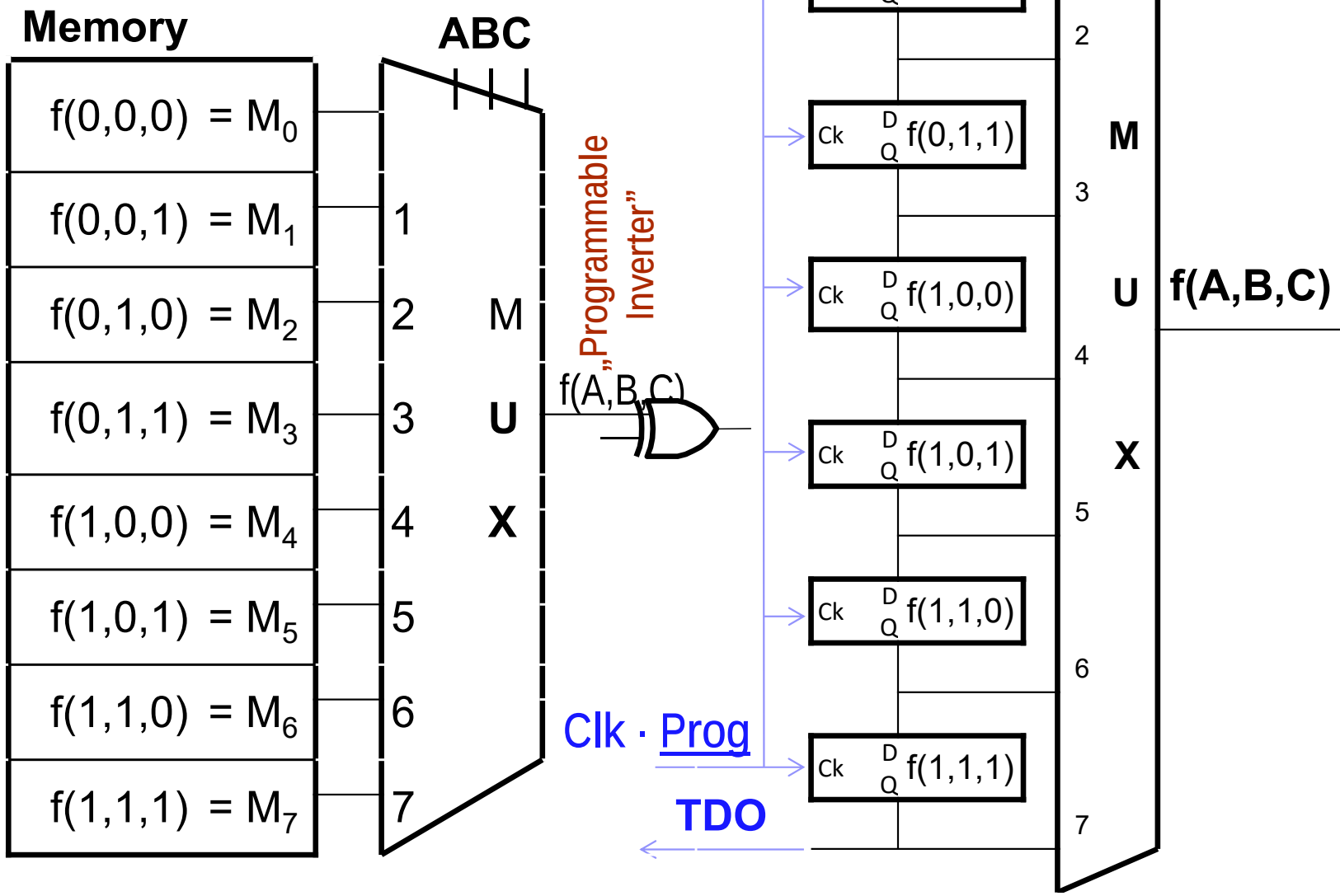
(b)

FPGA Programmable Logic Element (LE)

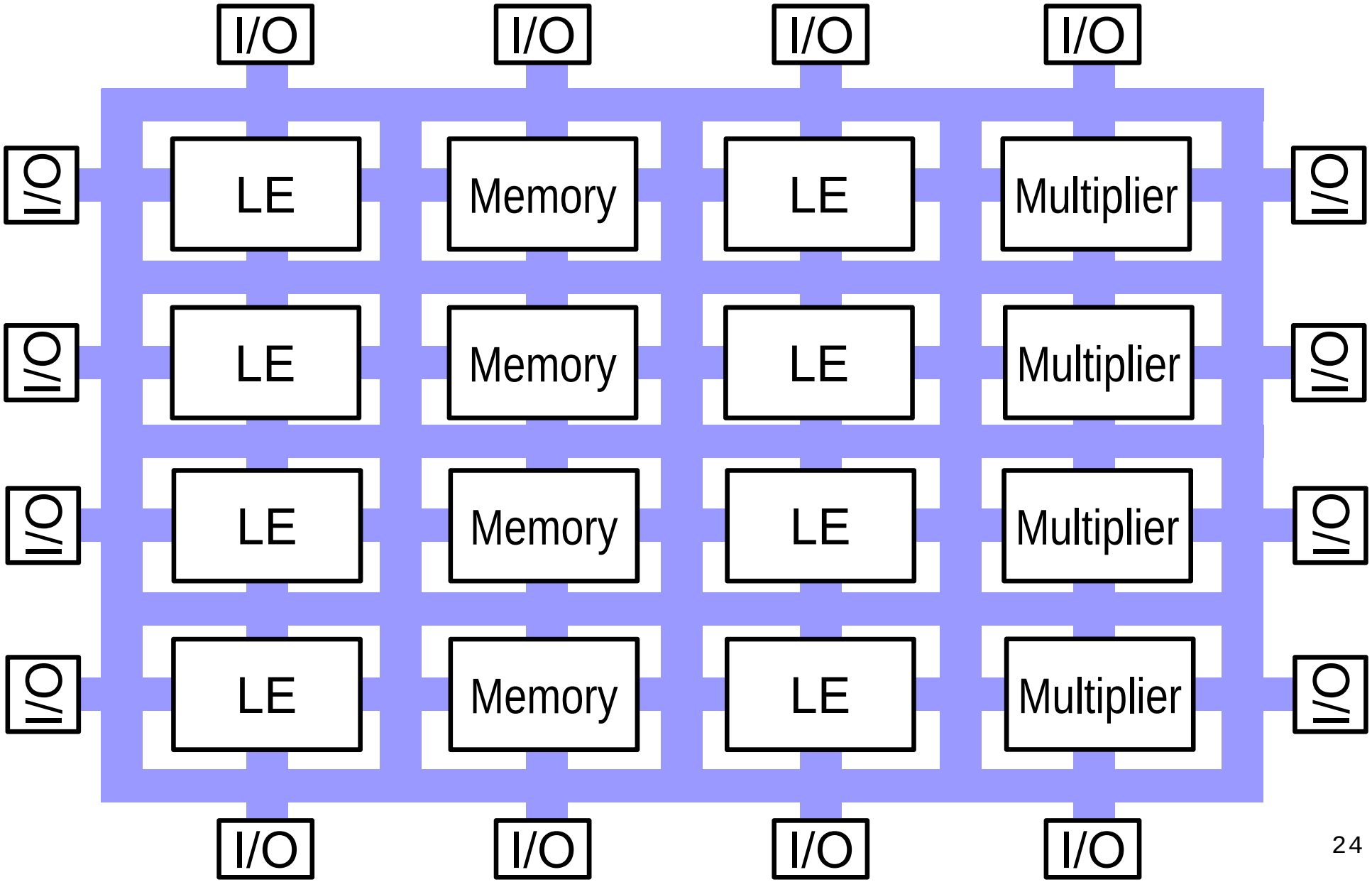
Truth Table

A B C	f(A,B,C)
0 0 0	f(0,0,0)
0 0 1	f(0,0,1)
0 1 0	f(0,1,0)
0 1 1	f(0,1,1)
1 0 0	f(1,0,0)
1 0 1	f(1,0,1)
1 1 0	f(1,1,0)
1 1 1	f(1,1,1)

Programmable Logic Element



FPGA


I/O
I/O
I/O
I/O
I/O
I/O
I/O
I/O
LE
Memory
LE
Multiplier
LE
Memory
LE
Multiplier
LE
Memory
LE
Multiplier
LE
Memory
LE
Multiplier
I/O
I/O
I/O
I/O

Outline

- Decoders, Multiplexers
- Registers + Common sense design strategies
  - Register with Parallel Load
  - Shift Registers
  - **Bidirectional Shift Register with Parallel Load**
- Binary Counters
  - Binary Counter with Parallel Load

Multi-function Registers

The most general register (Bidirectional Shift Register and Parallel Load) has the following capabilities:

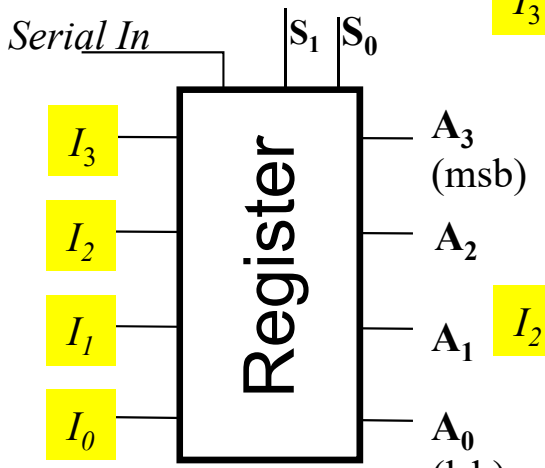
1. A clock pulse input to synchronize all operations.
2. A shift right operation and a serial input line associated to it.
3. A shift left operation and a serial input line associated to it.
4. A parallel load operation and n input lines associated to it.
5. *n* parallel output lines
6. A "no change" control state to leave the *n* parallel output lines unchanged for the next clock pulse.

Mode Control

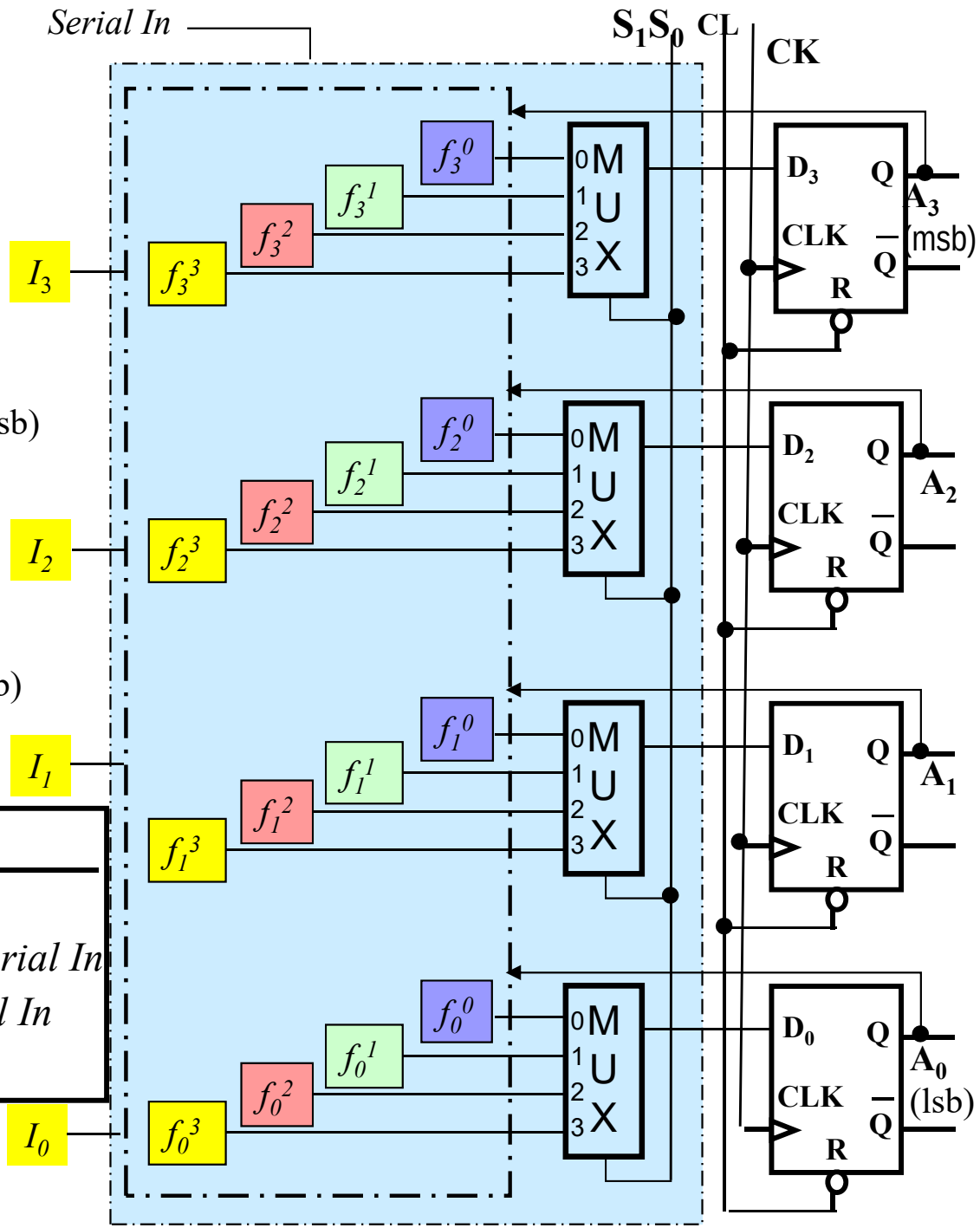
$S_1 S_0$	Register operation
0 0	No change
0 1	Shift right (msb→→lsb)
1 0	Shift left (lsb →→ msb)
1 1	Parallel load

Top-down design

Bidirectional Shift Register and Parallel Load with D Flip Flops

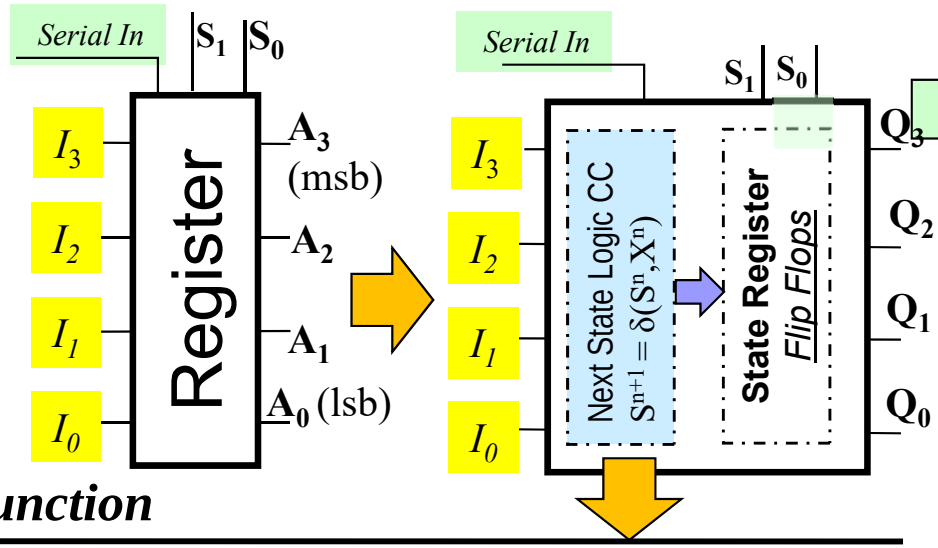


	S_1	S_0	Function
(f^0)	0	0	*No Change=Store*
(f^1)	0	1	*Shift Right =down: msb->lsb & D_3=Serial In*
(f^2)	1	0	*Shift Left* = up: *msb->lsb & D_o = Serial In*
(f^3)	1	1	*Parallel Load DB*

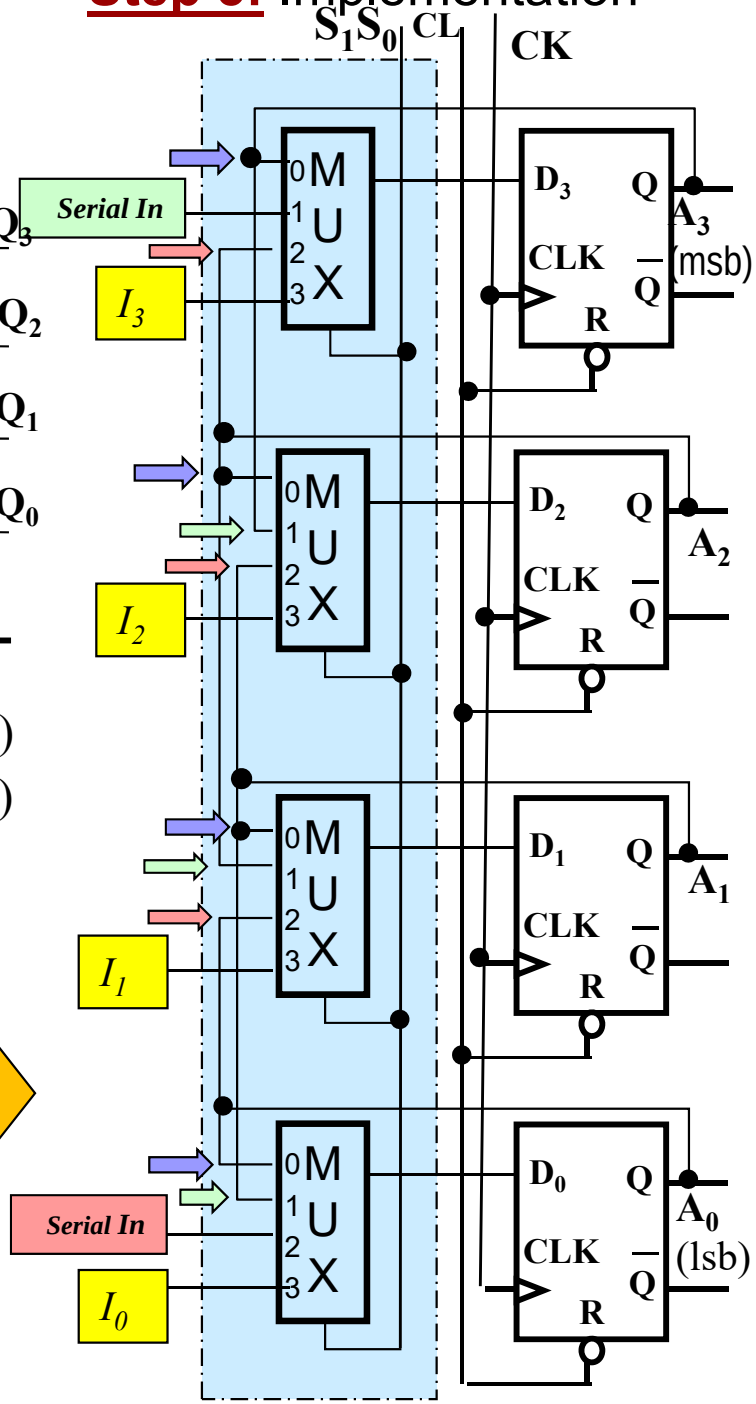


Top-down design

Bidirectional Shift Register and Parallel Load with D Flip Flops



Step 5: Implementation



Steps 1,2

	S_1	S_0	A_k^{n+1}	**Function**
(f^0)	0	0	A_k^n	*No Change=Store* (k={0,1,2,3})
(f^1)	0	1	A^n_{k+1}	*Shift Right* (k={0,1,2}: *msb->lsb* & D_3=*Serial In*)
(f^2)	1	0	A^n_{k-1}	*Shift Left* (k={1,2,3}: *lsb => msb* & D_0=*Serial In*)
(f^3)	1	1	I_k	*Parallel Load* (k={0,1,2,3})

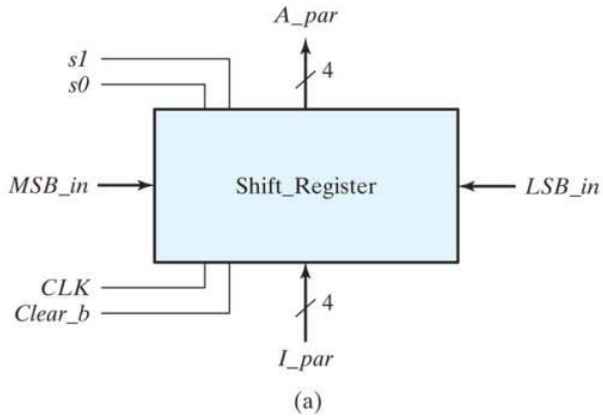
State / Transition Table (MEV format)

Steps 3,4

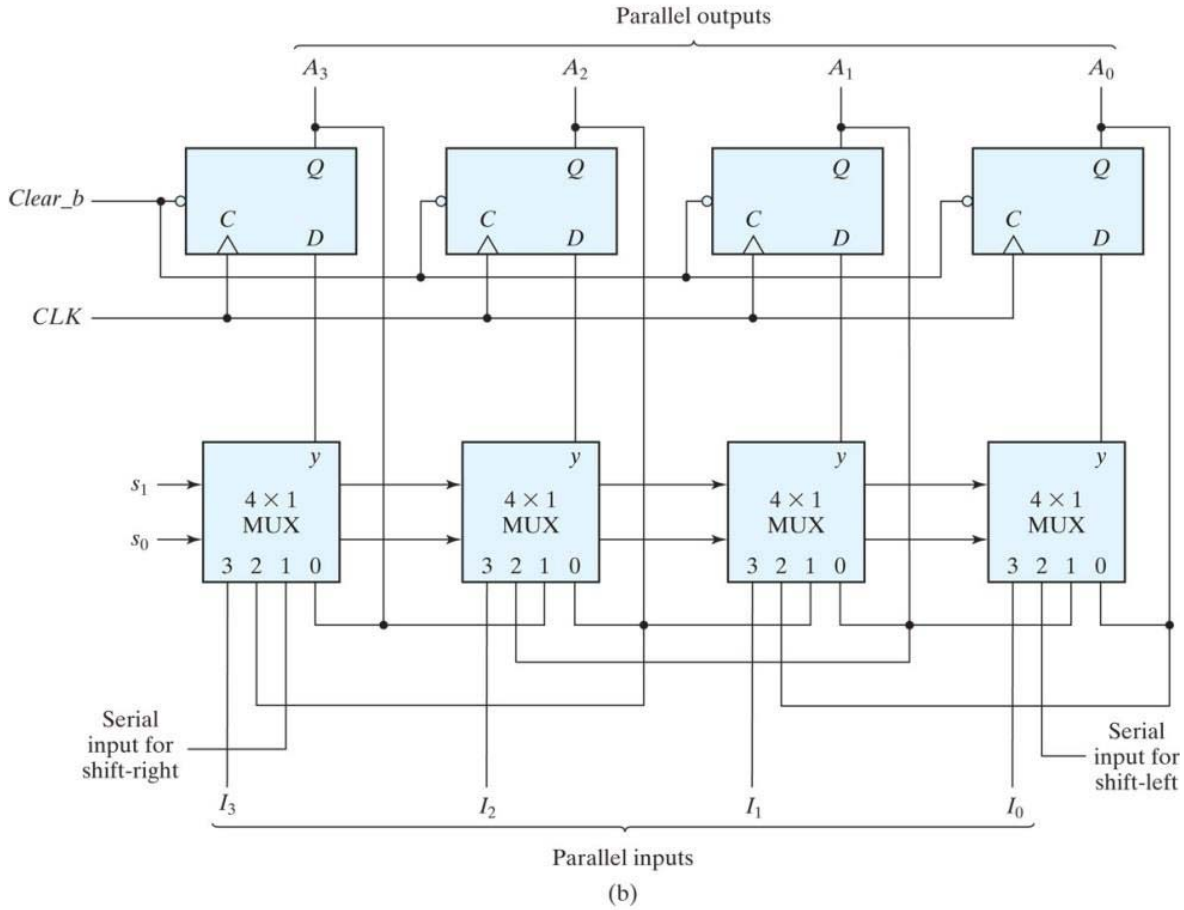
D-FF's excitation equations

	S_1	S_0	$D_k = A_k^{n+1}$
(f^0)	0	0	$D_k = A_k$; k={0,1,2,3}
(f^1)	0	1	$D_k = A_{k+1}$; k={0,1,2} D_3=*Serial In*
(f^2)	1	0	$D_k = A_{k-1}$; k={1,2,3} D_0=*Serial In*
(f^3)	1	1	$D_k = I_k$; k={0,1,2,3}

Four-bit universal shift register



(a)



(b)

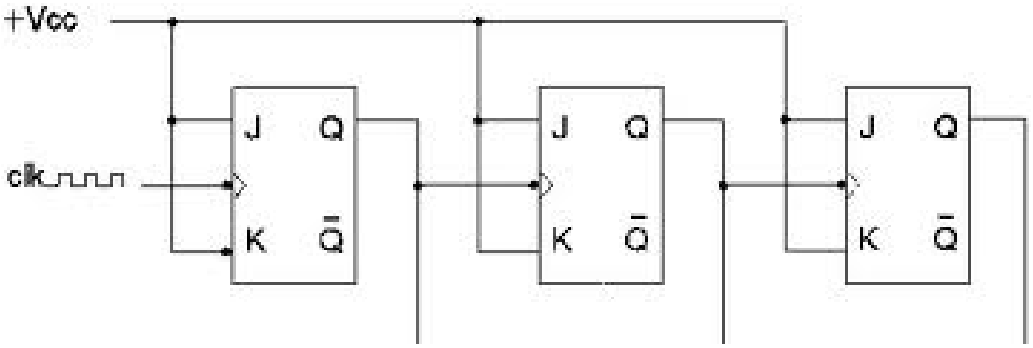
Outline

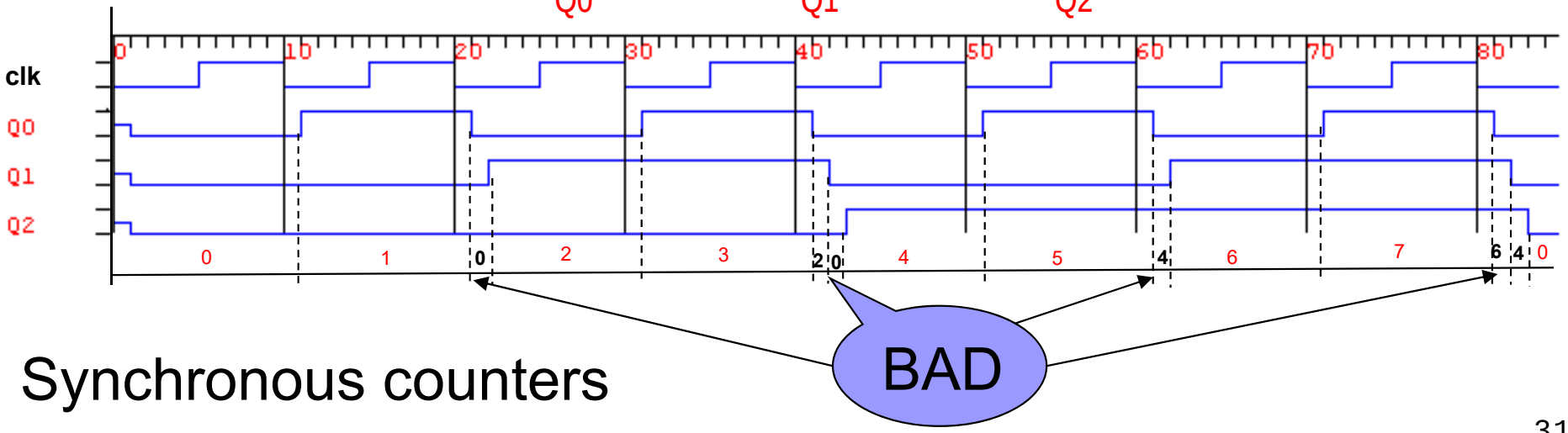
- Decoders, Multiplexers
- Registers + Common sense design strategies
  - Register with Parallel Load
  - Shift Registers
  - Bidirectional Shift Register with Parallel Load
- **Binary Counters**
  - Binary Counter with Parallel Load

Counters

- = Register that goes through a prescribed series of states
- There are two main types of counters:

1. Asynchronous counters: also known as Ripple counters - Flip flop's output's serve as clock for triggering connected flip flops

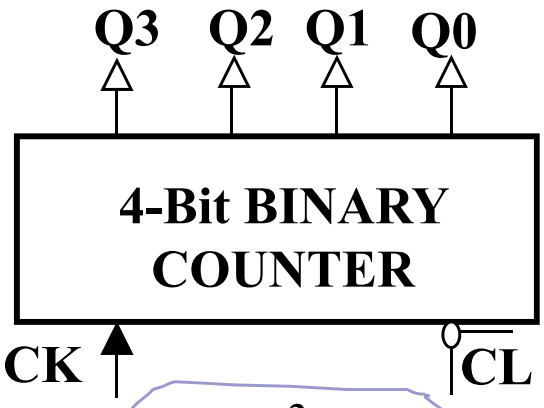




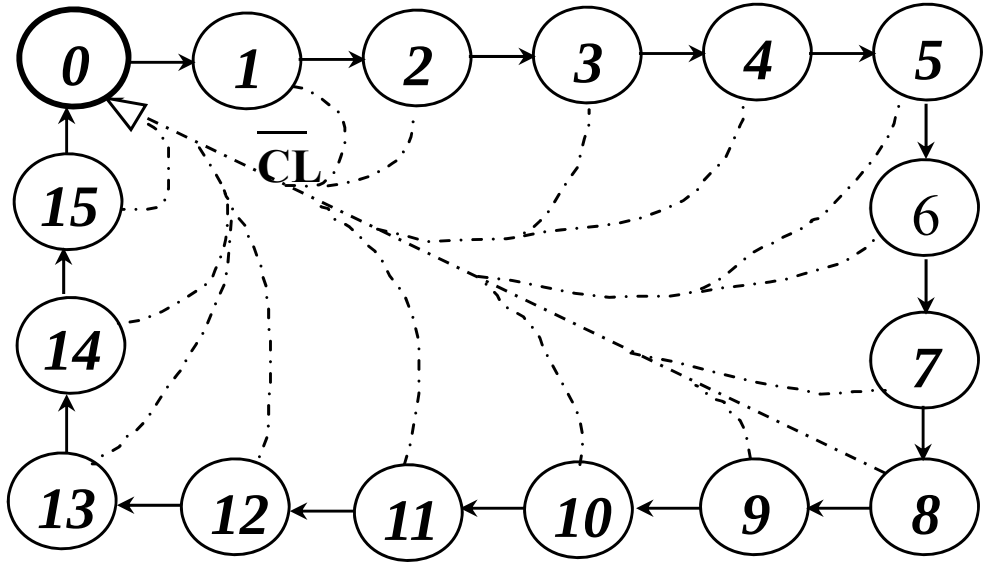
2. Synchronous counters
   – All flip flops are triggered by a clock signal at the same time

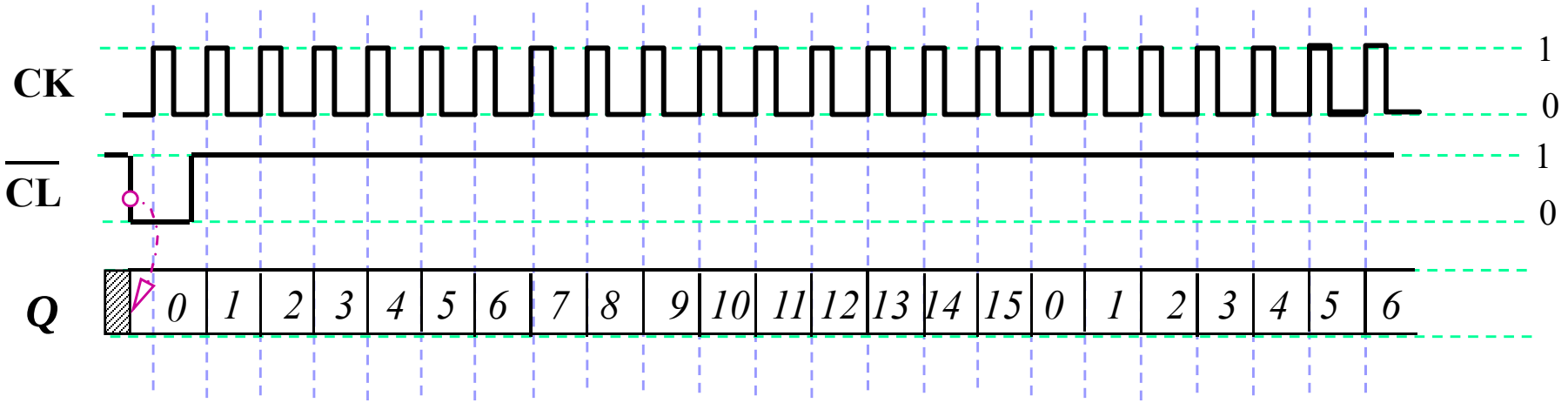
Modulo 16 Synchronous Counter using *D* Flip-Flops

The number of unique states that a counter may go through before the count sequence repeats itself is the **modulus** (or MOD) of the counter.



$$Q = \sum_{i=0}^{3} Qi \cdot 2^i$$





Modulo 16 Synchronous Counter

DECIMAL STATE	Present STATE OF THE COUNTER				The next state = FLIP FLOP INPUTS			
Q	Q3	Q2	Q1	Q0	D3	D2	D1	D0
0	0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	1	0
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	1	0	0
4	0	1	0	0	0	1	0	1
5	0	1	0	1	0	1	1	0
6	0	1	1	0	0	1	1	1
7	0	1	1	1	1	0	0	0
8	1	0	0	0	1	0	0	1
9	1	0	0	1	1	0	1	0
10	1	0	1	0	1	0	1	1
11	1	0	1	1	1	1	0	0
12	1	1	0	0	1	1	0	1
13	1	1	0	1	1	1	1	0
14	1	1	1	0	1	1	1	1
15	1	1	1	1	0	0	0	0

Using D flip-flops has the distinct advantage of a straightforward definition of the **flip-flop inputs: the current state of these inputs is the next state of the counter** $Q_{n+1} = D_n$. The logic equations for all four flip-flop inputs **D3, D2, D1**, and **D0** are derived from this truth table as functions of the current states of the counter's flip-flops: **Q3, Q2, Q1**, and **Q0**. Karnaugh maps can be used to simplify these equations.

Q1 Q0 \ Q3 Q2	00	01	11	10
00	*0*	*4*	*12*	*8*
01	*1*	*5*	*13*	*9*
11	*3*	*7*	*15*	*11*
10	*2*	*6*	*14*	*10*

D3

Q1 Q0 \ Q3 Q2	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	0	1	0	1
10	0	0	1	1

D2

Q1 Q0 \ Q3 Q2	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	1	0	0	1
10	0	1	1	0

D1

Q1 Q0 \ Q3 Q2	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

D0

Q1 Q0 \ Q3 Q2	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	0	0	0	0
10	1	1	1	1

Modulo 16 Synchronous Counter

Excitation Equations $D^n = Q^{n+1}$

Q3 Q2 \ Q1 Q0	00	01	11	10
D3				
00	0	0	1	1
01	0	0	1	1
11	0	1	0	1
10	0	0	1	1

$D3 = Q3\cdot\overline{Q2} + Q3\cdot\overline{Q1} + Q3\cdot\overline{Q0} + \overline{Q3}\cdot Q2\cdot Q1\cdot Q0$

Q3 Q2 \ Q1 Q0	00	01	11	10
D2				
00	0	1	1	0
01	0	1	1	0
11	1	0	0	1
10	0	1	1	0

$D2 = Q2\cdot\overline{Q0} + Q2\cdot\overline{Q1} + \overline{Q2}\cdot Q1\cdot Q0$

Q3 Q2 \ Q1 Q0	00	01	11	10
D1				
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

$D1 = \overline{Q1}\cdot Q0 + Q1\cdot\overline{Q0}$

Q3 Q2 \ Q1 Q0	00	01	11	10
D0				
00	1	1	1	1
01	0	0	0	0
11	0	0	0	0
10	1	1	1	1

$D0 = \overline{Q0}$

Modulo 16 Synchronous Counter Implementation

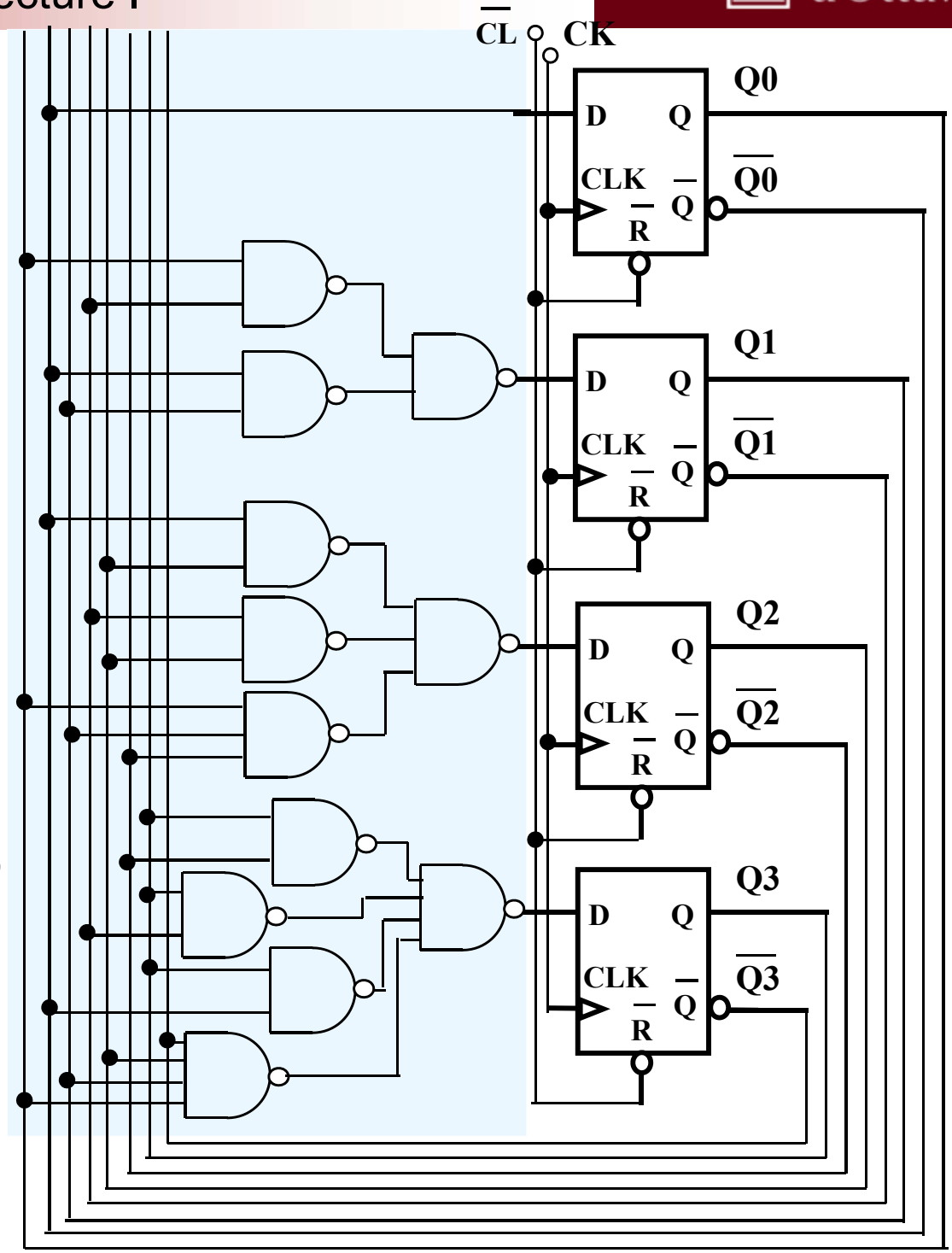
$$D^n = Q^{n+1}$$

$$\mathbf{D0} = \overline{Q0}$$

$$\mathbf{D1} = \overline{Q1}\cdot Q0 + Q1\cdot \overline{Q0}$$

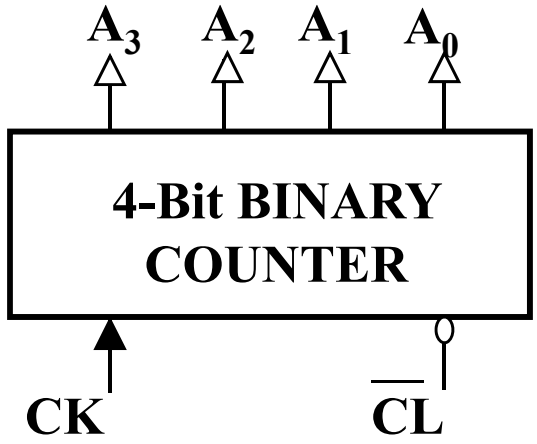
$$\mathbf{D2} = Q2\cdot \overline{Q0} + Q2\cdot \overline{Q1} + \overline{Q2}\cdot Q1\cdot Q0$$

$$\mathbf{D3} = Q3\cdot \overline{Q2} + Q3\cdot \overline{Q1} + Q3\cdot \overline{Q0} + \overline{Q3}\cdot Q2\cdot Q1\cdot Q0$$

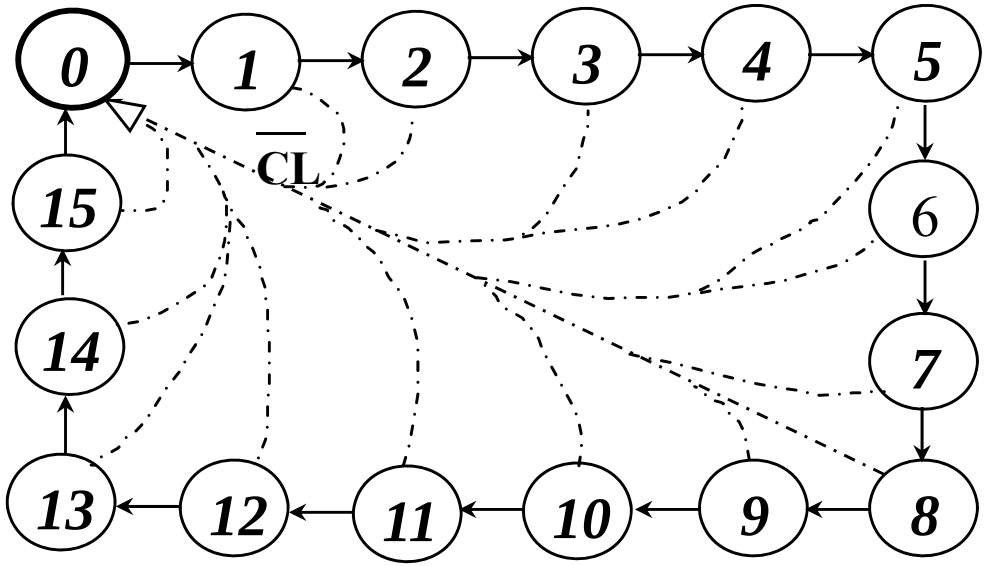


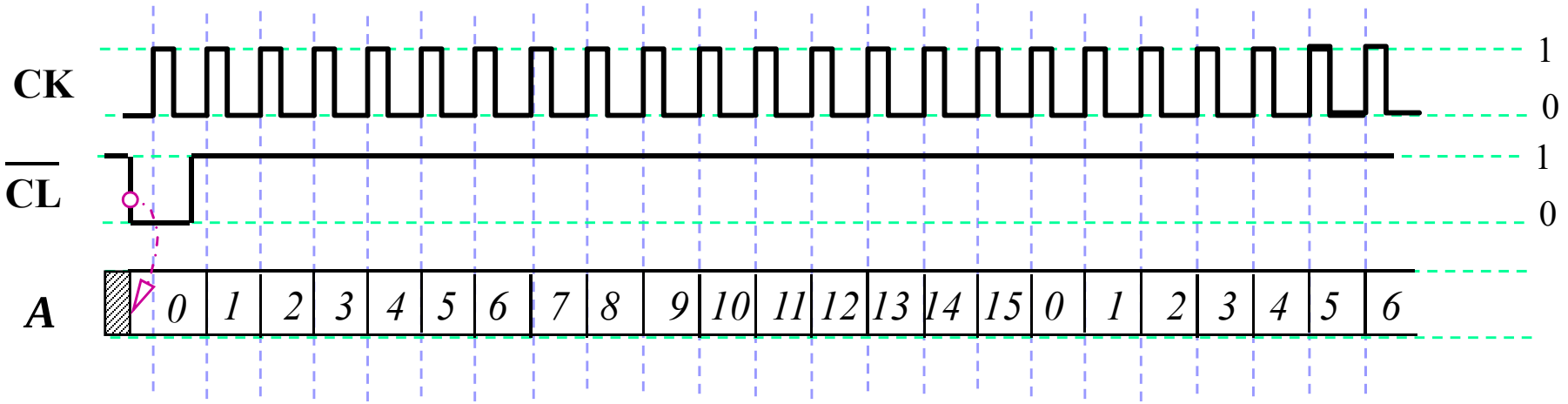
Binary Counter

Modulo 16 Synchronous Counter using *JK* Flip-Flops



$$A = \sum_{i=0}^{3} A_i \cdot 2^i$$





	Present state S^n				Next state S^{n+1}			
A	A_3	A_2	A_1	A_0	A_3^+	A_2^+	A_1^+	A_0^+
0	0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	1	0
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	1	0	0
4	0	1	0	0	0	1	0	1
5	0	1	0	1	0	1	1	0
6	0	1	1	0	0	1	1	1
7	0	1	1	1	1	0	0	0
8	1	0	0	0	1	0	0	1
9	1	0	0	1	1	0	1	0
10	1	0	1	0	1	0	1	1
11	1	0	1	1	1	1	0	0
12	1	1	0	0	1	1	0	1
13	1	1	0	1	1	1	1	0
14	1	1	1	0	1	1	1	1
15	1	1	1	1	0	0	0	0

"Common sense" design approach

We can notice from the State Table that:

- A_0 toggles its value at each clock pulse
  => **$J_0 = 1$; $K_0 = 1$.**
- A_1 toggles its value at time *T* only if at time (T - 1) A_0 is 1.

Present State			Next State	Flip-Flop Inputs	
A_2 (T − 1)	A_1 (T − 1)	A_0 (T − 1)	A_1 (T)	J_1	K_1
x	x	0	A_1 (t − 1)	0	0
x	x	1	$(A_1$ (t − 1))'	1	1

$J_1 = K_1 = A_0$ (T - 1)

- A_2 toggles its value at time T only if at time (T - 1) $A_1 = 1$ and $A_0 = 1$

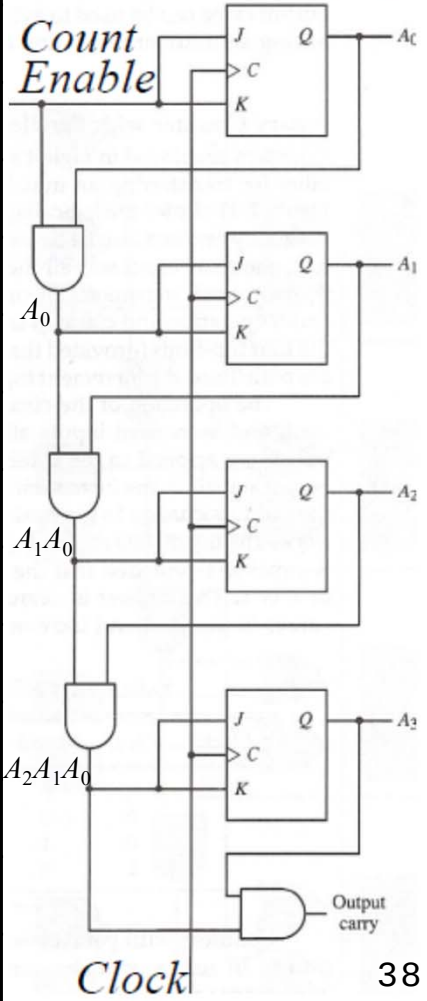
Present State			Next State	Flip-Flop Inputs
A_2 (T − 1)	A_1 (T − 1)	A_0 (T − 1)	A_2 (T)	J_2 K_2
x	0	0	A_2 (t − 1)	0 0
x	0	1	A_2 (t − 1)	0 0
x	1	0	A_2 (t − 1)	0 0
x	1	1	**$(A_2$ (t − 1))'**	1 1

$J_2 = K_2 = A_1$ (T - 1) A_0 (T - 1)

	Present state S^n				Next state S^{n+1}				A_3 input		A_2 input		A_1 input		A_0 input	
A	A_3	A_2	A_1	A_0	A_3^+	A_2^+	A_1^+	A_0^+	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	0	1	0	x	0	x	0	x	1	x
1	0	0	0	1	0	0	1	0	0	x	0	x	1	x	x	1
2	0	0	1	0	0	0	1	1	0	x	0	x	x	0	1	x
3	0	0	1	1	0	1	0	0	0	x	1	x	x	1	x	1
4	0	1	0	0	0	1	0	1	0	x	x	0	0	x	1	x
5	0	1	0	1	0	1	1	0	0	x	x	0	1	x	x	1
6	0	1	1	0	0	1	1	1	0	x	x	0	x	0	1	x
7	0	1	1	1	1	0	0	0	1	x	x	1	x	1	x	1
8	1	0	0	0	1	0	0	1	x	0	0	x	0	x	1	x
9	1	0	0	1	1	0	1	0	x	0	0	x	1	x	x	1
10	1	0	1	0	1	0	1	1	x	0	0	x	x	0	1	x
11	1	0	1	1	1	1	0	0	x	0	1	x	x	1	x	1
12	1	1	0	0	1	1	0	1	x	0	x	0	0	x	1	x
13	1	1	0	1	1	1	1	0	x	0	x	0	1	x	x	1
14	1	1	1	0	1	1	1	1	x	0	x	0	x	0	1	x
15	1	1	1	1	0	0	0	0	x	1	x	1	x	1	x	1

Use the "Common sense" design approach -> **see previous slide**

$$J_0 = K_0 = 1$$
$$J_1 = K_1 = A_0$$
$$J_2 = K_2 = A_0A_1$$
$$J_3 = K_3 = A_0A_1A_2$$



Or use the "the receipe"

	Present state S^n				Next state S^{n+1}				A_3 input		A_2 input		A_1 input		A_0 input	
A	A_3	A_2	A_1	A_0	A_3^+	A_2^+	A_1^+	A_0^+	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	0	1	0	x						
1	0	0	0	1	0	0	1	0	0	x						
2	0	0	1	0	0	0	1	1	0	x						
3	0	0	1	1	0	1	0	0	0	x						
4	0	1	0	0	0	1	0	1	0	x						
5	0	1	0	1	0	1	1	0	0	x						
6	0	1	1	0	0	1	1	1	0	x						
7	0	1	1	1	1	0	0	0	1	x						
8	1	0	0	0	1	0	0	1	x	0						
9	1	0	0	1	1	0	1	0	x	0						
10	1	0	1	0	1	0	1	1	x	0						
11	1	0	1	1	1	1	0	0	x	0						
12	1	1	0	0	1	1	0	1	x	0						
13	1	1	0	1	1	1	1	0	x	0						
14	1	1	1	0	1	1	1	1	x	0						
15	1	1	1	1	0	0	0	0	x	1						

	Present state S^n				Next state S^{n+1}				A_3 input		A_2 input		A_1 input		A_0 input	
A	A_3	A_2	A_1	A_0	A_3^+	A_2^+	A_1^+	A_0^+	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	0	1	0	x	0	x				
1	0	0	0	1	0	0	1	0	0	x	0	x				
2	0	0	1	0	0	0	1	1	0	x	0	x				
3	0	0	1	1	0	1	0	0	0	x	1	x				
4	0	1	0	0	0	1	0	1	0	x	x	0				
5	0	1	0	1	0	1	1	0	0	x	x	0				
6	0	1	1	0	0	1	1	1	0	x	x	0				
7	0	1	1	1	1	0	0	0	1	x	x	1				
8	1	0	0	0	1	0	0	1	x	0	0	x				
9	1	0	0	1	1	0	1	0	x	0	0	x				
10	1	0	1	0	1	0	1	1	x	0	0	x				
11	1	0	1	1	1	1	0	0	x	0	1	x				
12	1	1	0	0	1	1	0	1	x	0	x	0				
13	1	1	0	1	1	1	1	0	x	0	x	0				
14	1	1	1	0	1	1	1	1	x	0	x	0				
15	1	1	1	1	0	0	0	0	x	1	x	1				

Or use the "the receipe"

	Present state S^n				Next state S^{n+1}				A_3 input		A_2 input		A_1 input		A_0 input	
A	A_3	A_2	A_1	A_0	A_3^+	A_2^+	A_1^+	A_0^+	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	0	1	0	x	0	x	0	x		
1	0	0	0	1	0	0	1	0	0	x	0	x	1	x		
2	0	0	1	0	0	0	1	1	0	x	0	x	x	0		
3	0	0	1	1	0	1	0	0	0	x	1	x	x	1		
4	0	1	0	0	0	1	0	1	0	x	x	0	0	x		
5	0	1	0	1	0	1	1	0	0	x	x	0	1	x		
6	0	1	1	0	0	1	1	1	0	x	x	0	x	0		
7	0	1	1	1	1	0	0	0	1	x	x	1	x	1		
8	1	0	0	0	1	0	0	1	x	0	0	x	0	x		
9	1	0	0	1	1	0	1	0	x	0	0	x	1	x		
10	1	0	1	0	1	0	1	1	x	0	0	x	x	0		
11	1	0	1	1	1	1	0	0	x	0	1	x	x	1		
12	1	1	0	0	1	1	0	1	x	0	x	0	0	x		
13	1	1	0	1	1	1	1	0	x	0	x	0	1	x		
14	1	1	1	0	1	1	1	1	x	0	x	0	x	0		
15	1	1	1	1	0	0	0	0	x	1	x	1	x	1		

Or use the "the receipe"

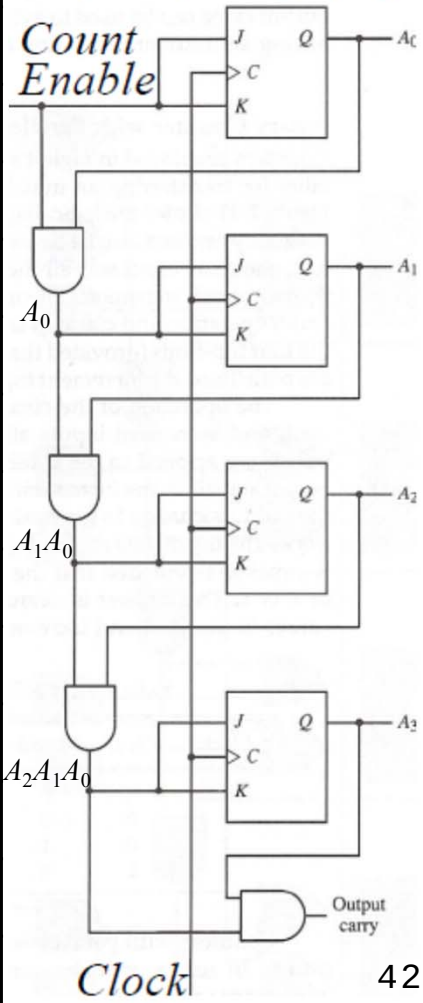
	Present state S^n				Next state S^{n+1}				A_3 input		A_2 input		A_1 input		A_0 input	
A	A_3	A_2	A_1	A_0	A_3^+	A_2^+	A_1^+	A_0^+	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	0	1	0	x	0	x	0	x	1	x
1	0	0	0	1	0	0	1	0	0	x	0	x	1	x	x	1
2	0	0	1	0	0	0	1	1	0	x	0	x	x	0	1	x
3	0	0	1	1	0	1	0	0	0	x	1	x	x	1	x	1
4	0	1	0	0	0	1	0	1	0	x	x	0	0	x	1	x
5	0	1	0	1	0	1	1	0	0	x	x	0	1	x	x	1
6	0	1	1	0	0	1	1	1	0	x	x	0	x	0	1	x
7	0	1	1	1	1	0	0	0	1	x	x	1	x	1	x	1
8	1	0	0	0	1	0	0	1	x	0	0	x	0	x	1	x
9	1	0	0	1	1	0	1	0	x	0	0	x	1	x	x	1
10	1	0	1	0	1	0	1	1	x	0	0	x	x	0	1	x
11	1	0	1	1	1	1	0	0	x	0	1	x	x	1	x	1
12	1	1	0	0	1	1	0	1	x	0	x	0	0	x	1	x
13	1	1	0	1	1	1	1	0	x	0	x	0	1	x	x	1
14	1	1	1	0	1	1	1	1	x	0	x	0	x	0	1	x
15	1	1	1	1	0	0	0	0	x	1	x	1	x	1	x	1

$J_0 = K_0 = 1$

$J_1 = K_1 = A_0$

$J_2 = K_2 = A_0A_1$

$J_3 = K_3 = A_0A_1A_2$



Outline

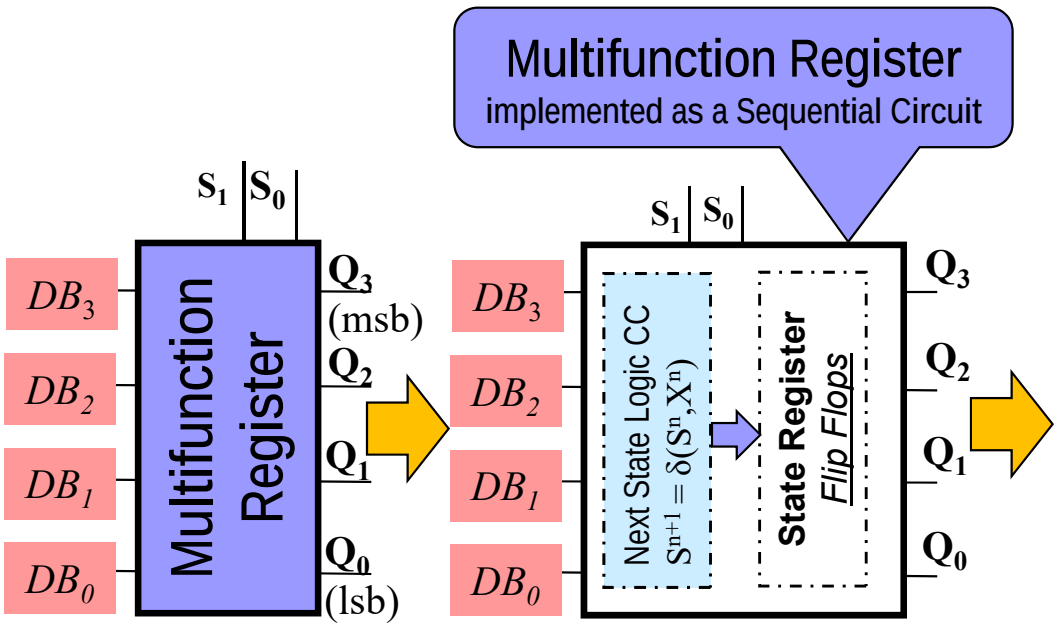
- Decoders, Multiplexers
- Registers + Common sense design strategies
  - Register with Parallel Load
  - Shift Registers
  - Bidirectional Shift Register with Parallel Load
- Binary Counters
  - **Binary Counter with Parallel Load**

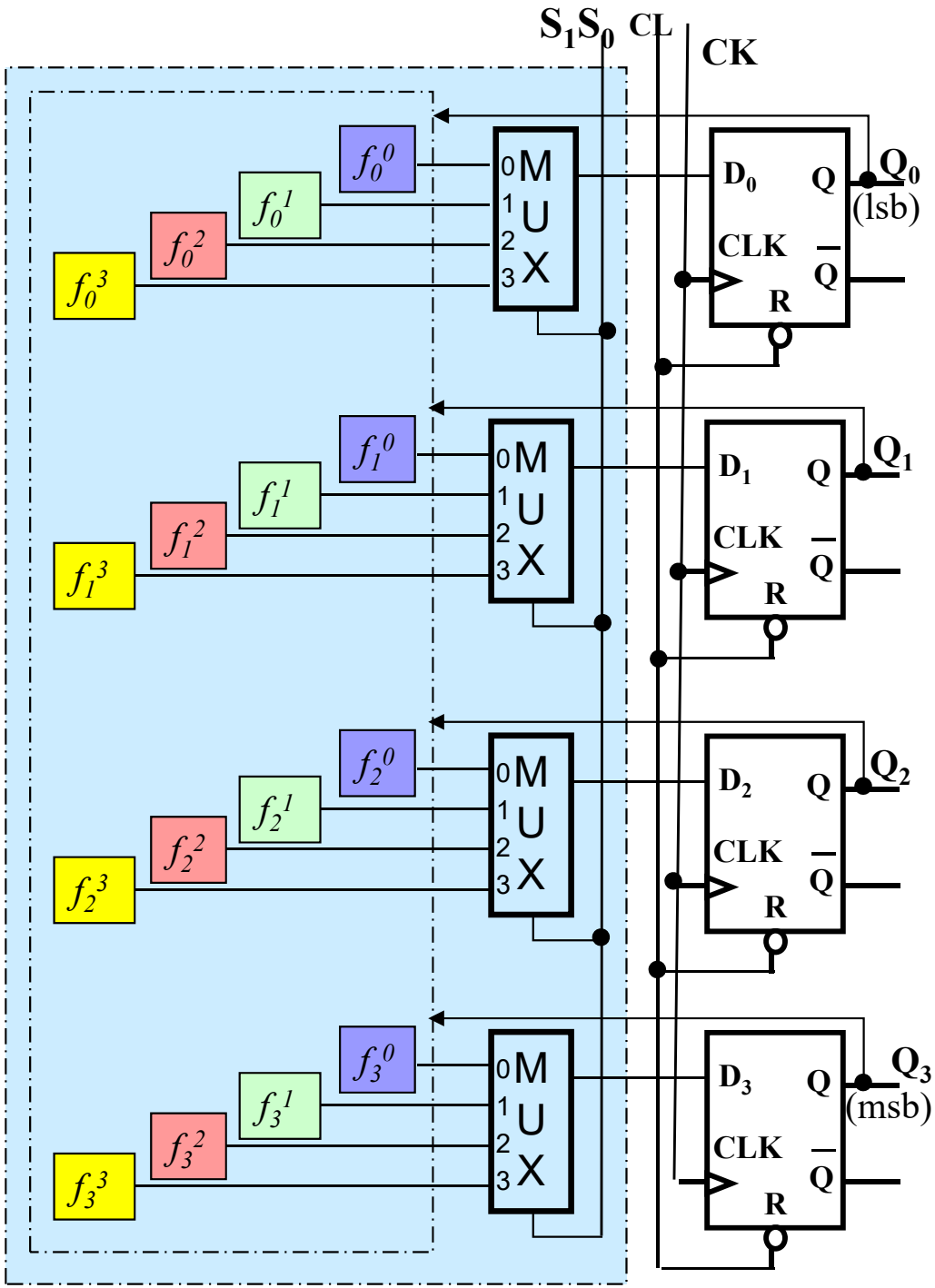
Top-down design

Multifunction Register
with D-Flip Flops

	S_1	S_0	Q_k^{n+1}	*Function*
(f_0)	0	0	Q_k^n	*Store*
(f_1)	0	1	N_k	*Count up*
(f_2)	1	0	DB_k	*Load*
(f_3)	1	1	0	*Clear*

$k = \{0,1,2,3\}$





Multifunction Register

	S_1 S_0	$D_k = Q_k^{n+1}$	*Function*
(f^0)	0 0	Q_k^n	*Store*
(f^1)	0 1	N_k	*Count up*
(f^2)	1 0	DB_k	*Load*
(f^3)	1 1	0	*Clear*

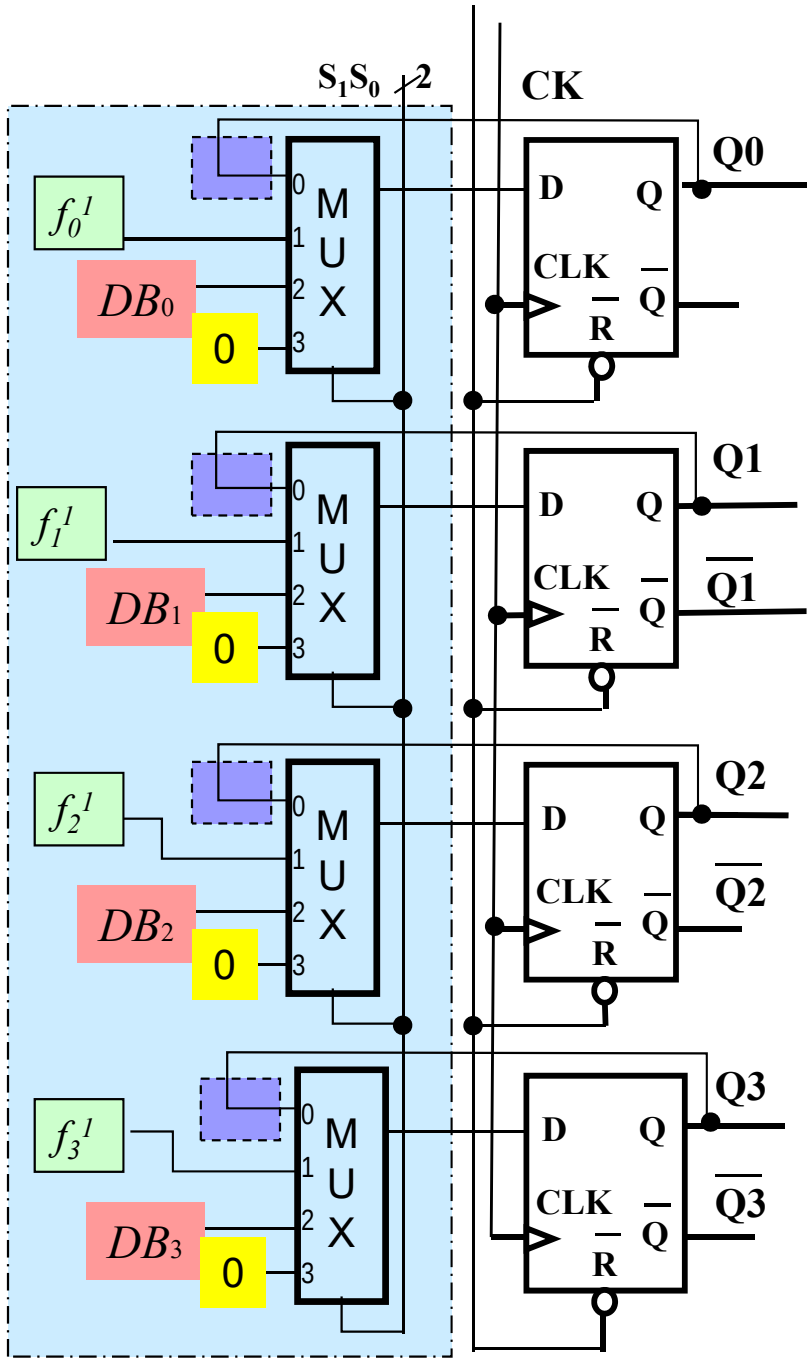
Use the "Common sense" design approach, (where obvious) or:

D- FF's excitation equation: $D^n = Q^{n+1}$ (1)

Store (f^0): $Q^{n+1} = Q^n$ (2) to be implemented with D-FF
From (1), (2) => $D^n = Q^{n+1} = Q^n$ i.e., $D_k = Q_k$, $k = \{0,1,2,3\}$

Clear (f^3): $Q^{n+1} = 0$ (3) with D-FF
From (1), (3) => $D^n = 0$ i.e., $D_k = 0$, $k = \{0,1,2,3\}$

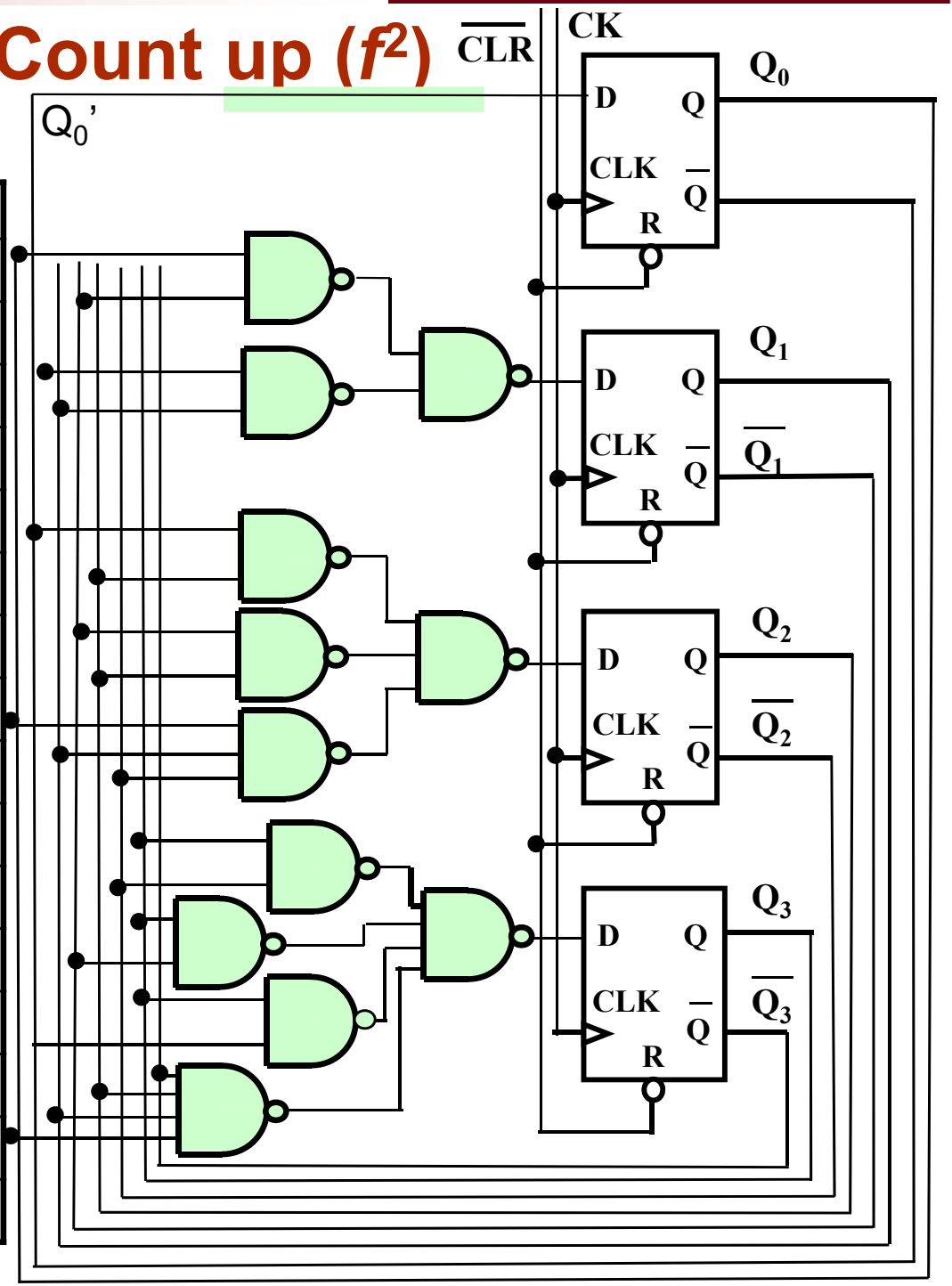
Load (f^2): $Q^{n+1} = DB^n$ (4) with D-FF
From (1), (4) => $D^n = DB^n$, i.e., $D_k = DB_k$, $k = \{0,1,2,3\}$



uOttawa

Multifunction Register - Count up (f^2)

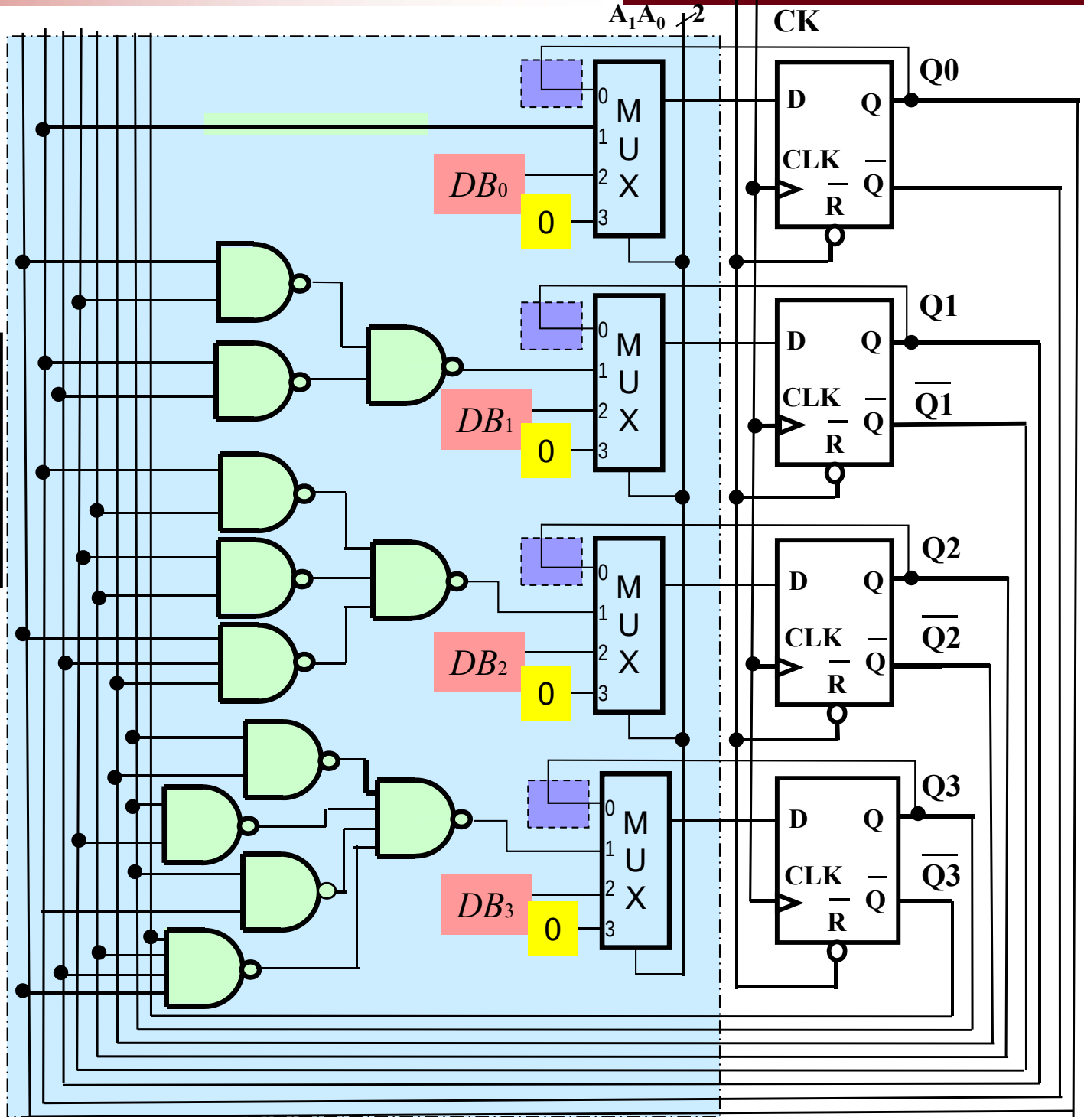
Q_3	Q_2	Q_1	Q_0	Q_3^+	Q_2^+	Q_1^+	Q_0^+	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	1	0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	1	0	0	1	1
0	0	1	1	0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	1	0	1	0	1
0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	1	0	1	1	1
0	1	1	1	1	0	0	0	1	0	0	0
1	0	0	0	1	0	0	1	1	0	0	1
1	0	0	1	1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	1	1	0	1	1
1	0	1	1	1	1	0	0	1	1	0	0
1	1	0	0	1	1	0	1	1	1	0	1
1	1	0	1	1	1	1	0	1	1	1	0
1	1	1	0	1	1	1	1	1	1	1	1
1	1	1	1	0	0	0	0	0	0	0	0



uOttawa

Multifunction Register Implementation

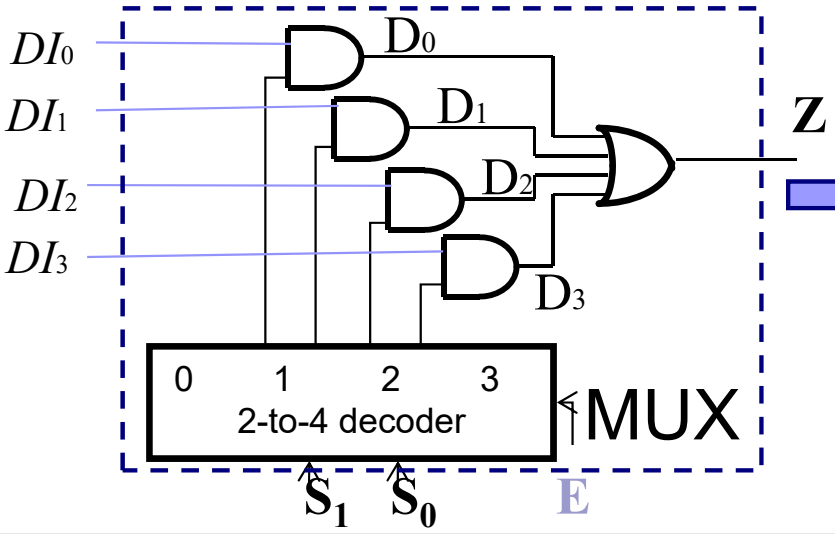
	A_1	A_0	$D_k = Q_k^{n+1}$	*Function*
(f_0)	0	0	Q_k^n	*Store*
(f_1)	0	1	N_k	*Count up*
(f_2)	1	0	DB_k	*Load*
(f_3)	1	1	0	*Clear*



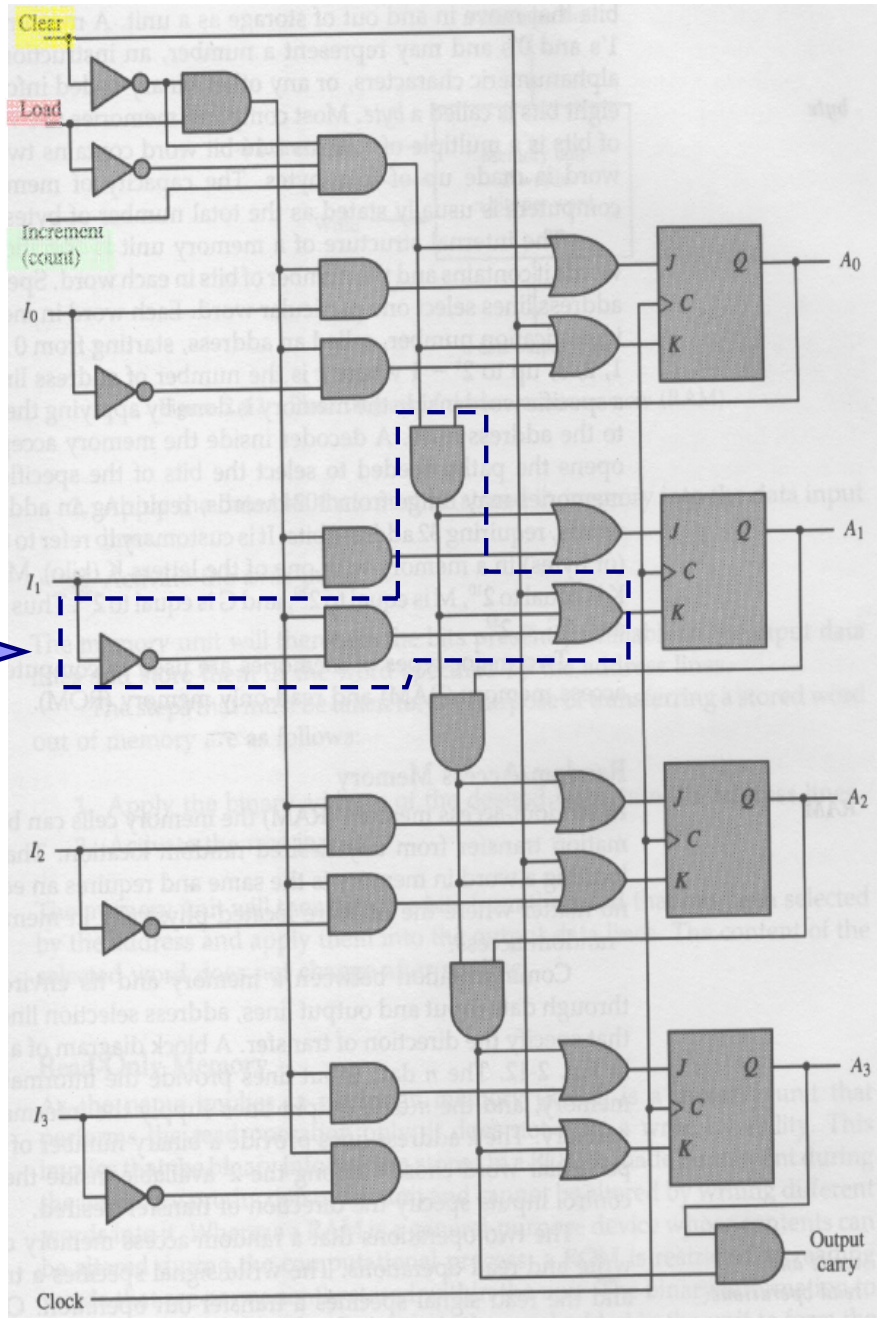
4-bit **Binary Counter with Parallel Load & Synchronous Clear** with JK FF

	A_1	A_0	Q_k^{n+1}	**Function**
(f_0)	0	0	Q_k^n	*Store = No change*
(f_1)	0	1	N_k	*Increment (Count up)*
(f_2)	1	0	DB_k	*Load*
(f_3)	1	1	0	*Clear*

$k = \{0,1,2,3\}$



Clock	Clear	Load	Increment	Operation
↑	0	0	0	No change
↑	0	0	1	Increment count by 1
↑	0	1	x	Load inputs
↑	1	x	x	Clear outputs to 0

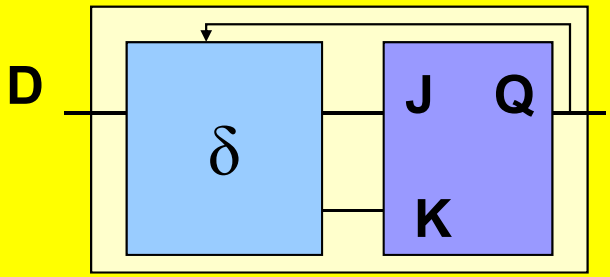


Design D-Latch using JK FF

START HERE

Given: D-Latch Characteristic Table

D	Q^n	Q^{n+1}
0	0	0
0	1	0
1	0	1
1	1	1



1

Step 1: Initial State table

JK Excitation Table

Q^t	$Q^{t+\Delta t}$	J^t	K^t
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

I

II

State table of Sequential Circuit to be Designed

D	Q^n	Q^{n+1}	J	K
0	0	0	0	x
0	1	0	x	1
1	0	1	1	x
1	1	1	x	0

Excitation table for Sequential Circuit to be Designed

Step 3

4

4

J

D \ Q^n	0	1
0	0 0	0 1
1	1 2	x 3

D

K

D \ Q^n	0	1
0	x 0	1 1
1	x 2	0 3

$\bar{D}$

4

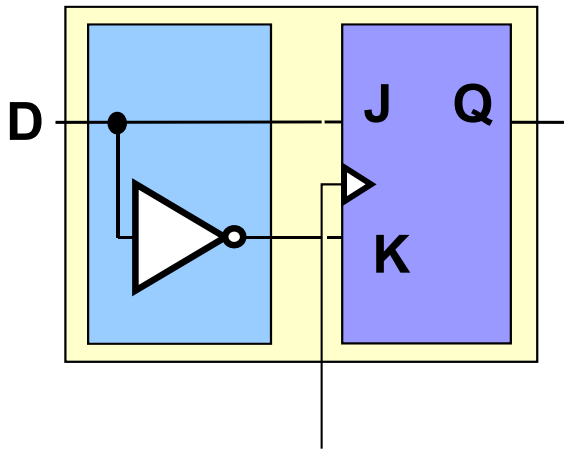
Step 4: SR input equations

$J = D$

$K = \bar{D}$

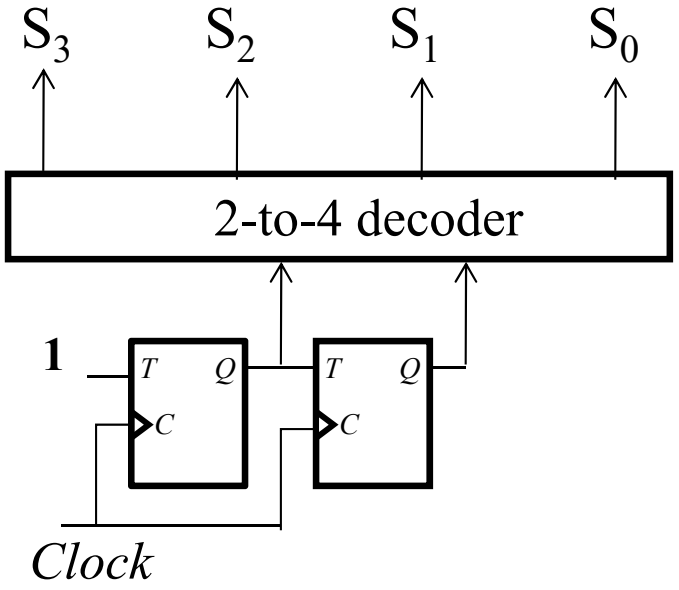
5

Step 5

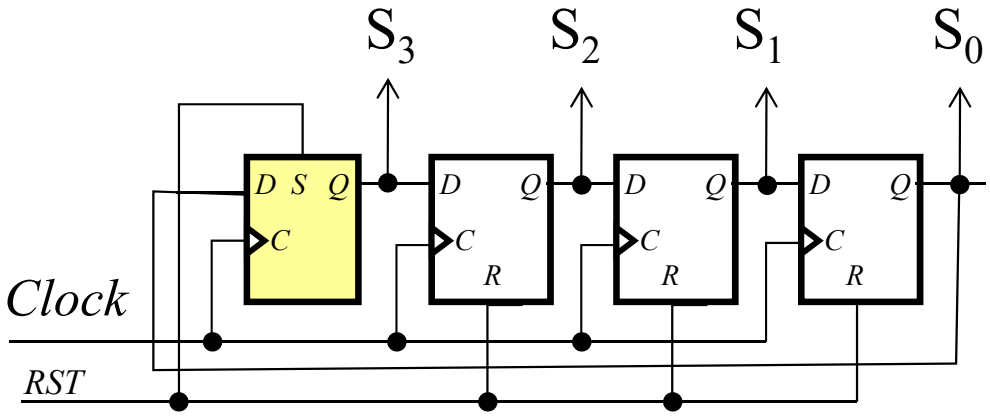


State Encoding
mod 4 counter

Full Encoding:
BINARY COUNTER



1-hot Encoding:
RING COUNTER



Memory Unit

uOttawa

L'Université canadienne
Canada's university

Dr. Voicu Groza

SITE Hall, Room 5017
562 5800 ext. 2159
Groza@EECS.uOttawa.ca

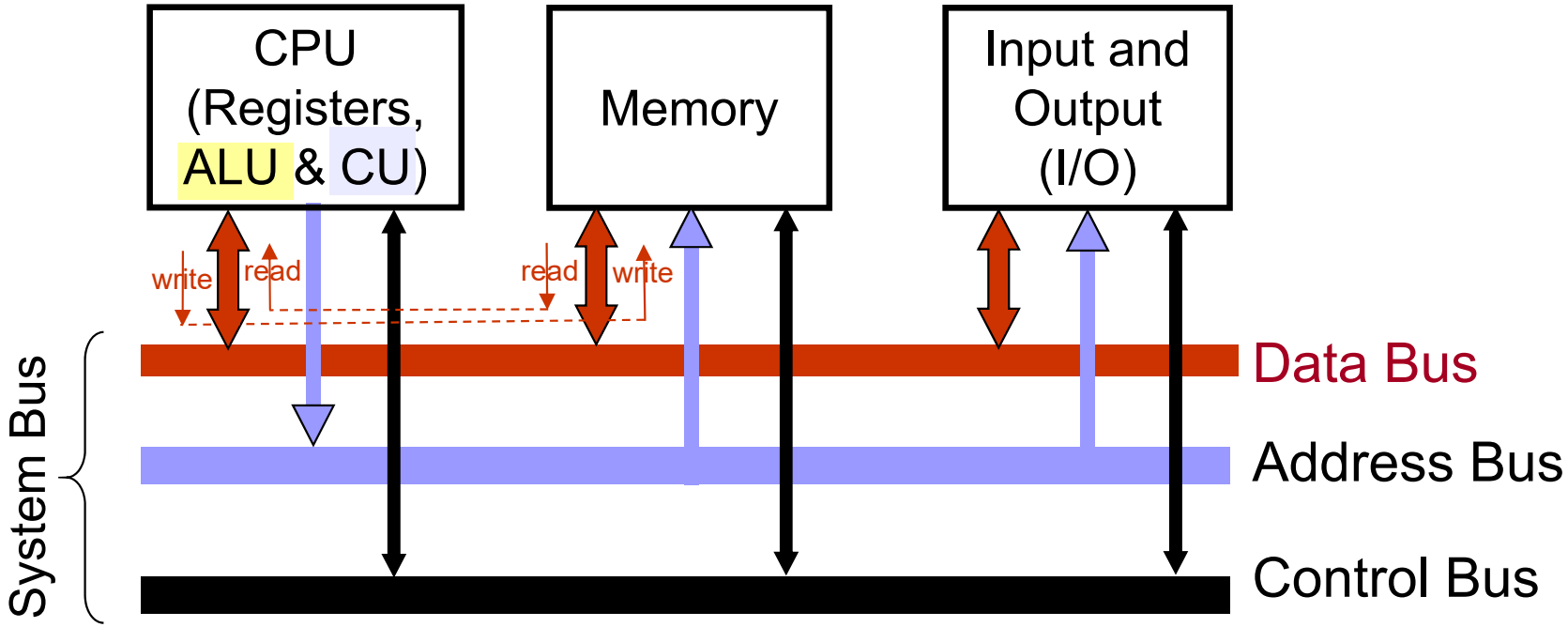
Université d'Ottawa | University of Ottawa

www.uOttawa.ca

Outline

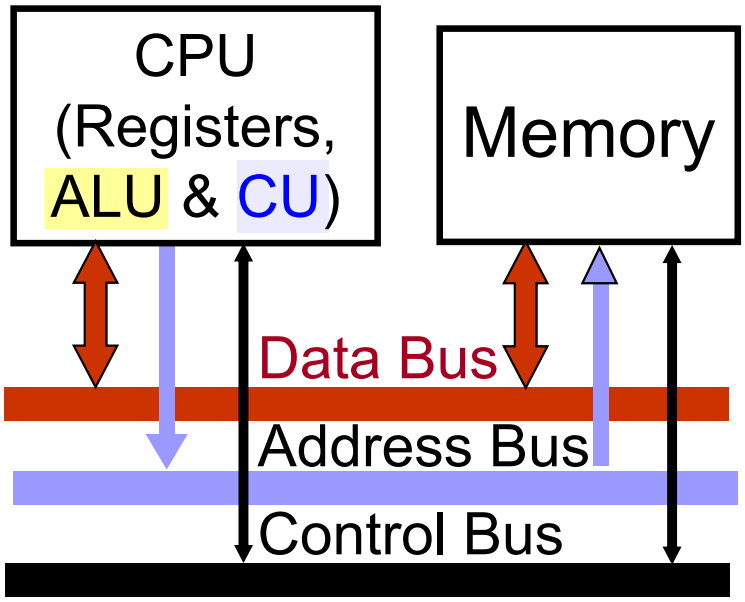
- The System Bus Model of a Computer
- Memory general Characteristics
- RAM (Random Access Memory)
- ROM (Read Only Memory)
- Memory design
- Memory hierarchy

The System Bus Model of a Computer

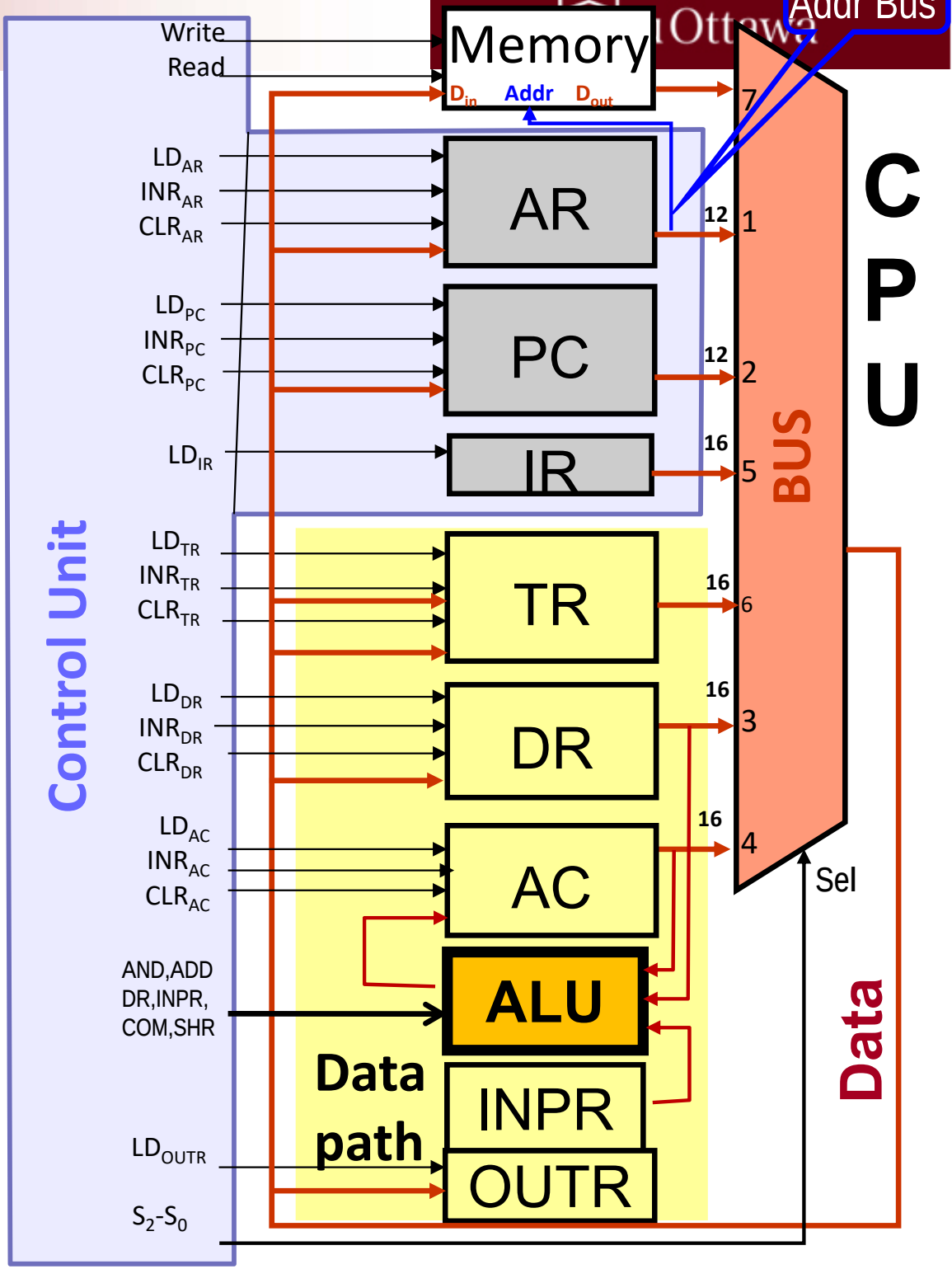


- The bus connects the CPU with other parts and reduces connections
- CPU puts address on the Bus and memory or I/O block receive address.
- Each instruction is **fetched** into the CPU from memory (read sequentially, i.e., one instruction at a time) and then **executed** mostly by the ALU.
- **C**ontrol **U**nit manages transfer of instructions & data through the Control Bus
- The output is displayed on output device such as CRT

BASIC COMPUTER

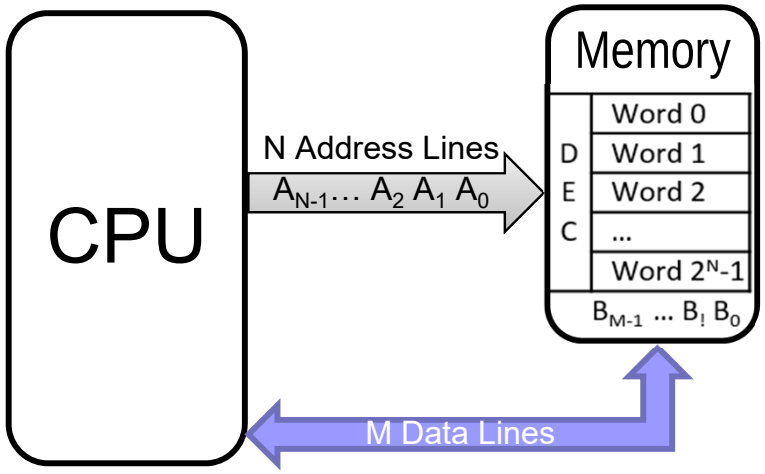


Most registers of the Basic Computer are implemented on the basis of the logic diagram of the multifunctional registers designed before.



Memory

- A computer has a Central Processing Unit (CPU) and a memory.
- The memory stores the data and the CPU can read (extract or fetch) the data from computer memory for processing, or, it can write (store) the data onto the computer memory.
- The memory can be seen as a set of registers of the same length that store binary data, called “words”. Each memory location (i.e., such a register) stores a word and has an address.



- The computer uses the address lines to locate the specific data word in the computer memory. Also, it uses the data lines to write a word into the memory location specified by the address lines, or to read a stored word from a memory location, also specified by the address lines, as shown in the figure.

- It can be noted from the figure above that the address lines run from CPU to memory, i.e., the address lines are unidirectional, and the data lines are bidirectional, i.e. CPU uses the data lines to both read from memory, or to write the data onto the memory.

Memory Unit

- Memories are collection of *registers* together with associated circuits needed to transfer information in and out of storage.
- The memory stores binary information in groups of bits called ***words***.
- Each *memory location* (register) holds one "word" that
  - can be read or written into the memory
  - is assigned a **unique** identification number called an **address**
    - addresses are represented as binary vectors, consecutively numbered
    - if *address* is encoded with *k* bits, memory can have a capacity of 2^k words

Address K bits	Location= M bit word
0000	WORD 0
0001	WORD 1
0010	WORD 2
0011	WORD 3
…	…
$(2^k-1)_2$	WORD 2^k-1

- *Data* is transferred to/from memory in multiples of words; i.e., it is NOT possible to read or write more or less bits than the size of the word
- One can access memory using only one address at a time, while simultaneous readings and writings are not possible
- A word in memory may contain any type of binary information that fits its size (e.g., a number, an instruction code, a memory address, or one or more alphanumeric characters)
- The internal structure of a memory is specified by the number of words it contains and the number of bits in each word.

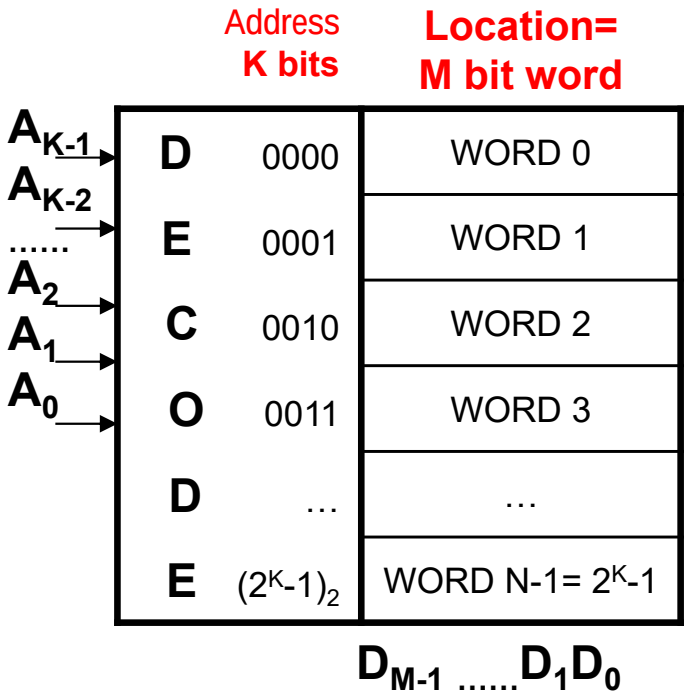
Memory Capacity

- Capacity = memory size = amount of data (measured in bits, bytes, words) stored in a memory.

- The memory size is generally specified as:

N x M

where,
  - N is the number of words stored in memory
  - M is the length (number of bits) of each word
- K address lines, with 2^K = N, allows accessing any of the N words
- For example, a memory chip of capacity 1024 x 8 means that the memory is capable of storing 1024 words where each word is of 8 bits or 1 byte.



- Further, in order to address 1024 x 8 memory words, 10 address lines are required (2^{10} = 1024)

Data

Bit	0
Nibble	0110
Byte	10110000
16-bit word (halfword)	11001001 01000110
32-bit word	10110100 00110101 10011001 01011000
64-bit word (double)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101
128-bit word (quad)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101 00001011 10100110 11110010 11100110 10100100 01000100 10100101 01010001

- A byte is a group of 8 bits ⇒ 16 bits = 2 bytes 32 bits = 4 bytes
- 1*K*(kilo) = 2^{10} 1*M*(mega) = 2^{20} 1*G*(giga) = 2^{30}
- ⇒ $64K = 2^6K = 2^6 \times 2^{10} = 2^{16}$

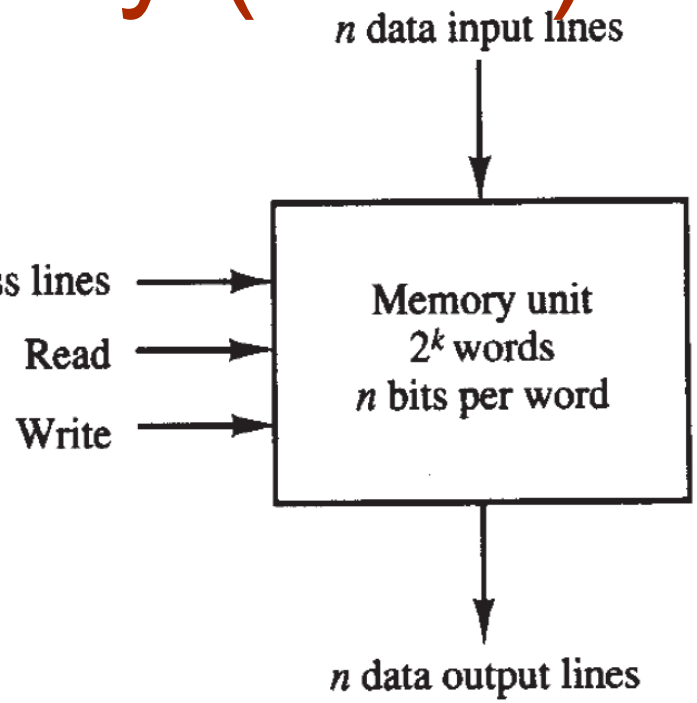
MEMORIES

Memory Technologies:

- Random Access Memory (RAM) - volatile
  - Static RAM (SRAM) – latches in an array of registers
  - Dynamic RAM (DRAM) – MOSFET “capacitors”
- Read Only Memory (ROM) – permanent (not programmable)
- Programmable ROM (PROM) – “fuse” principle
- EPROM – permanent, but erasable & reprogrammable
  - UV-EPROM
  - EEPROM
    - FLASH memory – erase all cells of the memory array at once
    - EEPROM – erase smaller blocks of the whole array

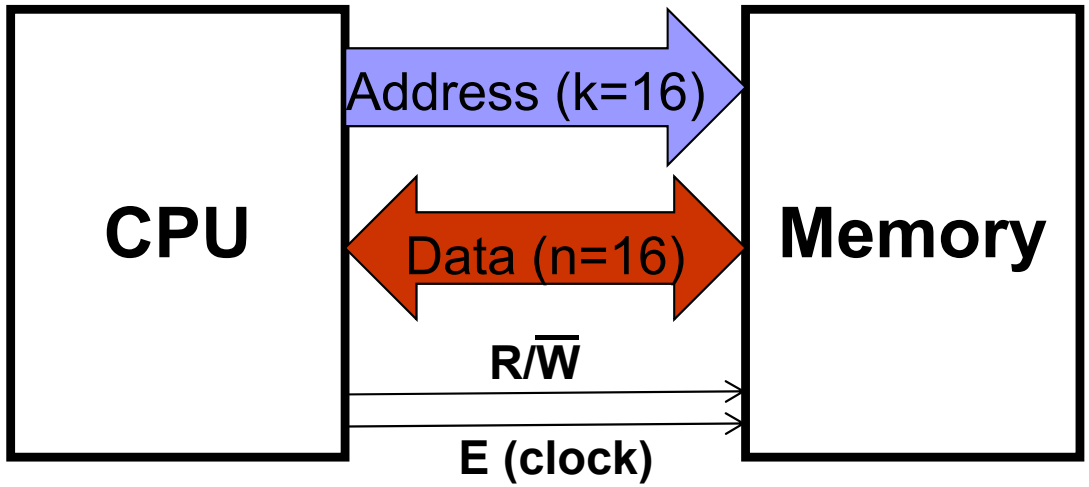
Random-Access Memory (RAM)

- Accessing the RAM is achieved through
  - data input and output lines,
  - address lines (to select which word in the memory is to be accessed), and
  - control lines that specify the type of access (Read, Write) and, as such, the direction of information transfer.



- The internal structure of a memory is specified by the number of words it contains and the number of bits in each word (n).
- **Memory Capacity** = the total amount of stored information that a storage device can hold. It is expressed as a quantity of bits, bytes or words
- For a memory unit with k address lines, there exists a maximum of 2^k different words (n-bit long) with addresses ranging from 0 to $2^k - 1$;
  - e.g. a memory with 16 address lines has a capacity of 2^{16} words ($2^6\ 2^{10} = 64$ k-words).
- The time duration for accessing any word in RAM is independent of where that word is physically located in the memory. Thus the word "random-access memory".

Address, Data and Control Buses

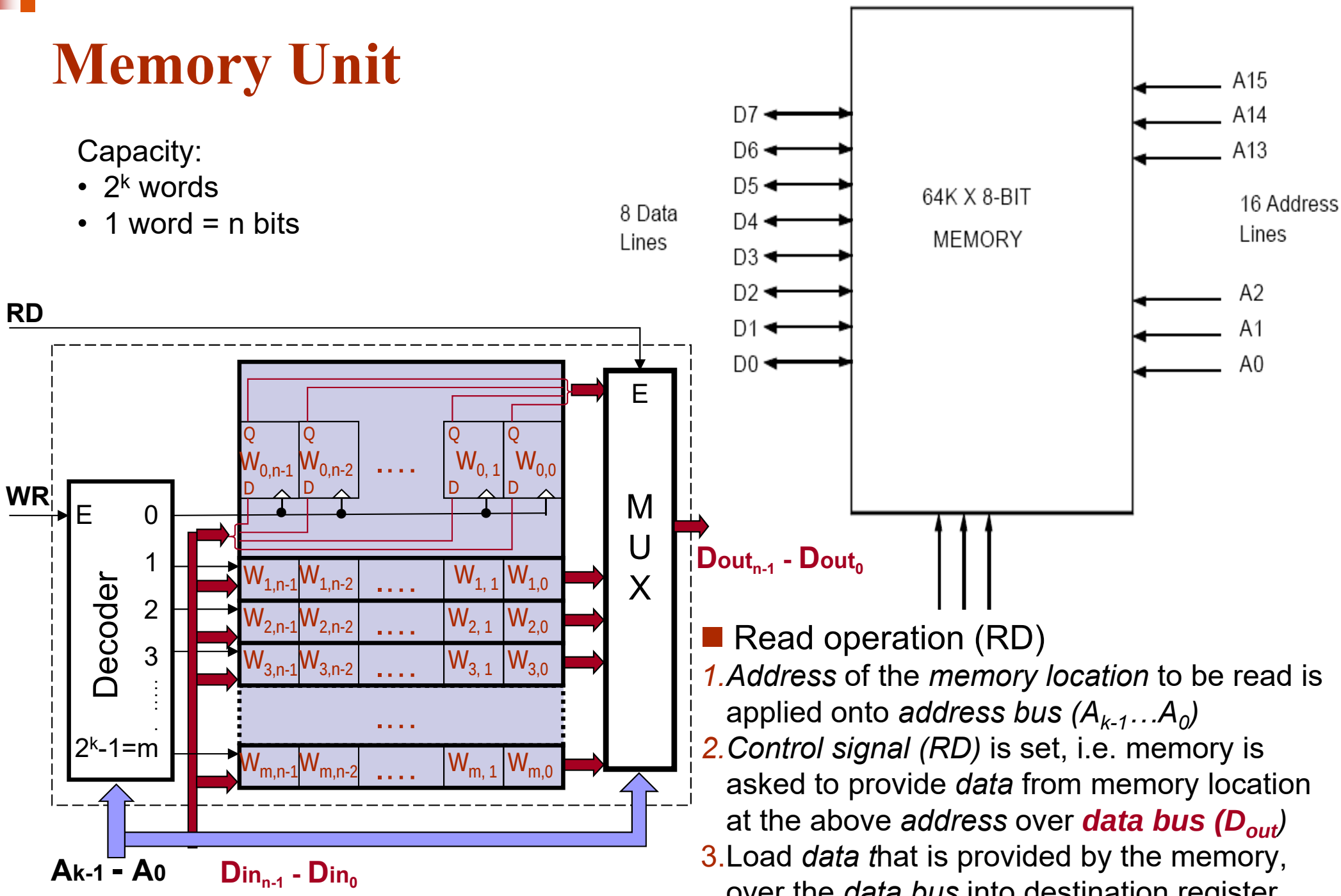


Example: if CPU has

- *k* = 16 bit **address** bus; i.e., CPU can access 2^{16} memory locations
- *n* = 16 bit **data bus**; i.e., CPU can access one word of16 bits (2 bytes) @ one memory location at a time,
- There are two types of operations a RAM can perform: **Read** and **Write**. These are specified by the control inputs (*Read* and *Write*)
- R/$\overline{W}$ tells memory if CPU is reading or writing
  - R/$\overline{W}$ high => Read
  - R/$\overline{W}$ low => Write

Memory Unit

Capacity:
- 2^k words
- 1 word = n bits



- Read operation (RD)

1. *Address* of the *memory location* to be read is applied onto *address bus* ($A_{k-1}…A_0$)
2. *Control signal (RD)* is set, i.e. memory is asked to provide *data* from memory location at the above *address* over ***data bus*** (D_{out})
3. Load *data t*hat is provided by the memory, over the *data bus* into destination register.

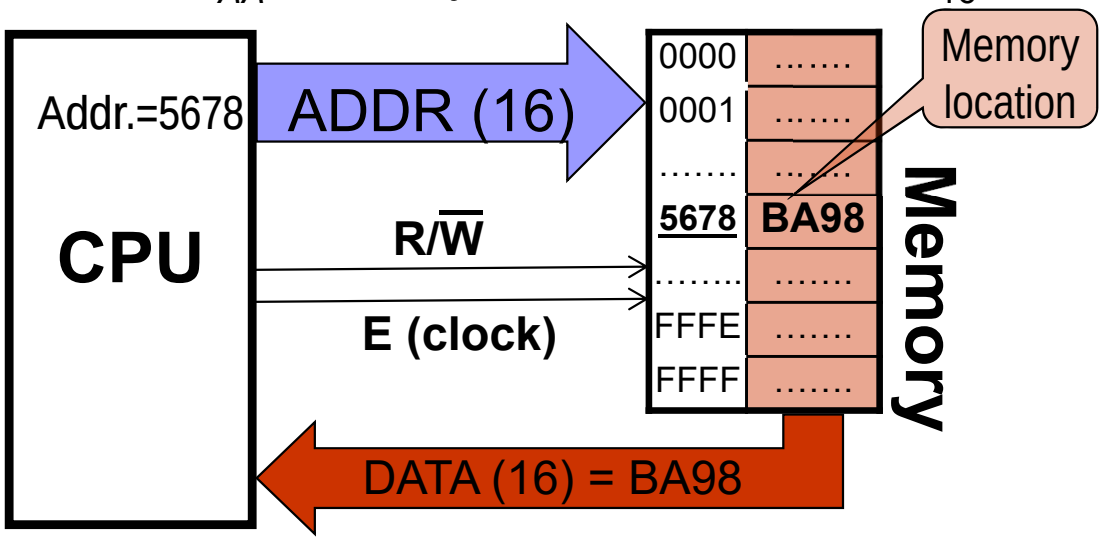
Read Cycle

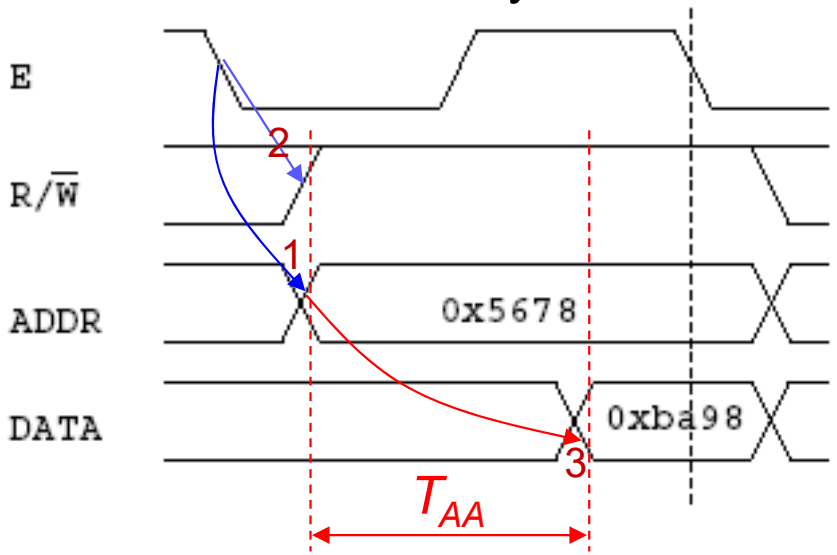
The steps to be taken to fetch a word out of RAM (read operation) are:

1. the *address* of the *memory location* which is to be read is applied onto the address lines ADDR.
2. the R/$\overline{W}$ (Read/Write) control line is brought to high (1) to indicate a **read**
3. after **access time** T_{AA}, the memory will put on the *Data bus* the **DATA** read from the memory location specified by ADDR.

Example: Assuming that the memory location with address 5678_{16} stores $BA98_{16}$, a CPU Read Cycle from that address will take place as follows:

1. CPU sends Address 5678_{16} (0101 0110 0111 1000) over 16 ADDR lines to Memory
2. CPU asserts R/$\overline{W}$=1 meaning it wants to read the content of location at address 5678_{16}
3. After T_{AA} memory sends to CPU $BA98_{16}$, i.e., the contents of memory location 5678_{16}





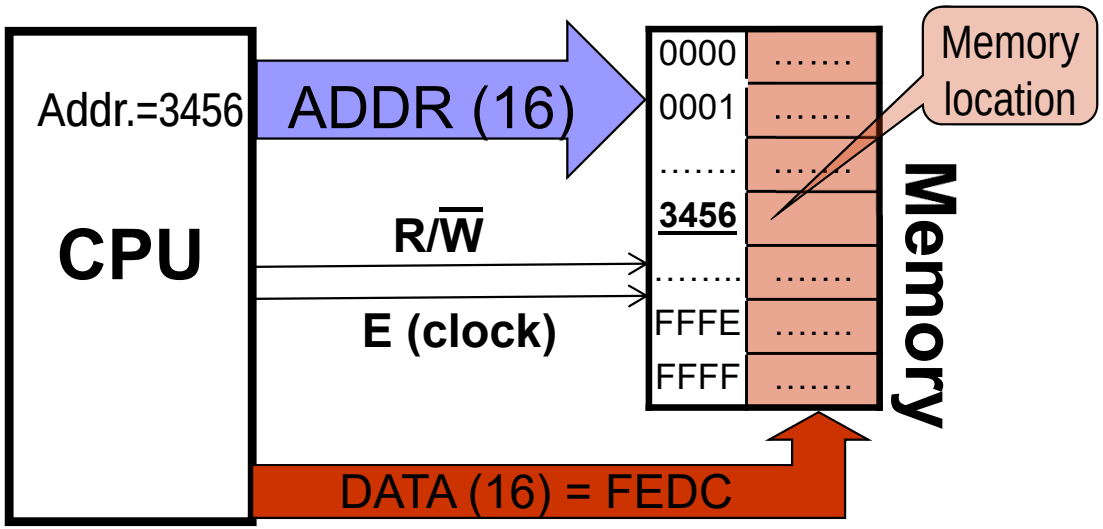
Write Cycle

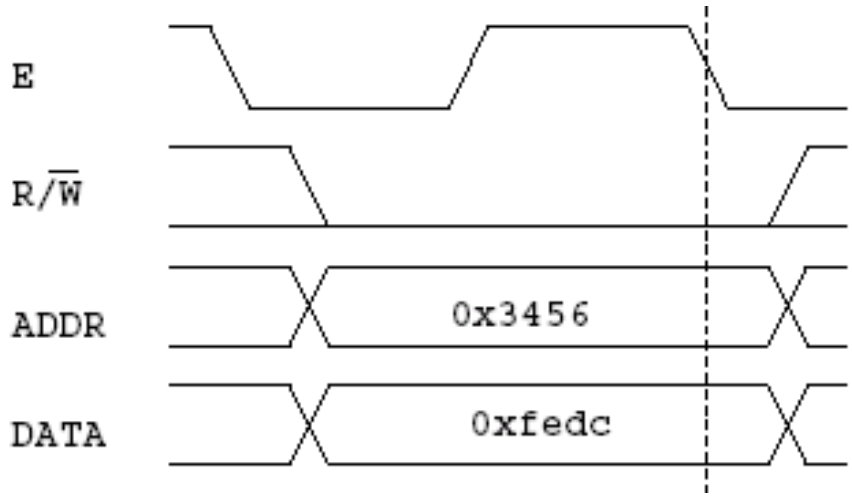
The steps to be taken to store a word into RAM (write operation) are the following:

1. CPU applies the binary vector address (**ADDR**) of the desired memory location into the address lines;
2. apply the **DATA** bits, that must be stored in memory, into the data input lines;
3. bring the R/$\overline{W}$ (Read/Write) line low to indicate a **write** (activate the *Write* control input).

In the below example, the CPU expects that the memory location at the given address will latch the data on the falling edge of the E-clock, at the latest.

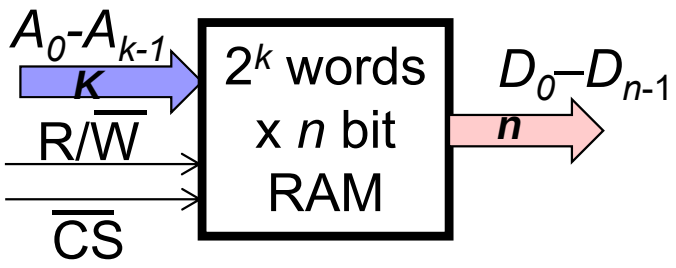
Example: Write 0xfedc to memory location at address 0x3456

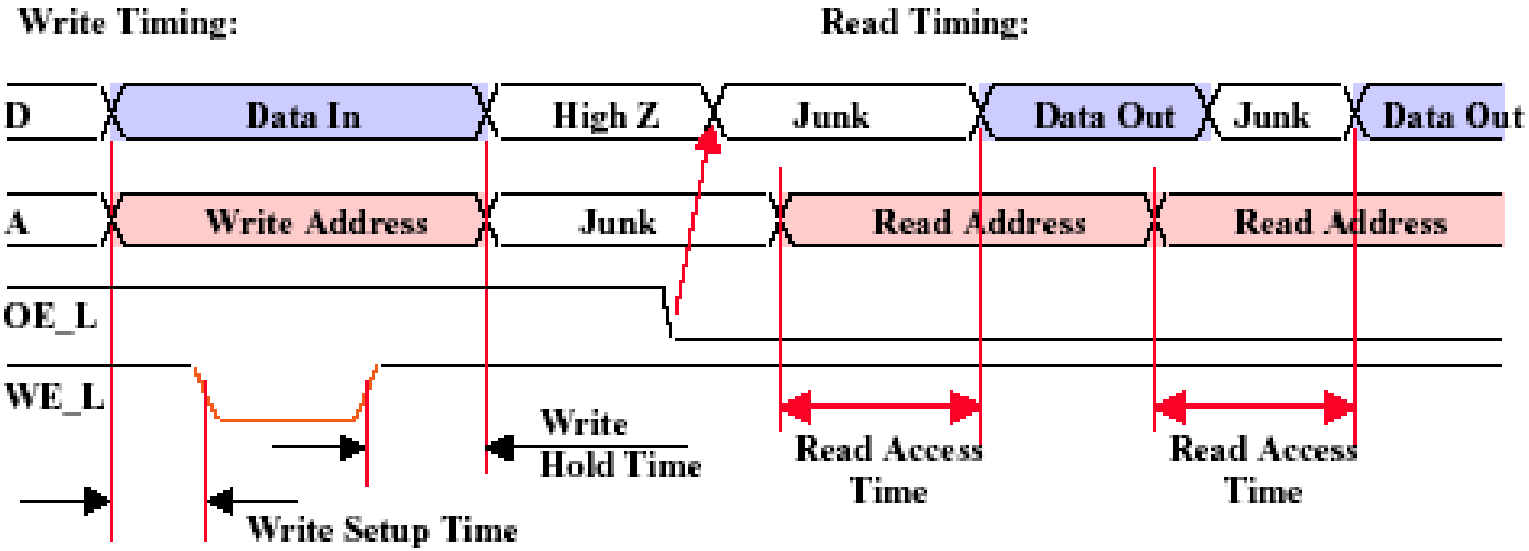




RAM Timing

- The **access time** (T_{AA}) is defined for reading, as the time delay from the moment when the *address* lines are made available to the time when the *data* become available at the output
- When reading data from a chip, after time period T_{AA}· *n*-bit data word appears on data lines D_0 - D_{k-1}
- When writing data to a chip the data lines must also be held for *Write Hold Time*

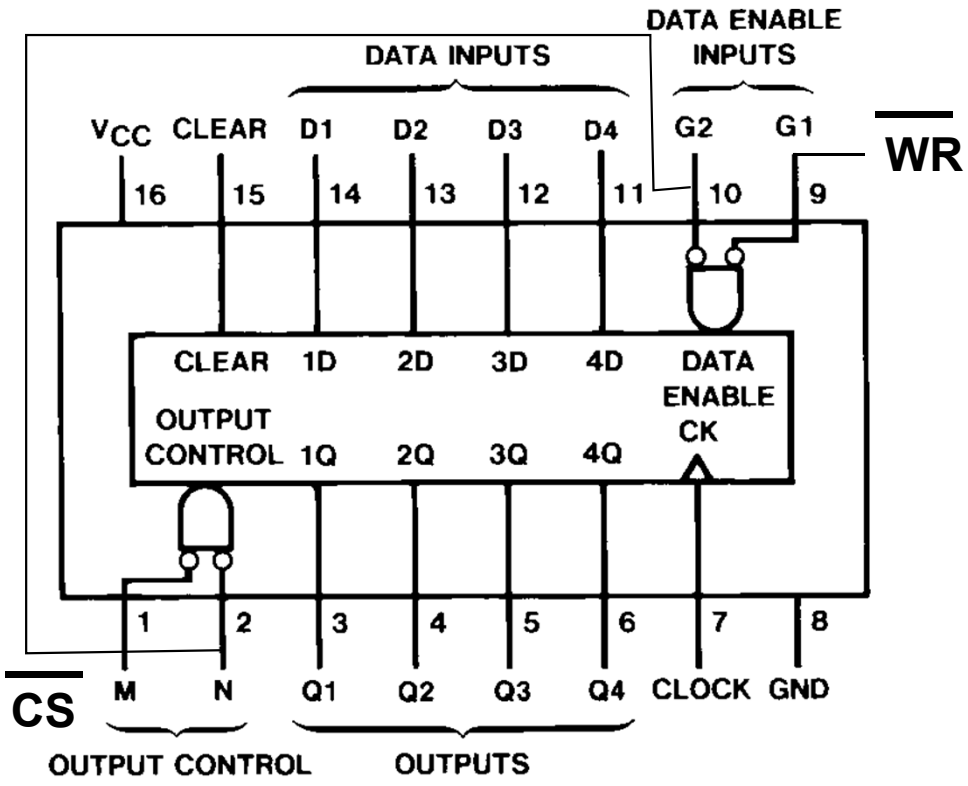


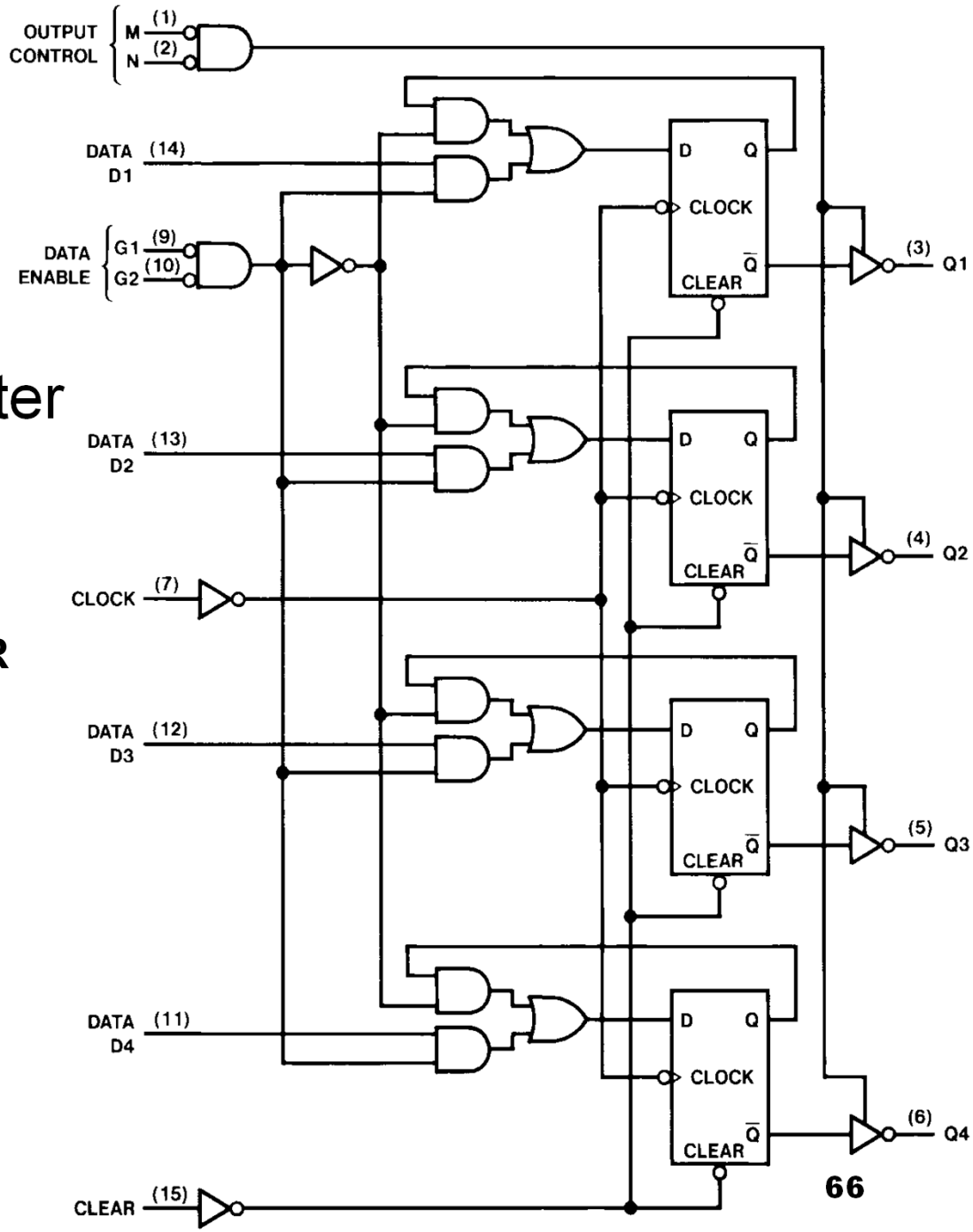


RAM that stores 4-bit words

- **Basic "brick**:" 74LS173

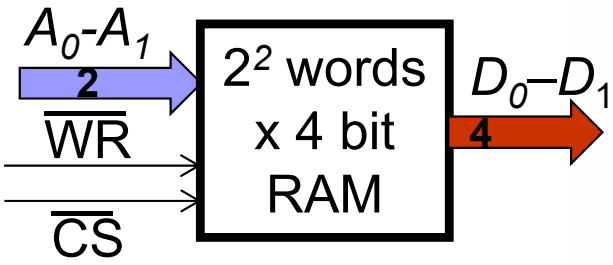
TRI-STATE 4-Bit D-Type Register

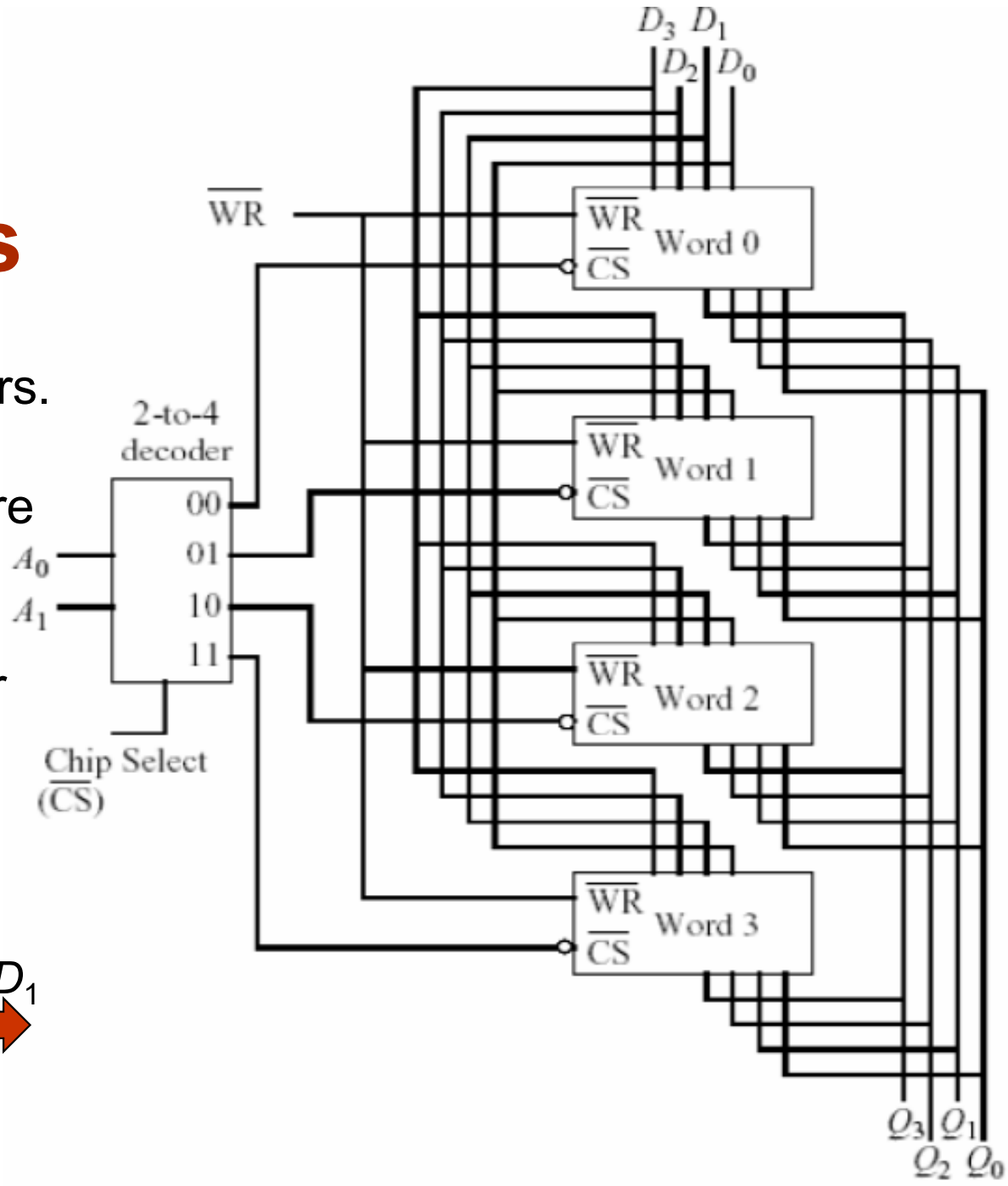




RAM module of 4 words of 4 bits

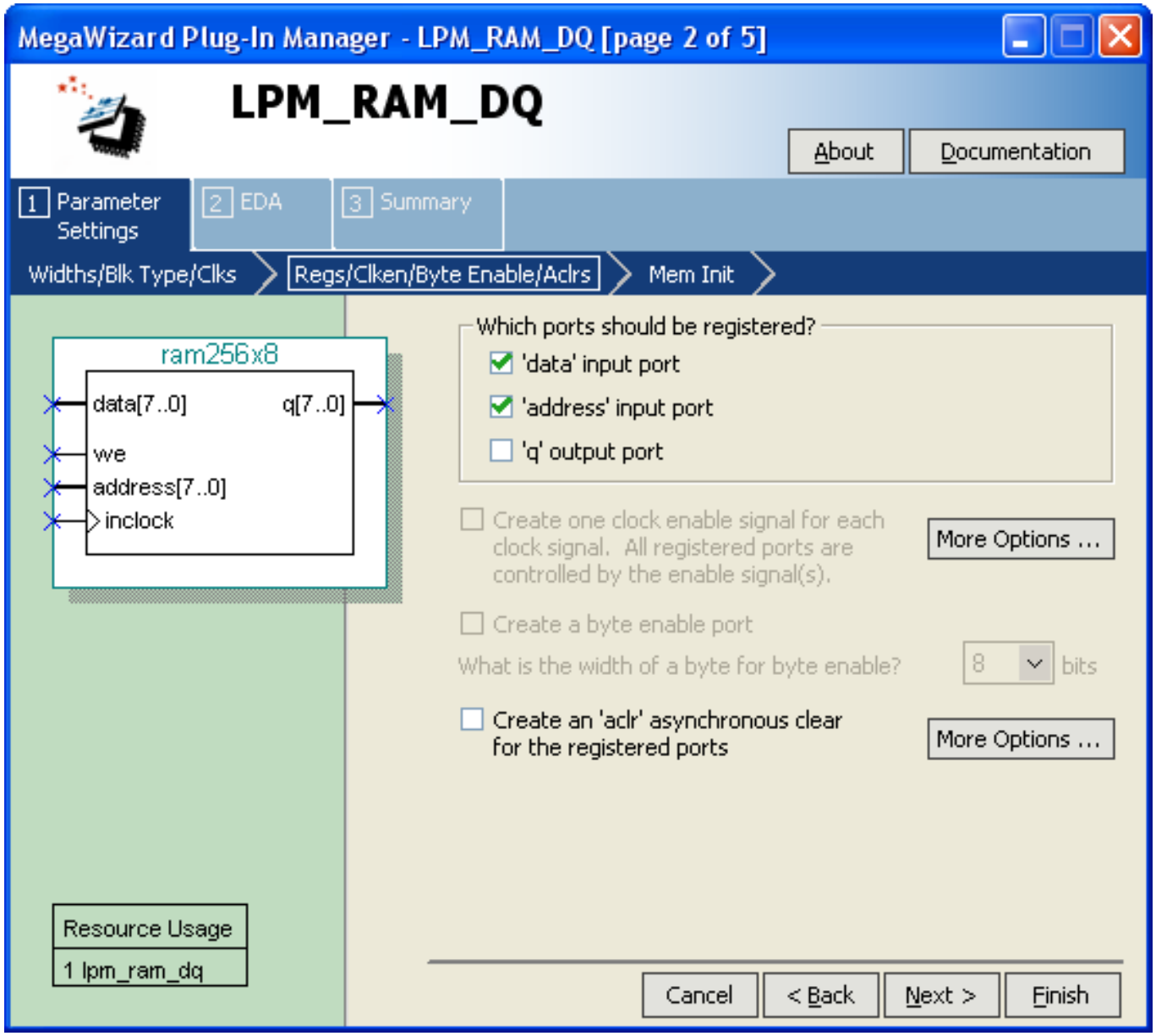
- RAM is a collection of registers.
- Four bit registers (such as 74LS173) can be used to store the words
- Using address vector A_1, A_0, one word can be selected for reading or writing out of 2^2 memory locations





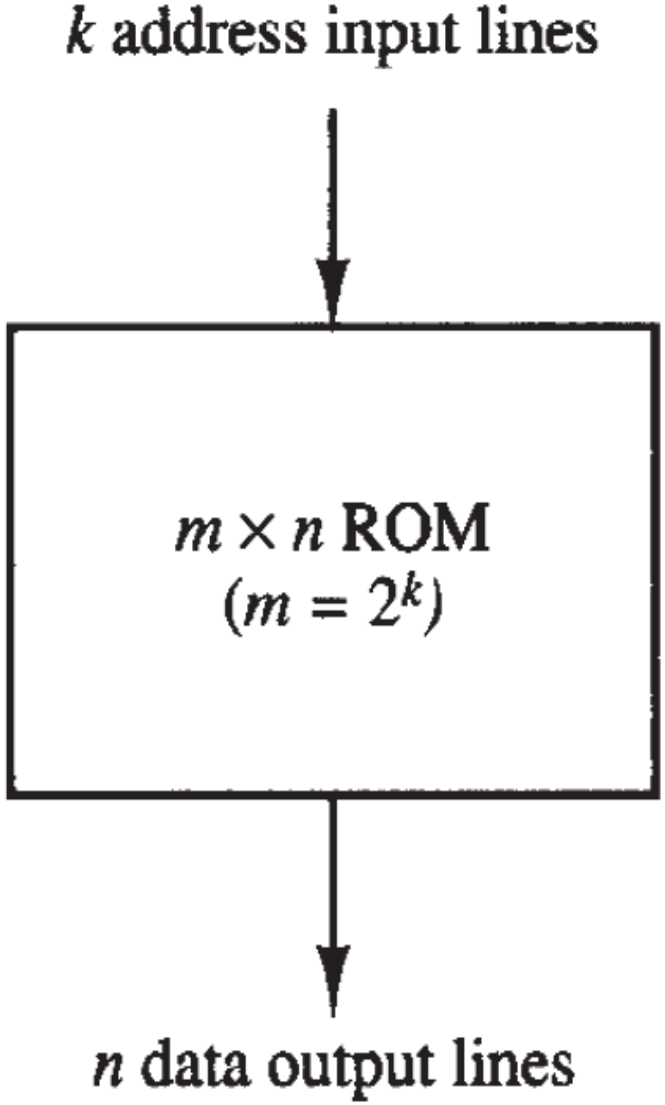
Altera memory module (ram256x8)

- Capacity:
  - 256 words
  - 8-bit long
- Interface:
  - Data in: data[7..0]
  - Data out: q[7..0]
  - address[7..0]
  - inclock
  - we



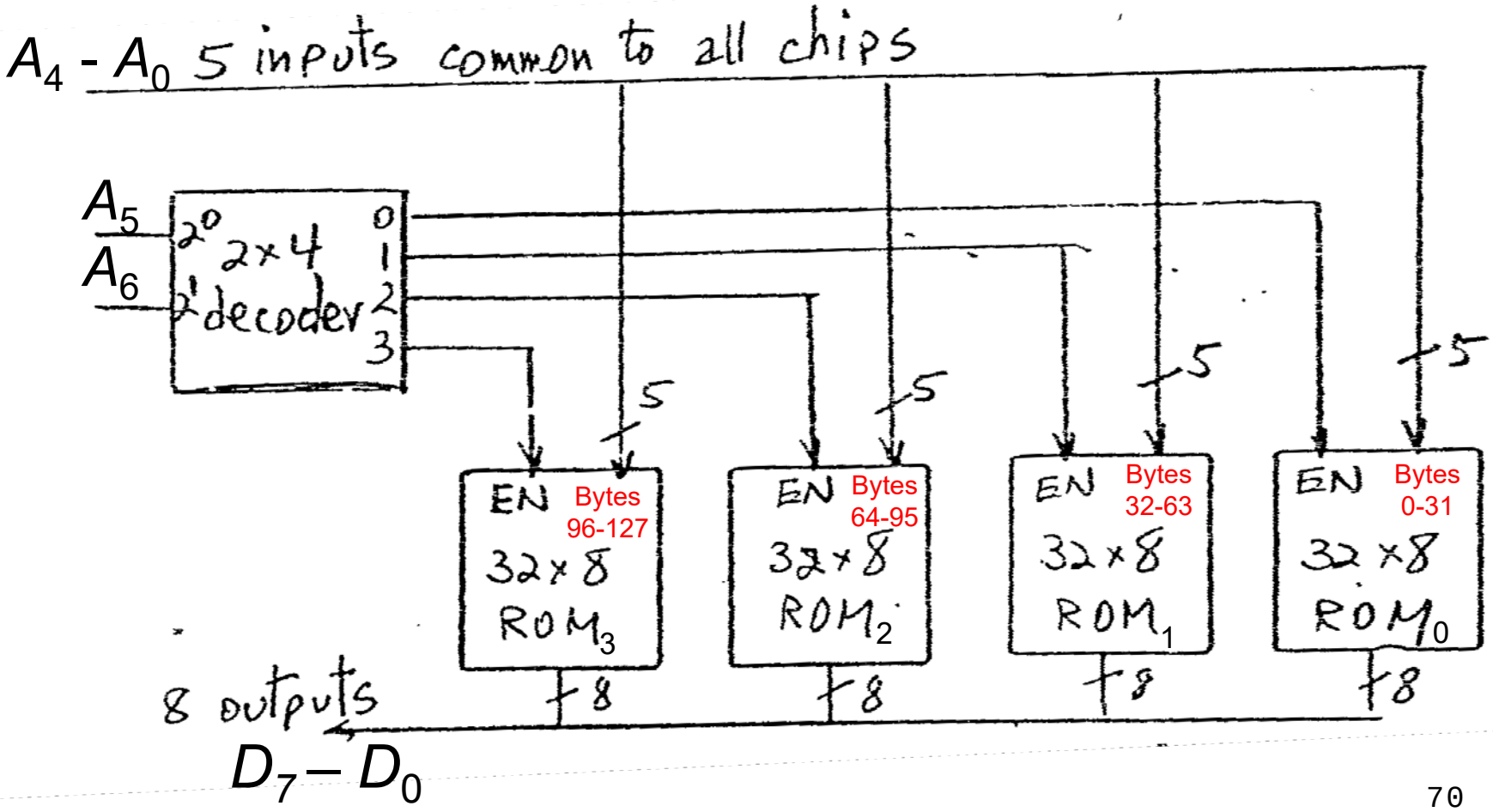
Read-Only Memory (ROM)

- A Read-Only Memory (ROM) is a memory unit that can only perform read operations (it does not have a "write" capability).
- As such, a typical ROM has no data input lines nor control inputs.
- An $m \times n$ ROM is an array of binary cells organized into m (= 2^k) words of ***n*** bits each, where k is the number of address bits.
- A ROM is classified as a combinational circuit.
- Figure: Block diagram of a typical ROM



ROM

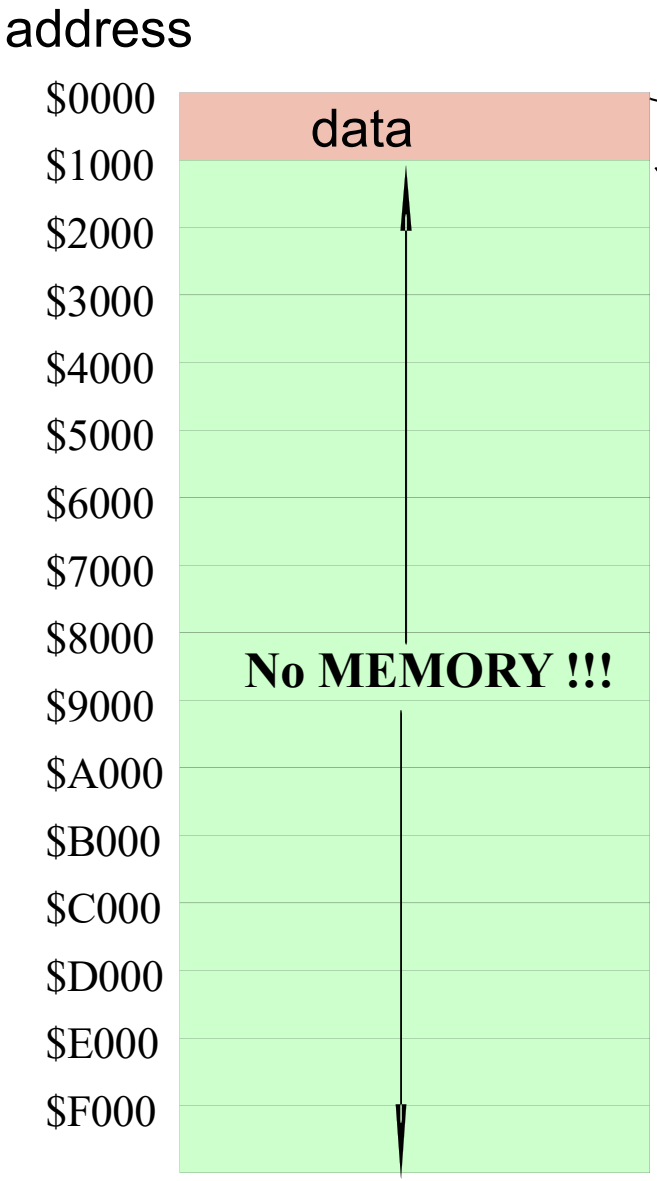
Given a 32 × 8 ROM chip with an enable input, show the external connections necessary to construct a 128 × 8 ROM with four chips and a decoder.

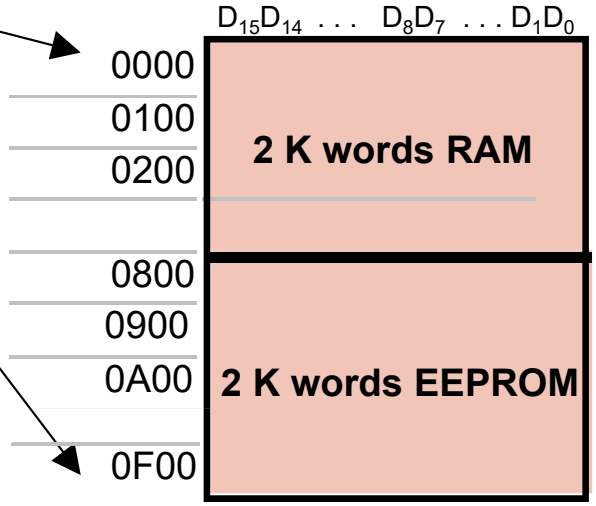


Memory Map

Memory **map** shows how all memory addresses are used:

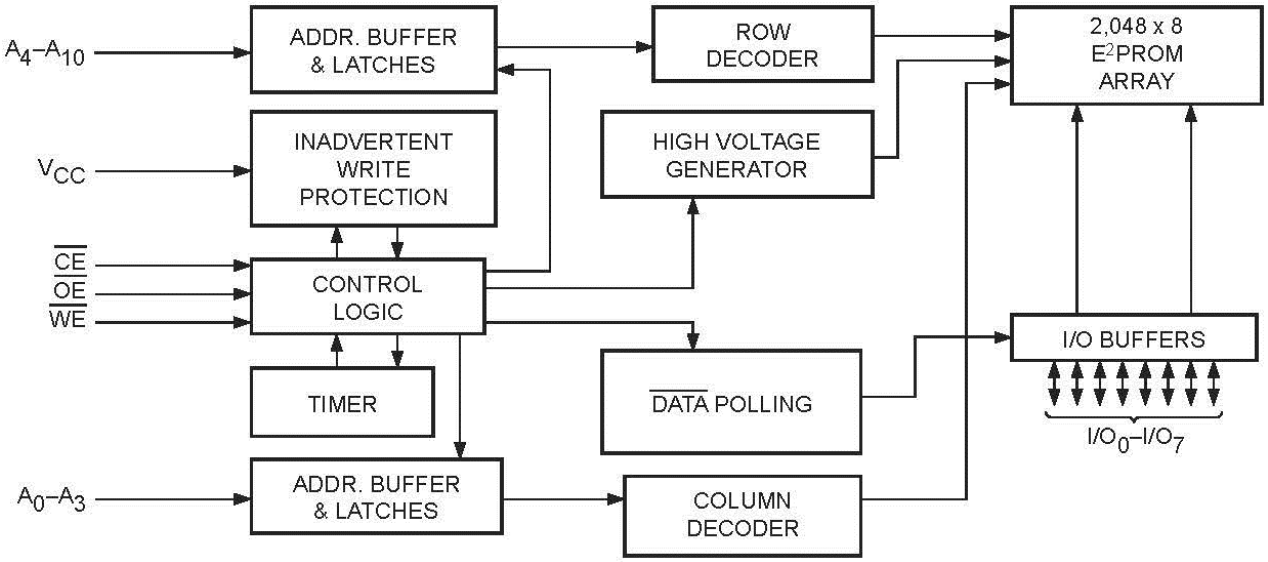
- A region may contain RAM
- Other region contains ROM
- Some regions may contain nothing



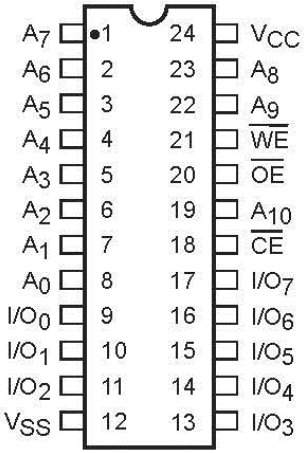


- ✓ Mano's **Basic Computer** has 12 bits for Address, so, its memory space is 4-kwords of 16 bits (2 bytes) each word, as shown above.
- ✓ Since this CPU has a 16 bit bus, it may have provisions for further expanding the memory to 64 k-words. Still, we want to build a memory as follows:
  - 1 K word RAM mapped to $0000 - $03FF
  - 1 K word RAM ($0400 to $07FF)
  - 2 K word EEPROM ($0800 to $0FFF)
  - 60 k words of **NOTHING** ($1000 to $FFFF)

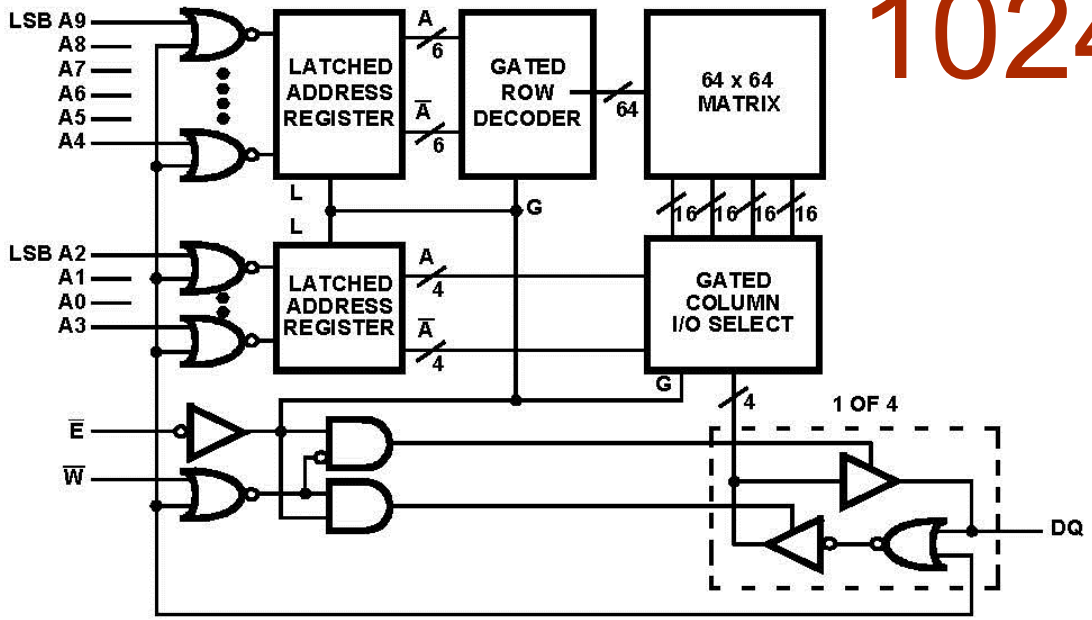
2048 x 8 CMOS E2PROM



Pin Name	Function
A_0–A_{10}	Address Inputs
I/O_0–I/O_7	Data Inputs/Outputs
$\overline{CE}$	Chip Enable
$\overline{OE}$	Output Enable
$\overline{WE}$	Write Enable
V_{CC}	5V Supply
V_{SS}	Ground
NC	No Connect

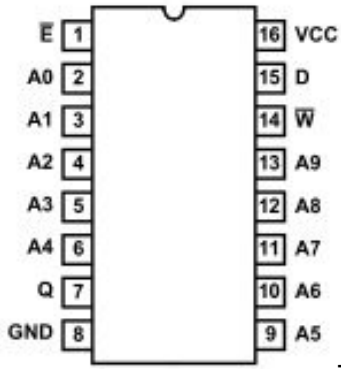


1024x4 CMOS RAM



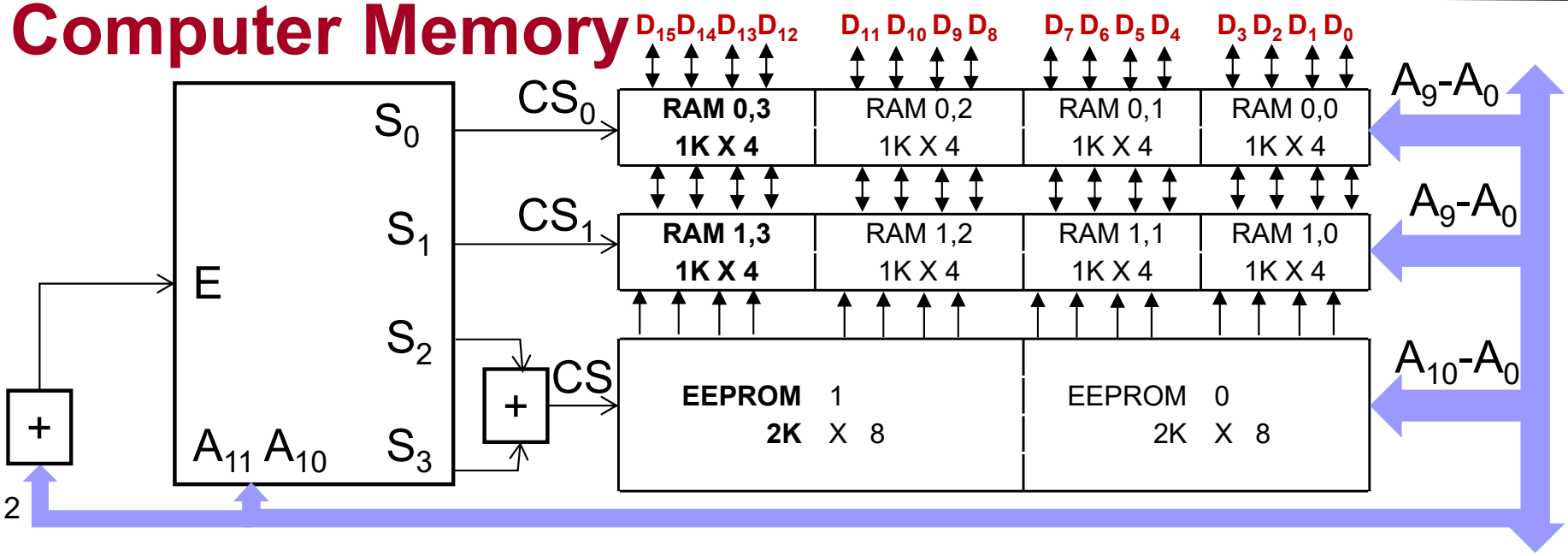
PIN	DESCRIPTION
A	Address Input
$\overline{E}$	Chip Enable
$\overline{W}$	Write Enable
D	Data Input
Q	Data Output

HM1-6508/883
(CERDIP)
TOP VIEW

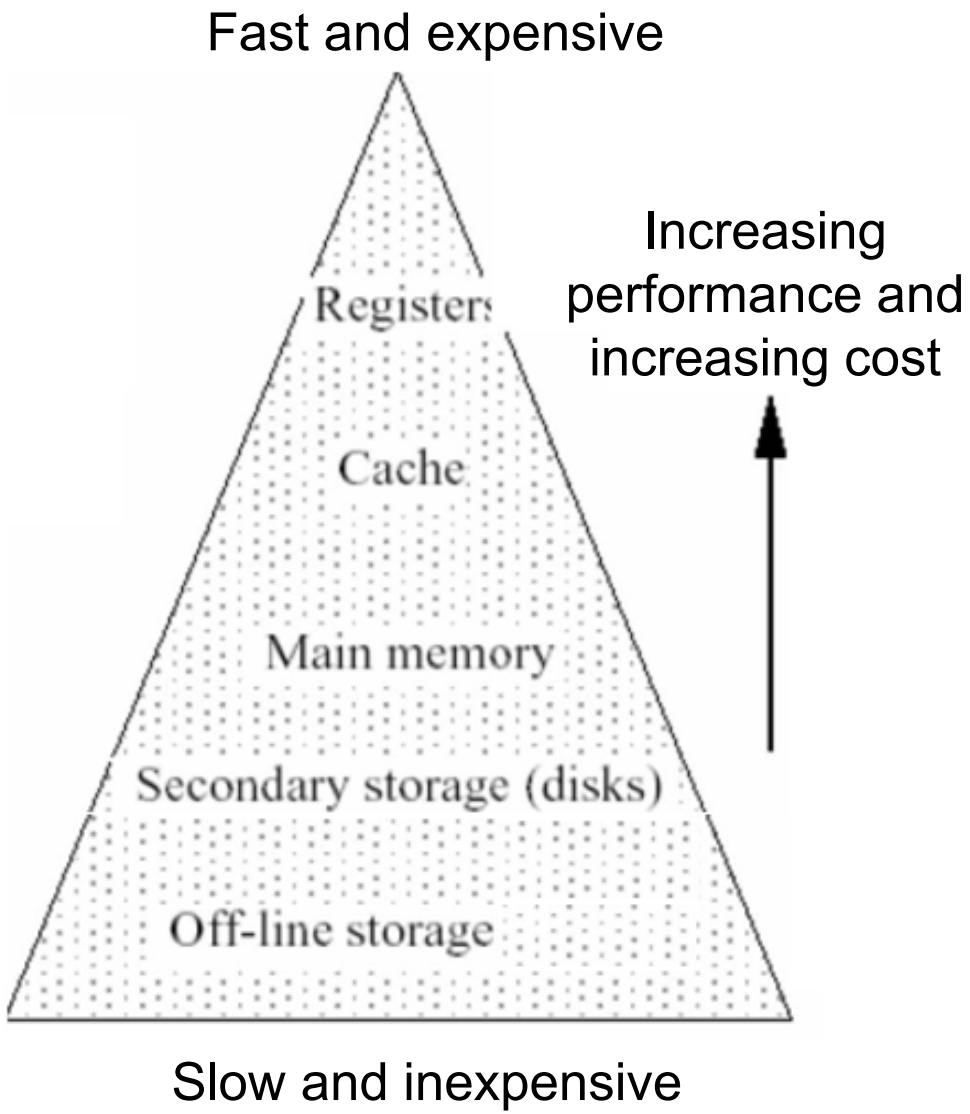


ADDR	ADDR (HEX)	$A_{15}A_{14}A_{13}A_{12}$	$A_{11}A_{10}A_9A_8$	$A_7A_6A_5A_4$	$A_3A_2A_1A_0$	$D_{15}D_{14}D_{13}D_{12}$	$D_{11}D_{10}D_9D_8$	$D_7D_6D_5D_4$	$D_3D_2D_1D_0$
0	0000	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0				
1	0001	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 1	RAM 0,3	RAM 0,2	RAM 0,1	RAM 0,0
	…	…	…	…	…	1K X 4	1K X 4	1K X 4	1K X 4
1023	03FF	0 0 0 0	0 0 1 1	1 1 1 1	1 1 1 1				
1024	0400	0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0				
1025	0401	0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 1	RAM 1,3	RAM 1,2	RAM 1,1	RAM 1,0
	…	…	…	…	…	1K X 4	1K X 4	1K X 4	1K X 4
2047	07FF	0 0 0 0	0 1 1 1	1 1 1 1	1 1 1 1				
2048	0800	0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 0				
2049	0801	0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 1				
	…	…	…	…	…				
	0BFF	0 0 0 0	1 0 1 1	1 1 1 1	1 1 1 1	EEPROM	1	EEPROM	0
	0C00	0 0 0 0	1 1 0 0	0 0 0 0	0 0 0 0	2K	X 8	2K	X 8
	0C01	0 0 0 0	1 1 0 0	0 0 0 0	0 0 0 1				
	…								
4095	0FFF	0 0 0 0	1 1 1 1	1 1 1 1	1 1 1 1				

Basic Computer Memory



Memory Hierarchy

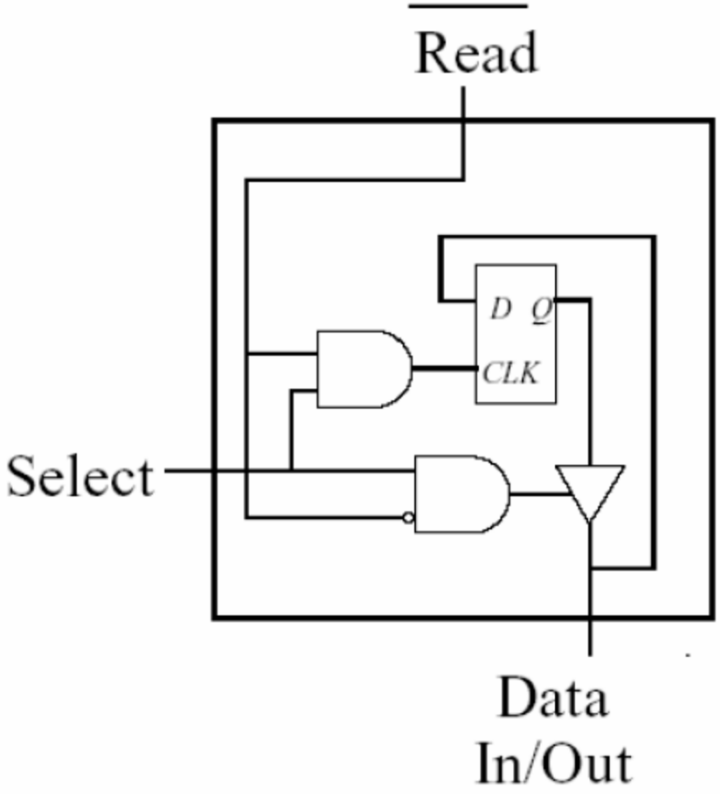


Annex

Static RAM (SRAM) cell

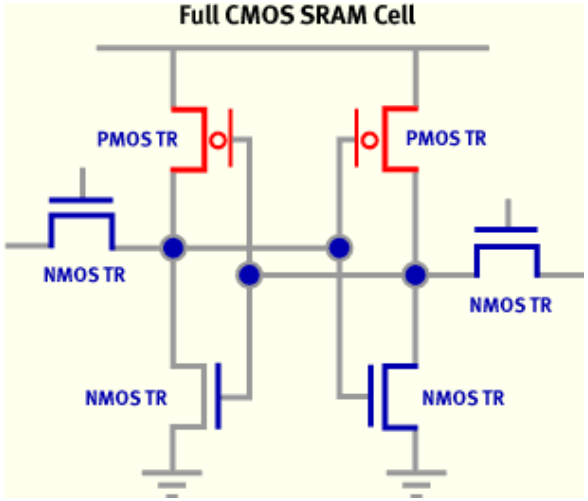
- Any location can be accessed in the same amount of time (RAM - Random Access Memory)
- The RAM chip based on D flip-flop with logic to allow the cell to select, read and written
- **Static** = a current flows through a branch of the latch
- Bi-directional line for data in and data out
- RAM - static, content of each location persist as long as power is ON (volatile)

SRAM Cell
logic diagram



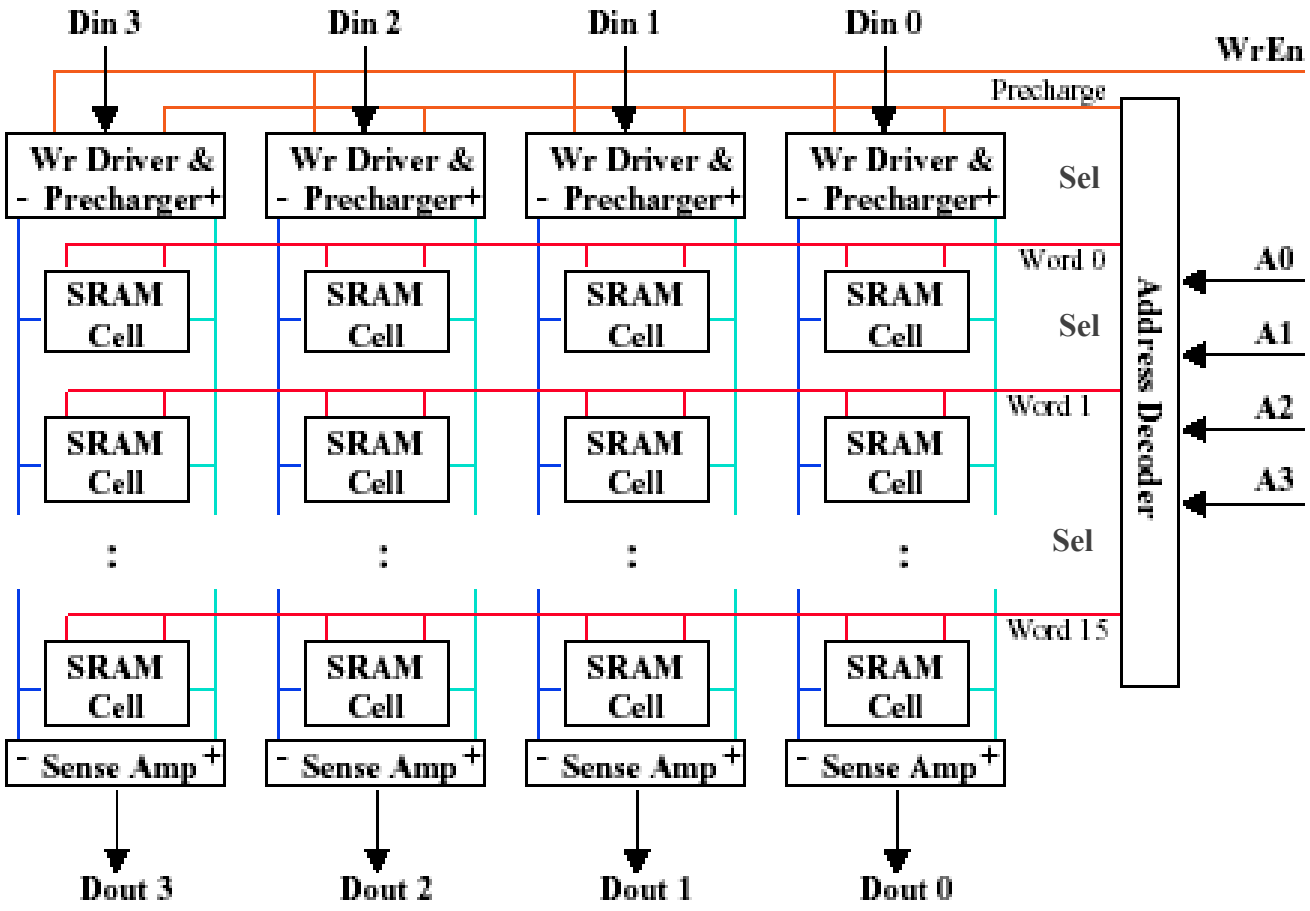
SRAM (Static RAM)

6-T SRAM Cell electronic diagram

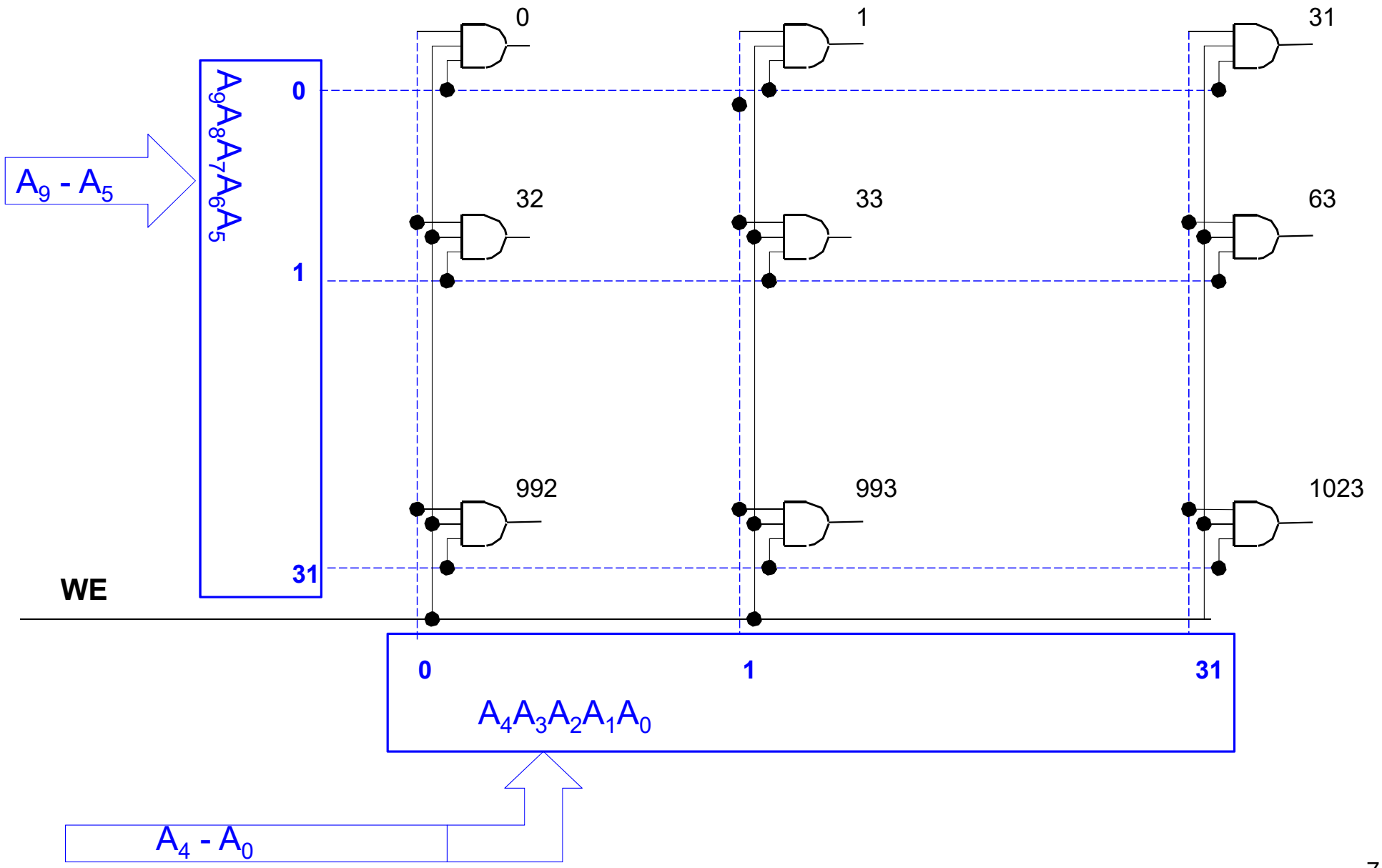


- Static: a current flows through a branch of the latch
- content will last “forever” (until lose power)
- Low density, high power, expensive, fast

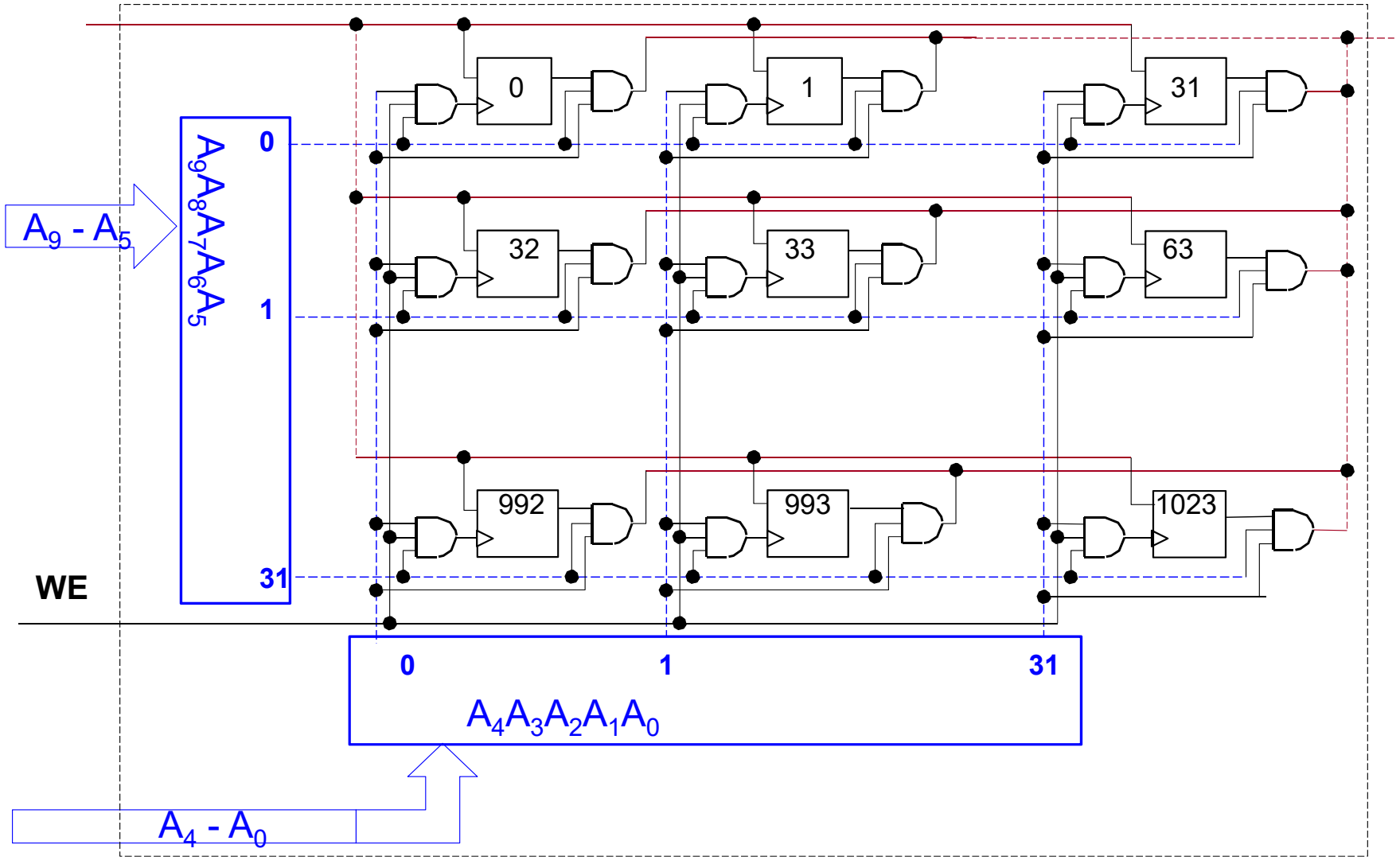
SRAM block diagram



Address Decoder = Matrix-type

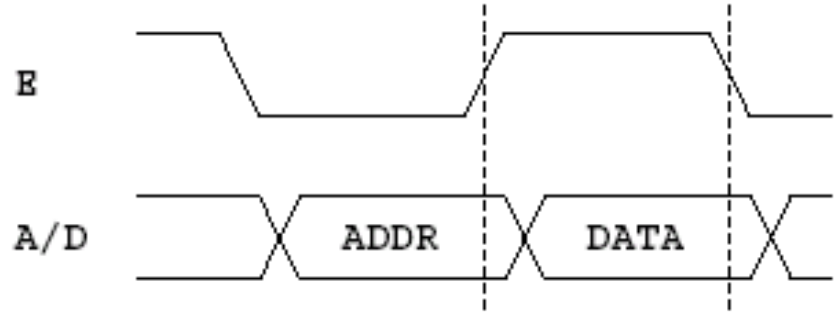


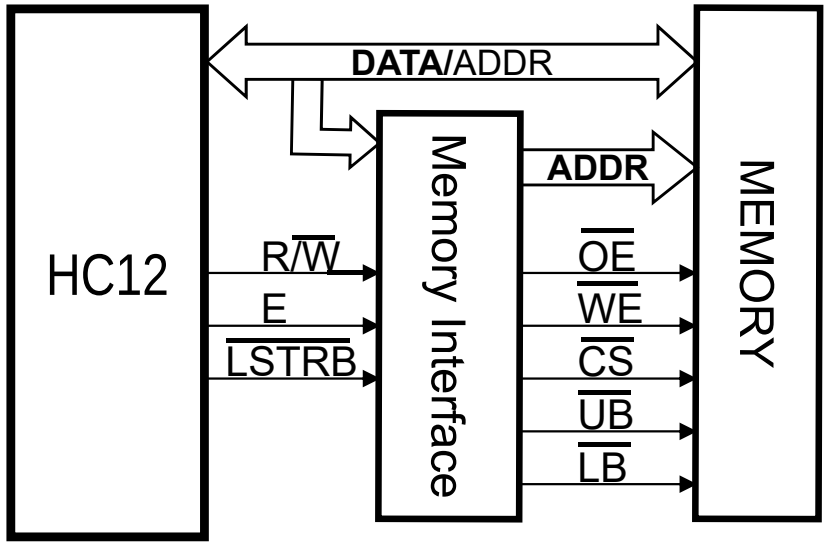
1 K x 1-Bit RAM module



Multiplexed Address/Data Bus

There are microprocessors which do not have enough pins for the number of signals they carry, such that they share the same pins for different functions by time-multiplexing signals. e.g., HC12 uses the same pins for DATA/ADDR as follows:





When the E-clock is

- –**low** => the sixteen DATA/ADDR lines (AD15-0) are used for **address**
- –**high** => the sixteen DATA/ADDR lines (AD15-0) are used for **data**

CS - Chip Select

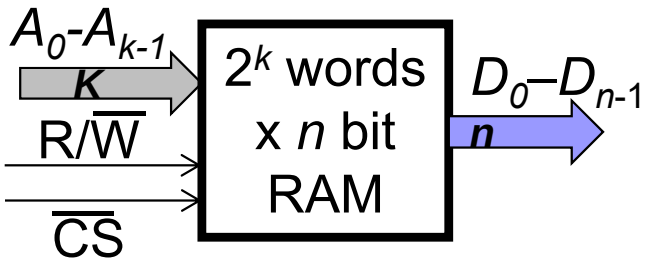
WE - Write Enable

OE - Output Enable (Read)

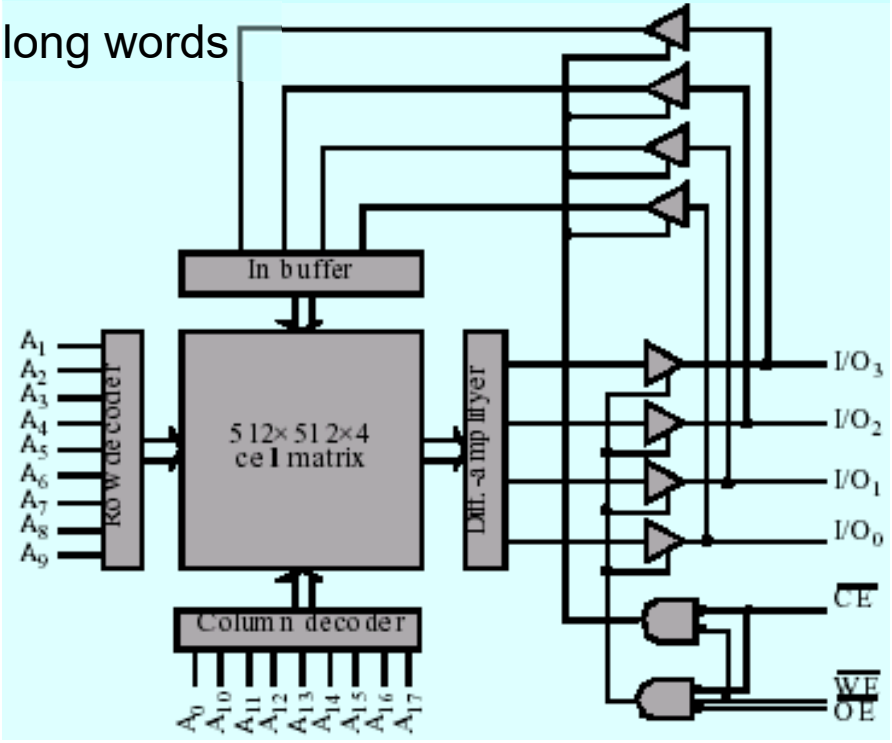
UB - High (upper) Byte Enable

LB - Lower Byte Enable

SRAM Chip



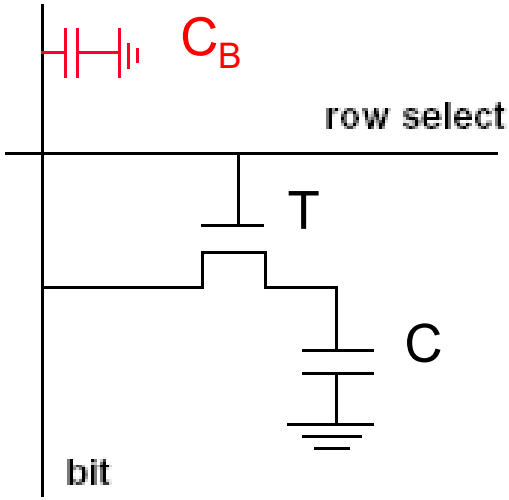
SRAM: 256 k-word ($2^{18} = 2^8$ x1024) *n*=4-bit long words



- *k*-bit address lines 0 to *k*-1 is applied to A_0 - A_{k-1} (in the next figure *k* = 18)
- $\overline{CS}$ chip select (active low) and
- R/$\overline{W}$ high => Read; R/$\overline{W}$ low => Write
- Data lines (I/O) are bi-directional
- Address lines A_0 - A_{k-1} contains address. The address lines are decoded into one of 2^k locations
- Each location has *n*-bit word (here *n* = 4)
- The chip therefore has a capacity of 2^k x *n* bits
  - here: 2^{18} x 4 bits = 256 k-word ($2^{18} = 2^8$ x1024) of 4-bit long words

DRAM (Dynamic RAM)

1-Transistor Cell



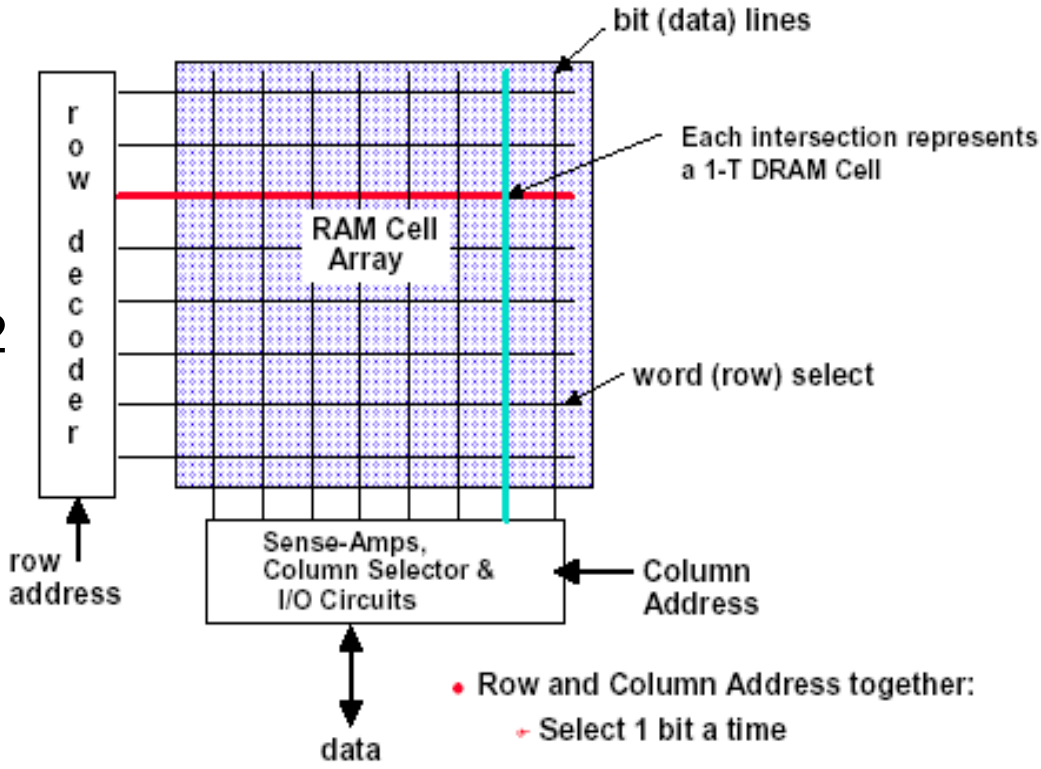
- **Write**:
  1. Drive bit line
  2. Select row

  => C charges at bit-line voltage
- **Read**:
  1. Precharge bit line to Vdd/2
  2. Select row
  3. Cell (C) and bit line (C_B) share charges

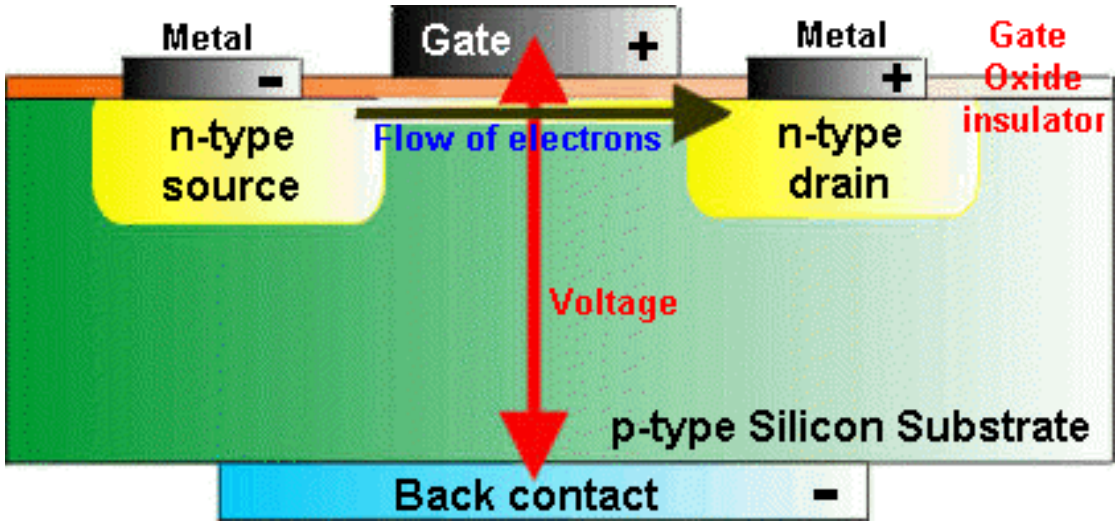
  ➔ Very small voltage changes on the bit line

  $V_B = V_{DD}/2\ [1 \pm C_B/(C + C_B)]$
  4. Sense amplifiers compare V_B with $V_{DD}/2$

  ➔ Can detect changes of ~1 M electrons
  5. Write: restore the voltage value on C_B and, as such, on C since C_B and C are connected together over transistor T
- **Refresh**
  1. It's a dummy read to every cell.

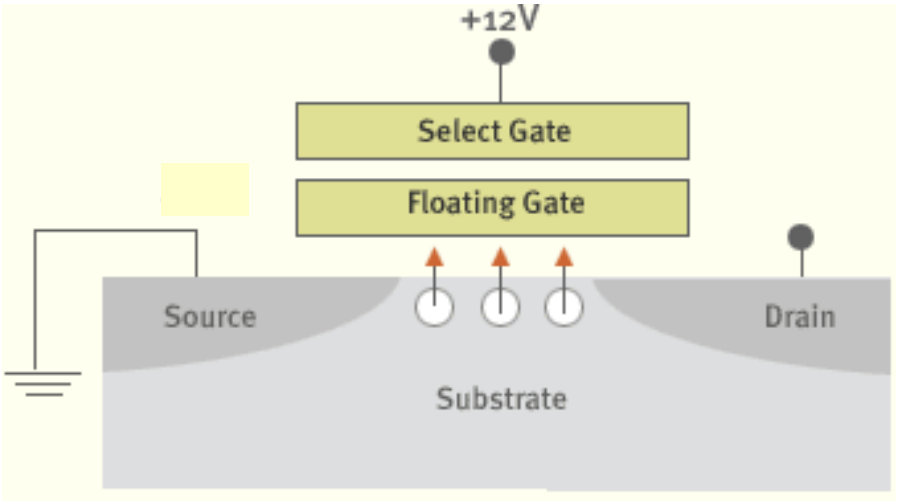


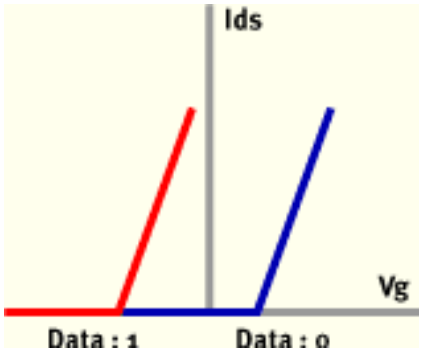
EPROM CELL


Metal
Gate
Metal
Gate
Oxide
insulator
n-type
source
Flow of electrons
n-type
drain
Voltage
p-type Silicon Substrate
Back contact

Programming EPROM Cell

- Changing a Flash Memory cell or bit to a zero is called programming.
- As electrons travel from the source to the drain through the substrate, the electric field generated by **high voltage** on the **select gate** causes some of the highest energy electrons to jump the gap and collect on the floating gate. The electrons now present on the floating gate counteract the voltage on the select gate and prevent the flash memory cell from turning on.
- No current flows from drain to source, resulting in a zero on the memory output pin.





Erasing EEPROM Cell

- Memory cells must be erased before they can be overwritten.
- The generated electric field pulls electrons from the floating gate Flash memories erase all cells in the array at the same time. EEPROM memories erase in smaller blocks.
- After "Erase Operation", a cell has data "1" and its threshold voltage becomes negative.