Semiconductor Memories: RAMs and ROMs

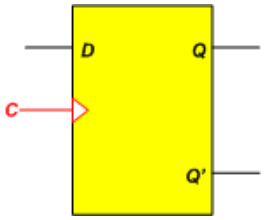
Lesson Objectives:

In this lesson you will be introduced to:

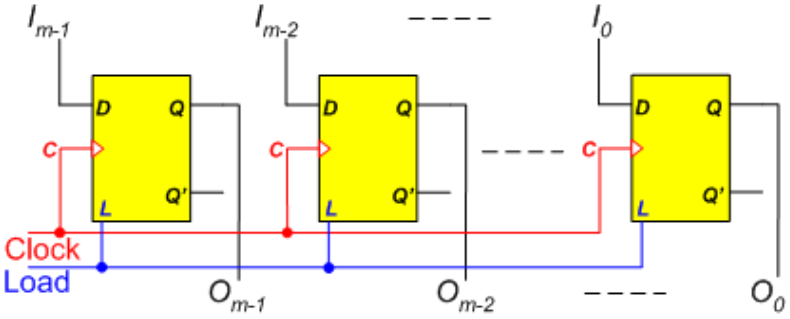
- Different memory devices like, ***RAM, ROM, PROM, EPROM, EEPROM,*** etc.
- Different terms like: ***read, write, access time, nibble, byte, bus, word, word length, address, volatile, non-volatile etc.***
- How to implement combinational and sequential circuits using ROM.

Introduction:

The smallest unit of information a digital system can store is a ***bit***, which can be stored in a ***flip-flop*** or a ***1-bit register***.



To store *m* bits of data, an ***m-bit register*** with parallel load capability may be used. Data available on the ***m***-bit input lines (I_0 to I_{m-1}) may be stored/*written* into this register under control of the clock by asserting the "*Load*" control input. The stored ***m*** bits of data may be *read* from the register outputs (O_0 to O_{m-1}).

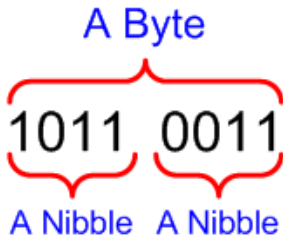


The ***m*** bits of data stored in a register make up a ***word***. It is simply a number of bits operated upon or considered by the hardware as a group. The number of bits in the word, ***m,*** is called ***word length***.

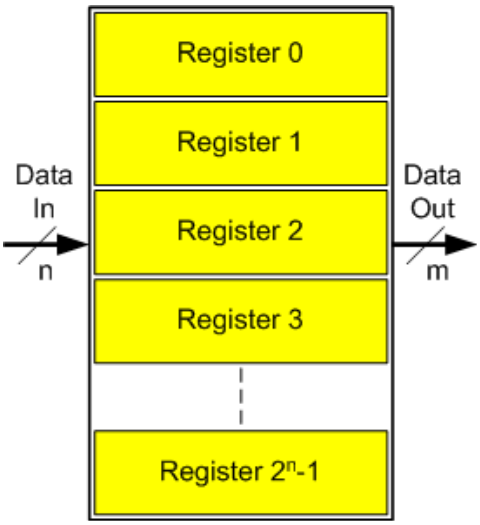
The *m* inputs of the register are provided through an *m*-bit input data *bus* and *m* outputs by an *m*-bit output data *bus*.

A ***bus*** is a number of signal lines, grouped together because of similarity of function, which connect two or more systems or subsystems.

A unit of 8-bits of information is referred to as a ***byte***, while 4-bits of information is referred to as a ***nibble***.



A ***memory*** device can be looked at as consisting of a number of equally sized registers sharing a *common set of inputs*, and a *common set of outputs*, as shown in the Figure.



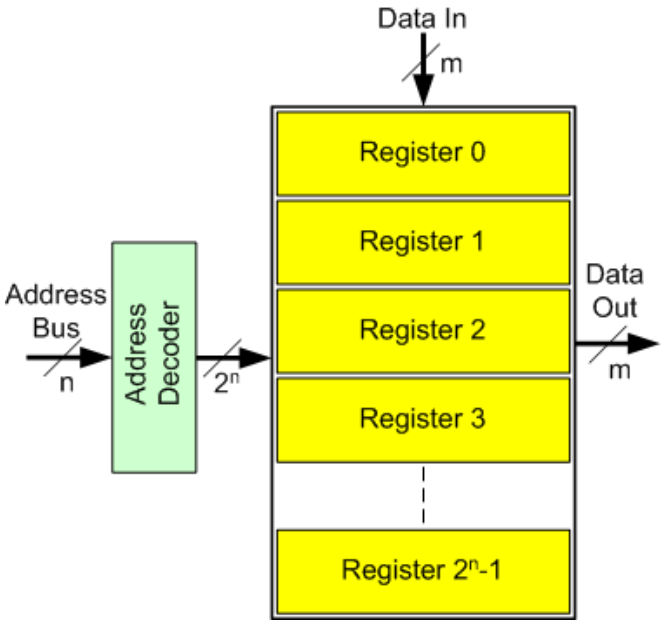
Storing data in a memory register is referred to as a memory ***write*** operation and looking up the contents of a memory register is referred to as a memory ***read*** operation.

In case of a write operation, the input data need to be written into one *particular* register in the memory device.

Since the input data lines are common to all registers of the memory device, only the selected register should have its ***load*** control signal asserted while the other registers should not.

If the number of registers is 2^n, n lines will be required to select the register to be written into. The n-lines are used as an input to a decoder where the decoder's 2^n outputs may be used as the ***load*** control inputs to the 2^n registers.

The load control signal of a particular register is asserted by a unique combination of the n-select lines. This unique combination is considered as the ***address*** for that particular register.

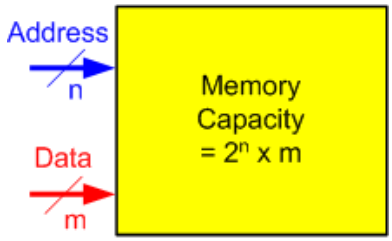


Thus, a memory device can be thought of as a collection of **addressable** registers.

A read or a write operation into the memory device has to specify the address of the particular register to be read or written into.

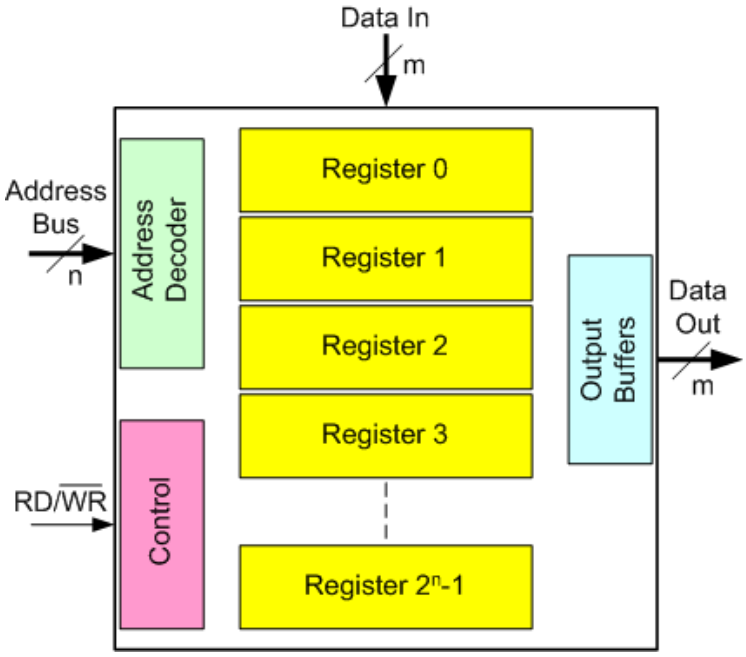
The ***capacity*** of the memory is specified in terms of the number of bits or the number of words available in this memory device.

For a memory device with ***n***-bit address lines and word (register) size of ***m***-bits, the memory has 2^n words (storage locations/registers) each having ***m*** bits for a total capacity of 2^n **x** ***m*** bits.



For example, if $n = 10$ and $m = 8$, the memory is a **"1024 x 8"** bit memory. Alternatively, it is said that the memory has 1K bytes.

A block diagram of the memory device is shown in the figure. The address inputs are decoded by **address decoder** to select one, and only one, of the memory words (registers), either for reading or writing.



The $RD/\overline{WR}$ line is a control signal that determines the type of operation to be performed; a read operation or a write operation.

$RD/\overline{WR}=1$ indicates a read operation, while $RD/\overline{WR}=0$ indicates a write operation.

To **read** the memory contents stored in a particular word, the address of this word is applied, and logic 1 is applied to the $RD/\overline{WR}$ line that enables the output buffers of the memory.

To **write** at a location, the address of the location to be written is provided at the address inputs, data is provided at the data inputs, and logic 0 is applied to the $RD/\overline{WR}$ line.

There is a time delay between the application of an address and the appearance of contents at the output, this is called the memory ***access time***. This depends both on the technology and on the structure used to implement the memory.

Random Access Memory (RAM):

For the shown above memory structure, the access time is independent of the sequence in which addresses are applied.

Such a memory is called ***random access memory (RAM).*** Thus, the contents of any one location can be accessed in essentially the same time as can the contents of any other location chosen at random.

RAMs are ***volatile*** memories that will only retain the stored data as long as power is **ON** but will lose this data when power is turned **OFF**.

RAMs are classified into two main categories: Static RAM ***(SRAM)*** and Dynamic RAM ***(DRAM).*** These will be studied in greater details in future courses.

Read Only Memory (ROM):

Read Only Memory (ROM) is memory whose stored data can only be read but cannot be re-written (altered).

It is a device in which "permanent" binary information has been stored.

ROMs are ***nonvolatile*** where stored data are not lost even when power is turned **OFF**.

The Figure shows a block diagram of a ROM.

Like RAMS, a ROM has ***n*** *address* inputs and ***m*** outputs. This corresponds to 2^n memory words each of *m* storage bits for a total capacity of 2^n **x** ***m*** bits.

Unlike RAMs, ROMs do not have data input lines, because they do not have a write operation.

ROMs are common to use in storing system-level programs that should be available at all times.

The most common example is the PC system BIOS (Basic Input Output System), which is stored in a ROM called the *system BIOS ROM*.

Several classes of ROMs are in common use. These may be categorized according to their fabrication technologies that influence the way data are introduced into the ROM. The process of storing the desired data into the ROM is referred to as *ROM programming*.

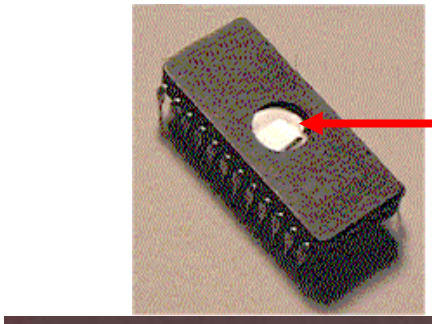
Types of ROMs:

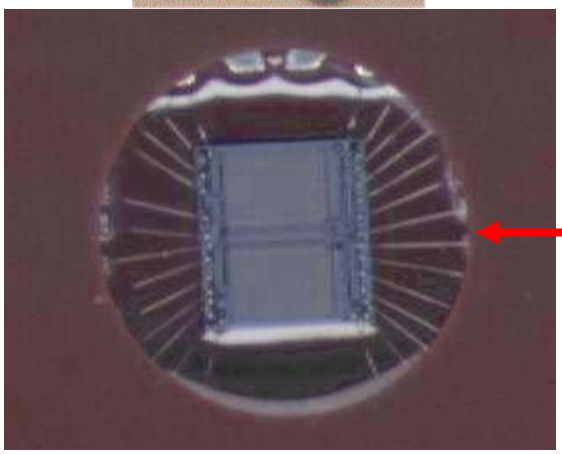
Following are the different types of ROMs.

1. Programming is done by the manufacturer during the last fabrication steps according to the truth table provided by the customer. This type is known as mask programmable ROMs or simply ***ROM***. Data stored this way can never be altered.

2. ROM is provided with fuses to allow users to introduce the desired data by electrically blowing some of these fuses. This type is referred to as a *programmable ROM*, or ***PROM***. Fuse blowing is irreversible and, once programmed the ROM stored pattern cannot be altered.

3. The ROM uses *erasable floating-gate* memory cells that allow erasure of the stored data by Ultra-Violet light. In this type, programming is performed *electrically* by the user using special hardware programmers. Data, thus stored, can later be erased globally (all memory bits = 1) by exposing the memory array to UV-light. This ROM type is referred to as UV-*erasable, programmable ROM*, or simply ***EPROM***. The EPROM IC package is provided with a quartz window to allow UV-light penetration to the memory array.





4. When special electrically erasable memory cells are used, the ROM can be electrically erased at the byte level. Thus individual bytes may be addressed and programmed or erased as desired. This type is referred to as *electrically erasable, programmable ROM*, or ***EEPROM*** or ***E^2PROM***. The ***E^2PROM*** technology is an expensive low-capacity technology and is thus not used for high density or low-cost applications.

5. The most recent ROM technology is the *flash* technology that combines the low-cost and high-density advantages of the UV-EPROM technology and the flexibility of electrical erase of ***E^2PROM*** technology. This technology is electrically erasable but the erasure is performed either globally (the full array) or partially on complete sub-arrays (sectors).

Combinational Circuit Implementation Using ROM:

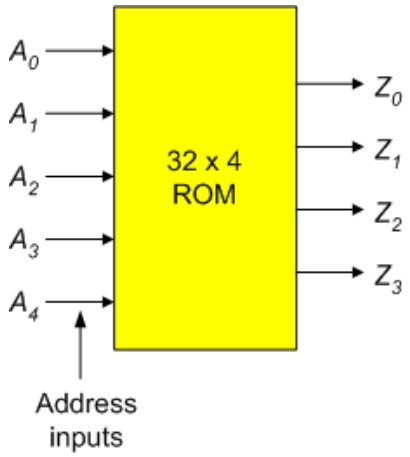
ROM devices can be used to implement complex combinational circuits directly from truth tables without need for minimization.

For an ***n***-input, ***m***-output combinational circuit, a 2^n **x** ***m*** ROM is needed (2^n words each of ***m*** storage bits). The designer needs only to specify a ROM table that gives the information stored in each of the 2^n words.

When a combinational circuit is implemented using a ROM, the function may either be expressed in the sum of minterms form, or using a truth table.

As an example, the ROM shown in the figure may be considered as a combinational circuit with four outputs, each a function of the five input variables.

Outputs $Z_0 - Z_3$ can be expressed as sum of minterms as follows:
$Z_0 (A_4, A_3, A_2, A_1, A_0) = \sum m\ (2, 3, 18, 21, 31)$
$Z_1 (A_4, A_3, A_2, A_1, A_0) = \sum m\ (0, 1, 17, 25, 31)$
$Z_2 (A_4, A_3, A_2, A_1, A_0) = \sum m\ (1, 6, 11, 29, 30)$
$Z_3 (A_4, A_3, A_2, A_1, A_0) = \sum m\ (7, 8, 16, 28, 29)$



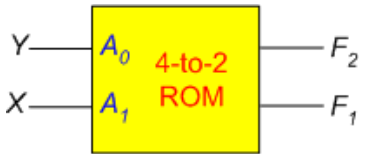
Example 1:

Consider a combinational circuit which is specified by the following two functions:
F_1 (X, Y) = $\sum$m (1, 2, 3)
F_2 (X, Y) = $\sum$m (0, 2)

The truth table for this circuit is as shown.

X	Y	F_1	F_2
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	0

In this example, the ROM that implements the two combinational functions must have two address inputs and two outputs. Thus, its size must be 4 x 2 (since 2^n **x** ***m*** is the size of ROM).



The ROM table for this example is as shown.

ROM Address		Stored Information	
A_1	A_0	F_1	F_2
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	0

Example 2:

Design a combinational circuit using a ROM. The circuit accepts a 3-bit number and generates an output binary number that is equal to the square of the input number.

The first step is to derive the truth table for the combinational circuit as shown. Three inputs and six outputs are needed to accommodate all possible numbers.

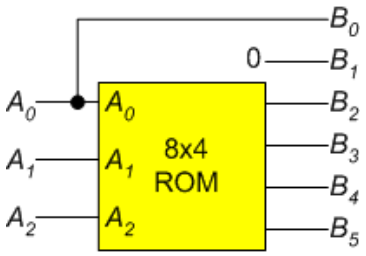
Inputs			Outputs						
A_2	A_1	A_0	B_5	B_4	B_3	B_2	B_1	B_0	Decimal
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	0	0	0	0	1	49

By observation, we note that output B_0 is always equal to input A_0, and output B_1 is always 0. Thus, there is no need to store B_0 and B_1 in the ROM. We actually need to only store values of the four outputs (B_5 through B_2) in the ROM.

The table shown specifies all the information that needs to be stored in the ROM, and figure shows the required connections of the combinational circuit. The output B_1 is connected to logic 0 and output B_0 is connected to A_0 always to get $B_1 = 0$ and $B_0 = A_0$.

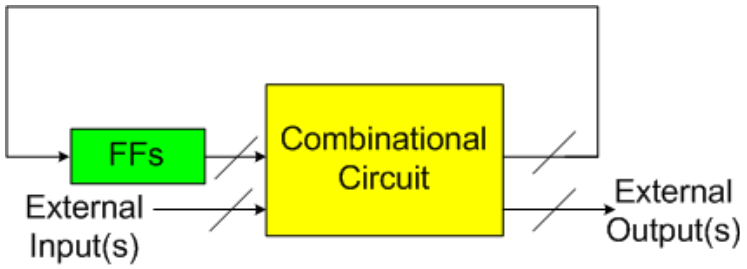
ROM Address			Stored Information			
A_2	A_1	A_0	B_5	B_4	B_3	B_2
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	0	0	0

The minimum size ROM needed must have three inputs and four outputs, for a total of 8 x 4 = 32 bits.

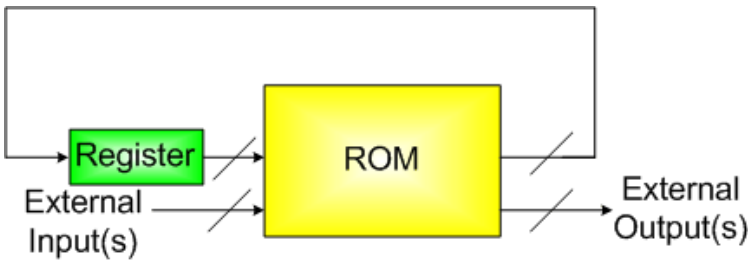


Synchronous Sequential Circuit Implementation Using ROM:

The block diagram of a sequential circuit is shown in the figure.



Since ROM can implement combinational logic, so this part can be replaced by a ROM and Flip-Flops can be replaced by a register as shown in the figure.



Example 3:

Design a sequential circuit whose state transition table is given, using a ROM and a register.

Present State		Input	Next State		Output
Q_2	Q_1	X	Q_2^+	Q_1^+	Y
0	0	0	0	0	0
0	0	1	0	1	0
1	0	0	0	1	0
1	0	1	0	0	1
0	1	0	1	0	0
0	1	1	0	1	0
1	1	0	1	1	0
1	1	1	0	0	1

The next-state and output information are obtained from the table as:

$Q_1^+ = \sum m\ (1, 2, 5, 6)$

$Q_2^+ = \sum m\ (4, 6)$

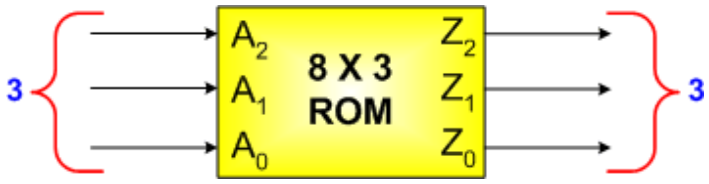
$Y\ (Q_2, Q_1, X) = \sum m\ (3, 7)$

The ROM can be used to implement the combinational circuit and register will provide the flip-flops.

The **number of address inputs** to the ROM is equal to the number of flip-flops plus the number of external inputs.

The **number of outputs** of the ROM is equal to the number of flip-flops plus the number of external outputs.

In this example, 3 inputs and 3 outputs of the ROM are required; so its size must be 8 x 3.



The ROM table is identical to the state transition table with Present State and Inputs specifying the **address** of ROM and Next State and Outputs specifying the ROM **outputs (stored information)**. It is shown below:

ROM Address			Stored Information		
A_2	A_1	A_0	Z_2	Z_1	Z_0
0	0	0	0	0	0
0	0	1	0	1	0
1	0	0	0	1	0
1	0	1	0	0	1
0	1	0	1	0	0
0	1	1	0	1	0
1	1	0	1	1	0
1	1	1	0	0	1

The next state values must be connected from the ROM outputs to the register inputs as shown in the figure below.