4.1-storage systems and technology

Memory as a Linear Array

A byte-addressable memory with N bytes is the logical equivalent of a C++ array, declared as

```
byte memory[N] ;  // Address ranges from 0 through (N – 1)
```

A memory of 512 MB = 512•2^{20} bytes = 2^{29} bytes and the memory is byte-addressable, so N = 512•1048576 = 536,870,912.

The term "**random access**" is similar to an array in which the time to access an entry does not depend on the index. A magnetic tape is a typical **sequential access device**.

There are two major types of random-access memory:

- **RAM** (**R**andom **A**ccess **M**emory. Read-Write Memory, "primary memory", "core memory")
- **ROM** (Read-Only Memory).

Some ROM have their contents set at time of manufacture, other types called **PROM** (Programmable ROM), can have contents **changed only once by special devices called PROM Programmers**. There is also **EPROM** (Erasable PROM), in which the contents can be **erased and rewritten many times**.

Registers associated with the memory system

All memory types, both RAM and ROM can be characterized by two registers and a number of control signals. Consider a memory of 2^N words, each having M bits. Then

- MAR (**Memory Address Register**) is an N-bit register used to specify the memory address
- MBR (**Memory Buffer Register**) is an M-bit register used to hold data to be written to the memory or just read from the memory. This register is also called the MDR (Memory Data Register).

We need two control signals to specify the three options for a RAM unit. One standard set is

$\overline{\text{Select}}$ – the memory unit is selected. This signal is active low.

R / $\overline{\text{W}}$ – if 0 the CPU writes to memory, if 1 the CPU reads from memory.

To control RAM.

$\overline{\text{Select}}$	R / $\overline{\text{W}}$	Action
1	0	Memory contents are not changed.
1	1	Memory contents are not changed.
0	0	CPU writes data to the memory.
0	1	CPU reads data from the memory.

Note that when $\overline{\text{Select}}$ = 1, nothing is happening to the memory. It is not being accessed by the CPU and the contents do not change. When $\overline{\text{Select}}$ = 0, the memory is active and something happens.

To control ROM

$\overline{\text{Select}}$	Action
1	CPU is not accessing the memory.
0	CPU reads data from the memory.

Memory access time is the time required for the memory to access the data; specifically, it is the time between the instant that the memory address is stable in the MAR and the data are available in the MBR. Note that the table above has many access times of 70 or 80 ns. The unit "ns" stands for "nanoseconds", one–billionth of a second.

Memory cycle time is the minimum time between two independent memory accesses. It should be clear that the cycle time is at least as great as the access time, because the memory cannot process an independent access while it is in the process of placing data in the MBR.

SRAM (Static RAM) and DRAM (Dynamic RAM)

There are a number of memory technologies that were developed in the last half of the twentieth century.

Each of static RAM and dynamic RAM may be considered to be a semiconductor memory. As such, both types are volatile, in that they lose their contents when power is shut off.

Static RAM

Static RAM (SRAM) is a memory technology based on flip-flops. SRAM has an access time of 2 – 10 nanoseconds. From a logical view, all of main memory can be viewed as fabricated from SRAM, although such a memory would be unrealistically expensive.

Dynamic RAM

Dynamic RAM (DRAM) is a memory technology based on capacitors – circuit elements that store electronic charge. Dynamic RAM is cheaper than static RAM and can be packed more densely on a computer chip, thus allowing larger capacity memories. DRAM has an access time in the order of 60 – 100 nanoseconds, slower than SRAM.

The Idea of Address Space

We now must distinguish between the idea of **address space** and **physical memory**. An N–bit address will specify 2^N different addresses. The memory address is specified by a binary number placed in the Memory Address Register (MAR). The number of bits in the MAR determines the range of addresses that can be generated. N address lines can be used to specify 2^N distinct addresses, numbered 0 through $2^N - 1$. This is called the **address space** of the computer.

Byte Addressing vs. Word Addressing

We have noted above that N address lines can be used to specify 2^N distinct addresses, numbered 0 through $2^N - 1$. We now ask about the size of the addressable items. We have seen that most modern computers are byte-addressable; the size of the addressable item is therefore 8 bits or one byte. There are other possibilities. We now consider the advantages and drawbacks of having larger entities being addressable.

As a simple example, consider a computer with a 16–bit address space. The machine would have 65,536 ($64K = 2^{16}$) addressable entities. The maximum memory size would depend on the size of the addressable entity.

Byte Addressable	64 KB
16-bit Word Addressable	128 KB
32-bit Word Addressable	256 KB

For a given address space, the maximum memory size is greater for the larger addressable entities. This may be an advantage for certain applications, although this advantage is reduced by the very large address spaces we now have: 32–bits is common and 64–bit address spaces are easily available. The sizes of these address spaces are quite large.

32–bit address space	4, 294, 967, 296 bytes	(4 gigabytes)
64–bit address space	about $1.8467 \bullet 10^{19}$ bytes	16 billion gigabytes.

The advantages of byte-addressability are clear when we consider applications that process data one byte at a time. Access of a single byte in a byte-addressable system requires only the issuing of a single address. In a 16–bit word addressable system, it is necessary first to compute the address of the word containing the byte, fetch that word, and then extract the byte from the two-byte word. Although the processes for byte extraction are well understood, they are less efficient than directly accessing the byte. For this reason, many modern machines are byte addressable.

Big-Endian and Little-Endian

➔ in *Gulliver's Travels* by Jonathan Swift a war over which end of a boiled egg should be broken – the big end or the little end.

Consider the 32–bit number represented by the eight–digit hexadecimal number 0x01020304, stored at location Z in memory. In all byte-addressable memory locations, this number will be stored in the four consecutive addresses Z, (Z + 1), (Z + 2), and (Z + 3). The difference between big-endian and little-endian addresses is where each of the four bytes is stored. In our example 0x01 represents bits 31 – 24,
0x02 represents bits 23 – 16,
0x03 represents bits 15 – 8, and 0x04 represents bits 7 – 0 of the word.

As a 32-bit signed integer, the number 0x01020304 can be represented in decimal notation as $1\bullet16^6 + 0\bullet16^5 + 2\bullet16^4 + 0\bullet16^3 + 3\bullet16^2 + 0\bullet16^1 + 4\bullet16^0 = 16{,}777{,}216 + 131{,}072 + 768 + 4 = 16{,}909{,}060$. For those who like to think in bytes, this is $(01)\bullet16^6 + (02)\bullet16^4 + (03)\bullet16^2 + 04$, the same result.

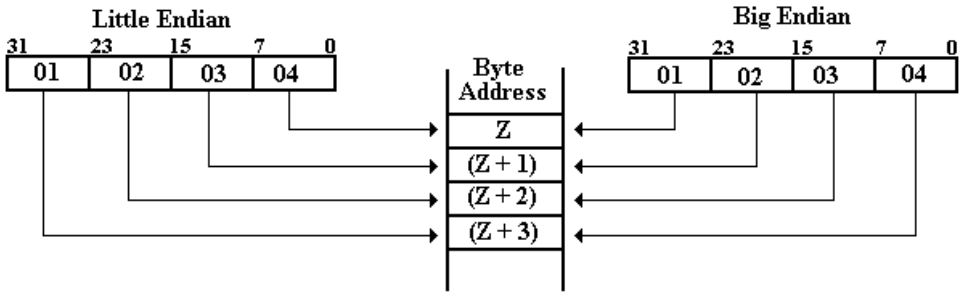
The "big end" contains the most significant digits of the number and the "little end" contains the least significant digits of the number. We now consider how these bytes are stored in a byte-addressable memory. Recall that each byte, comprising two hexadecimal digits, has a unique address in a byte-addressable memory, and that a 32-bit (four-byte) entry at address Z occupies the bytes at addresses Z, (Z + 1), (Z + 2), and (Z + 3). The hexadecimal values stored in these byte addresses are shown below.

Address	Big-Endian	Little-Endian
Z	01	04
Z + 1	02	03
Z + 2	03	02
Z + 3	04	01

Just to be complete, consider the 16–bit number represented by the four hex digits 0A0B. Suppose that the 16-bit word is at location W; i.e., its bytes are at locations W and (W + 1). The most significant byte is 0x0A and the least significant byte is 0x0B. The values in the two addresses are shown below.

Address	Big-Endian	Little-Endian
W	0A	0B
W + 1	0B	0A

The figure below shows a graphical way to view these two options for ordering the bytes copied from a register into memory. We suppose a 32-bit register with bits numbered from 31 through 0. Which end is placed first in the memory – at address Z? For big-endian, the "big end" or most significant byte is first written. For little-endian, the "little end" or least significant byte is written first.



There seems to be no advantage of one system over the other. Big–endian seems more natural to most people. Big-endian computers include the IBM 360 series, Motorola 68xxx, and SPARC by Sun.

Little-endian computers include the Intel Pentium and related computers.

The big-endian vs. little-endian debate shows in file structures when computer data are "serialized" –
Little-endian Windows BMP, MS Paintbrush, MS RTF, GIF
Big-endian Adobe Photoshop, JPEG, MacPaint

Some applications support both orientations, with a flag in the header record indicating which is the ordering used in writing the file.

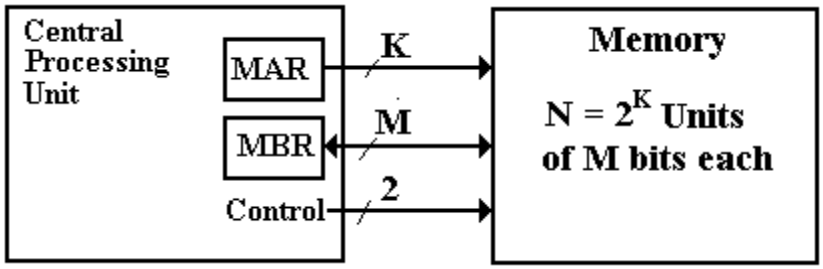
Logical View of Memory

As often is the case, we utilize a number of logical models of our memory system, depending on the point we want to make. The simplest view of memory is that of a **monolithic linear memory**; specifically a memory fabricated as a single unit (**monolithic**) that is organized as a singly dimensioned array (**linear**). This is satisfactory as a logical model, but it ignores very many issues of considerable importance.

Consider a memory in which an M–bit word is the smallest addressable unit. For simplicity, we assume that the memory contains $N = 2^K$ words and that the address space is also $N = 2^K$. The memory can be viewed as a one-dimensional array, declared something like

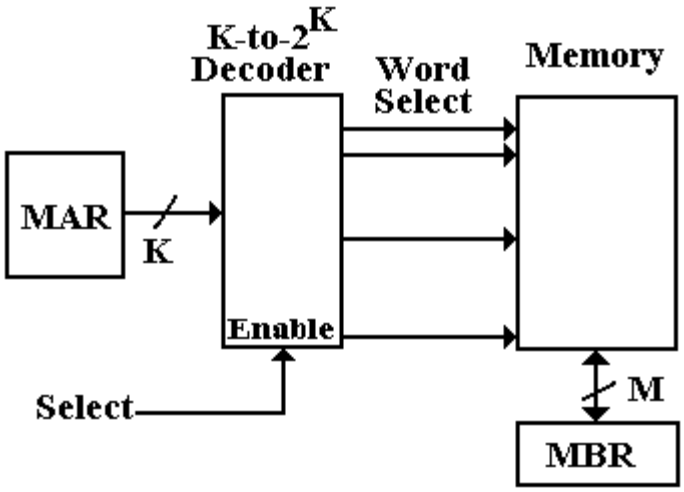
Memory : Array [0 .. (N – 1)] of M–bit word.

The **monolithic view** of the memory is shown in the following figure.



In this monolithic view, the CPU provides K address bits to access $N = 2^K$ memory entries, each of which has M bits, and at least two control signals to manage memory.

The **linear view** of memory is a way to think logically about the organization of the memory. This view has the advantage of being rather simple, but has the disadvantage of describing accurately only technologies that have long been obsolete. However, it is a consistent model that is worth mention. The following diagram illustrates the linear model.



There are two problems with the above model,

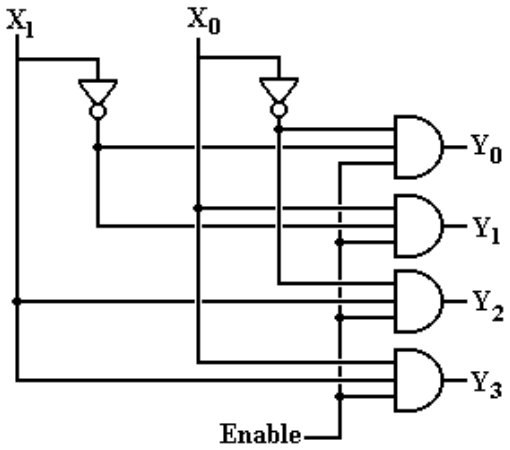
- The **speed of the memory**; its access time will be exactly that of plain variety DRAM (**d**ynamic **r**andom **a**ccess **m**emory), which is at best 50 nanoseconds. We must have better performance than that, so we go to other memory organizations.
- The "show–stopper" problem is **the design of the memory decoder**. Consider two examples for common memory sizes: 1MB (2^{20} bytes) and 4GB (2^{32} bytes) in a **byte–oriented memory**.
  A 1MB memory would use a 20–to–1,048,576 decoder, as $2^{20} = 1{,}048{,}576$.
  A 4GB memory would use a 32–to–4,294,967,296 decoder, as $2^{32} = 4{,}294{,}967{,}296$.

Neither of these decoders can be manufactured at acceptable cost using current technology.
At this point, it will be helpful to divert from the main narrative and spend some time in reviewing the structure of decoders. We shall use this to illustrate the problems found when attempting to construct large decoders. In particular, we note that larger decoders tend to be slower than smaller ones. As a result, larger memories tend to be slower than smaller ones.

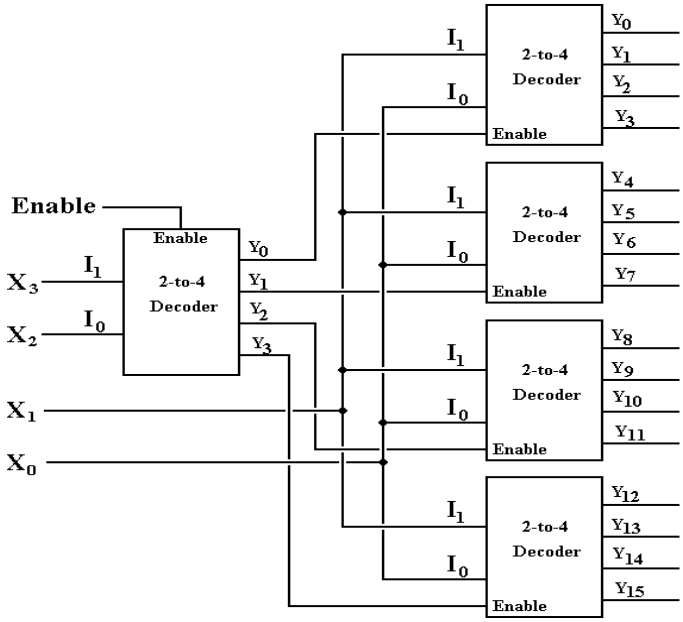
The Structure and Use of Decoders

For the sake of simplicity ,we use a 2–to–4 enabled high, active–high decoder as an example. The inferences from this figure can be shown to apply to larger decoders, both active–high and active–low, though the particulars of active–low decoders differ a bit.

An N–to–2^N active–high decoder has N inputs, 2^N outputs, and 2^N N–input AND gates. The corresponding active–low decoder would have 2^N N–input OR gates. Each of the N inputs to either design will drive $2^{N-1} + 1$ output gates. As noted above, a 1M memory would require a 20–to–1,048,576 decoder, with 20–input output gates and each input driving 524,899 gates. This seems to present a significant stretch of the technology. On the positive side, the output is available after two gate delays.



There is another way to handle this, use multiple levels of decoders. To illustrate this, consider the use of 2–to–4 decoders to build a 4–to–16 decoder.

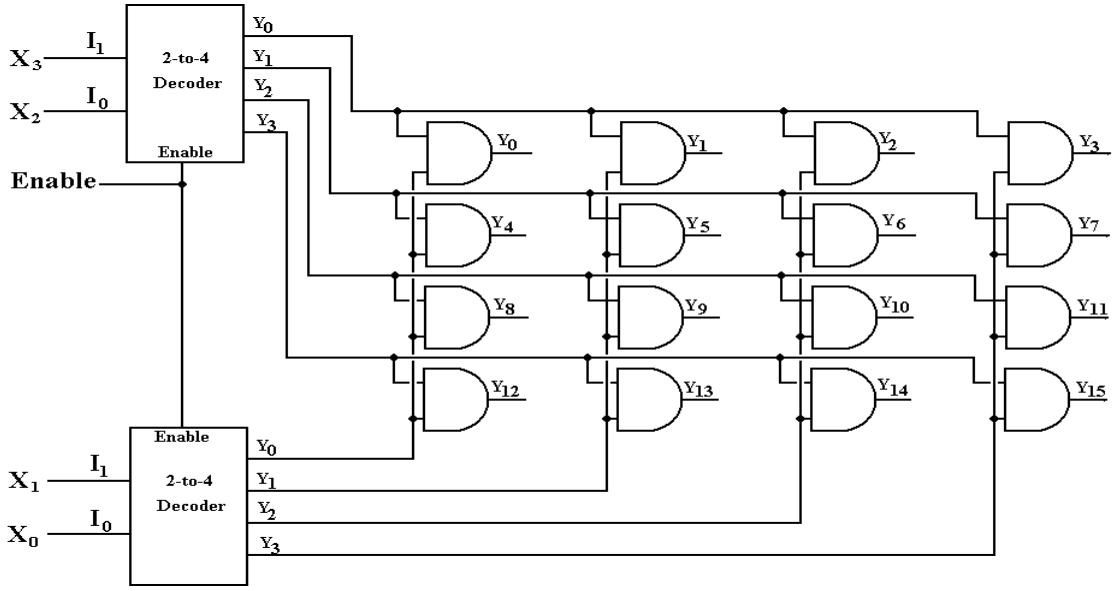


Here, each level of decoder adds two gate delays to the total delay in placing the output. For this example, the output is available 4 gate delays after the input is stable. We now investigate the generalization of this design strategy to building large decoders.

Suppose that 8–to–256 (8–to–2^8) decoders, with output delays of 2 gate delays, were stock items. A 1MB memory, using a 20–to–1,048,576 (20–to–2^{20}) decoder, would require three layers of decoders: one 4–to–16 (4–to–2^4) decoder and two 8–to–256 (8–to–2^8) decoders. For this circuit, the output is stable six gate delays after the input is stable.

A 4GB memory using a 32–to–4,294,967,296 (32–to–2^{32}) decoder, would require four levels of 8–to–256 (8–to–2^8) decoders. For this circuit, the output is stable eight gate delays after the input is stable. While seemingly fast, this does slow a memory.

There is a slight variant of the decoder that suggests a usage found in modern memory designs. It is presented here just to show that this author knows about it. This figure generalizes to fabrication of an N–to–2^N from two (N/2)–to–$2^{N/2}$ decoders. In this design, a 1MB memory, using a 20–to–1,048,576 (20–to–2^{20}) decoder, would require two decoders, each being 10–to–1,024 (10–to–2^{10}) and a 4GB memory using a 32–to–4,294,967,296 (32–to–2^{32}) decoder, would require two decoders, each being 16–to–65,536 (16–to–2^{16}).



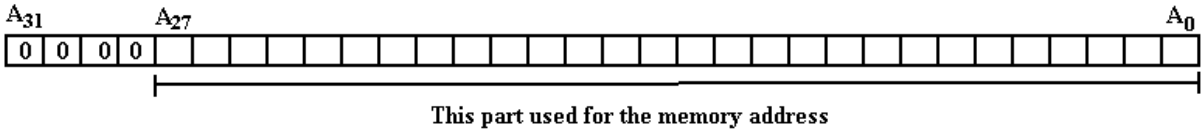
The Physical View of Memory

In a multi–level memory that uses cache memory, the goal in designing the primary memory is to have a design that keeps up with the cache closest to it, and not necessarily the CPU. All modern computer memory is built from a collection of memory chips. This design allows an efficiency boost due to the process called "**memory interleaving**".

Suppose a computer with byte-addressable memory, a 32–bit address space, and 256 MB (2^{28} bytes) of memory. Such a computer is based on this author's personal computer, with the memory size altered to a power of 2 to make for an easier example. The addresses in the MAR can be viewed as 32–bit unsigned integers, with high order bit A_{31} and low order bit A_0. Putting aside issues of virtual addressing (important for operating systems), we specify that only 28-bit addresses are valid and thus a valid address has the following form.

Later in this chapter, we shall investigate a virtual memory system that uses 32–bit addresses mapped to 28–bit physical addresses. For this discussion, we focus on the physical memory.

Here is a depiction of a 32–bit address, in which the lower order 28 bits are used to reference addresses in physical memory.

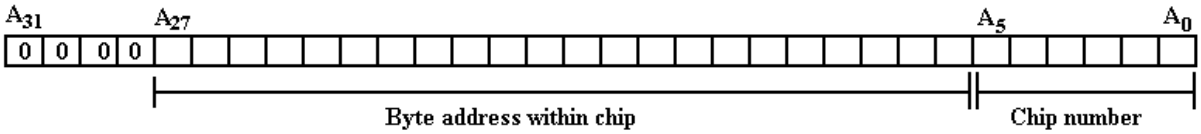


The memory of all modern computers comprises a number of chips, which are combined to cover the range of acceptable addresses. Suppose, in our example, that the basic memory chips are 4MB chips. The 256 MB memory would be built from 64 chips and the address space divided as follows:

6 bits to select the memory chip as $2^6 = 64$, and

22 bits to select the byte within the chip as $2^{22} = 4 \bullet 2^{20} = 4M$.

The question is which bits select the chip and which are sent to the chip. Two options commonly used are **high-order memory interleaving** and **low-order memory interleaving**. Other options exist, but the resulting designs would be truly bizarre. We shall consider only low-order memory interleaving in which the low-order address bits are used to select the chip and the higher-order bits select the byte. The advantage of low–order interleaving over
high–order interleaving will be seen when we consider the principle of locality.



This low-order interleaving has a number of performance-related advantages. These are due to the fact that consecutive bytes are stored in different chips, thus byte 0 is in chip 0, byte 1 is in chip 1, etc. In our example

Chip 0 contains bytes 0, 64, 128, 192, etc., and
Chip 1 contains bytes 1, 65, 129, 193, etc., and
Chip 63 contains bytes 63, 127, 191, 255, etc.

Suppose that the computer has a 64 bit–data bus from the memory to the CPU. With the above low-order interleaved memory it would be possible to read or write eight bytes at a time, thus giving rise to a memory that is close to 8 times faster. Note that there are two constraints on the memory performance increase for such an arrangement.

1) The number of chips in the memory – here it is 64.
2) The width of the data bus – here it is 8, or 64 bits.

In this design, the chip count matches the bus width; it is a balanced design.

A design implementing the address scheme just discussed might use a 6–to–64 decoder, or a pair of 3–to–8 decoders to select the chip. The high order bits are sent to each chip and determine the location within the chip. The next figure suggests the design.

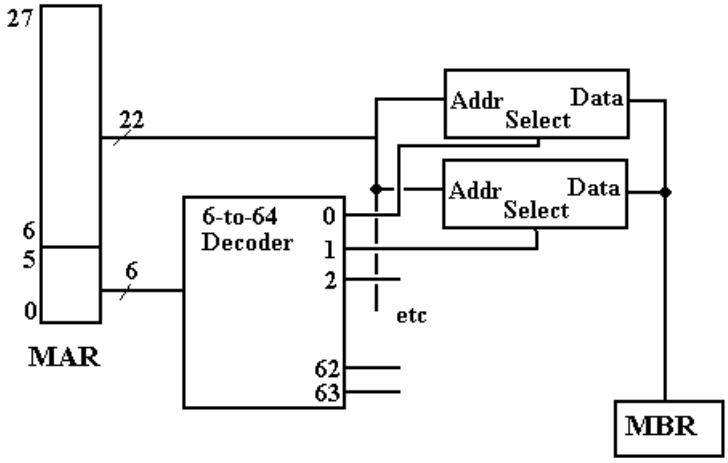


Figure: Partial Design of the Memory Unit

Note that each of the 64 4MB–chips receives the high order bits of the address. At most one of the 64 chips is active at a time. If there is a memory read or write operation active, then exactly one of the chips will be selected and the others will be inactive.

A Closer Look at the Memory "Chip"

So far in our design, we have been able to reduce the problem of creating a 32–to–2^{32} decoder to that of creating a 22–to–2^{22} decoder. We have gained the speed advantage allowed by interleaving, but still have that decoder problem. We now investigate the next step in memory design, represented by the problem of creating a workable 4MB chip.

The answer that we shall use involves creating the chip with eight 4Mb (megabit) chips. The design used is reflected in the figures below.

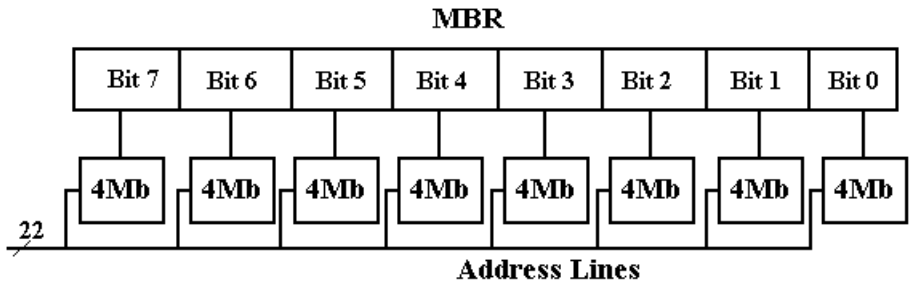


Figure: Eight Chips, each holding 4 megabits, making a 4MB "Chip"

There is an immediate advantage to having one chip represent only one bit in the MBR. This is due to the nature of chip failure. If one adds a ninth 4Mb chip to the mix, it is possible to create a simple parity memory in which single bit errors would be detected by the circuitry (not shown) that would feed the nine bits selected into the 8-bit memory buffer register.

A larger advantage is seen when we notice the decoder circuitry used in the 4Mb chip. It is logically equivalent to the 22–to–4194304 decoder that we have mentioned, but it is built using two 11–to–2048 decoders; these are at least not impossible to fabricate.

Think of the 4194304 (2^{22}) bits in the 4Mb chip as being arranged in a two dimensional array of 2048 rows (numbered 0 to 2047), each of 2048 columns (also numbered 0 to 2047). What we have can be shown in the figure below.

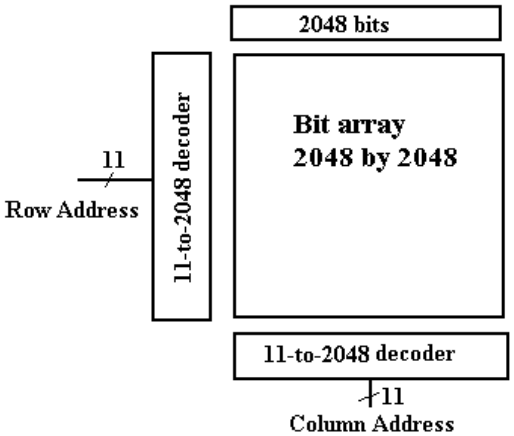
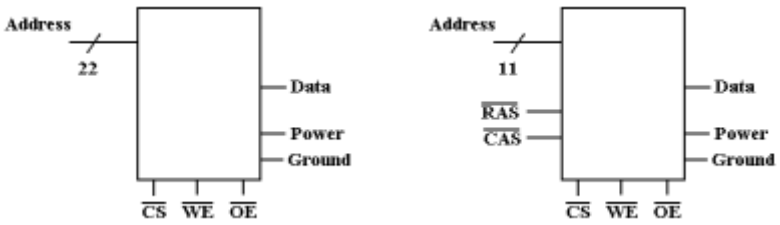


Figure: Memory with Row and Column Addresses

We now add one more feature, to be elaborated below, to our design and suddenly we have a really fast DRAM chip. For ease of chip manufacture, we split the 22–bit address into an 11–bit row address and an 11–bit column address. This allows the chip to have only 11 address pins, with two extra control (RAS and CAS – 14 total) rather than 22 address pins with an additional select (23 total). This makes the chip less expensive to manufacture.



Pin Count	Address Lines	22	11
	Row/Column	0	2
	Power & Ground	2	2
	Data	1	1
	Control	3	3
	Total	**28**	**19**

Separate row and column addresses require two cycles to specify the address.

We send the 11–bit row address first and then send the 11–bit column address. At first sight, this may seem less efficient than sending 22 bits at a time, but it allows a true speed–up. We merely add a 2048–bit row buffer onto the chip and when a row is selected; we transfer all 2048 bits in that row to the buffer at one time. The column select then selects from this
on–chip buffer. Thus, our access time now has two components:

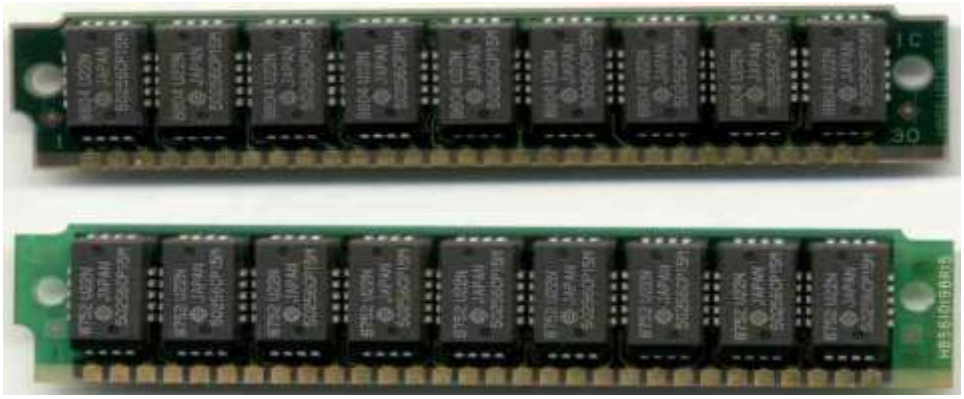
1) The time to select a new row, and
2) The time to copy a selected bit from a row in the buffer.

This design is the basis for all modern computer memory architectures.

Commercial Memory Chips

The standard memory modules are designed to plug directly into appropriately sized sockets on the motherboard. There are two main types of memory modules:.

SIMM (Single In–Line Memory Module) cards have 72 connector pins in a single row (hence the "single in–line") and are limited in size to 64 MB. Here is a picture of a SIMM card. It has 60 connectors, arranged in two rows of 30. It appears to be parity memory, as we see nine chips on each side of the card. That is one chip for each of the data bits, and a ninth chip for the parity bit for each 8–bit byte.



DIMM (Dual In–Line Memory Module) cards standardly have 168 connector pins in two rows. As of 2011, a 240–pin DIMM module with 1GB capacity was advertised on Amazon. Here is a picture of a DIMM card. It appears to be an older card, with only 256 MB capacity. Note the eight chips; this has no parity memory.

SDRAM – Synchronous Dynamic Random Access Memory

As we mentioned above, the relative slowness of memory as compared to the CPU has long been a point of concern among computer designers. One recent development that is used to address this problem is SDRAM – synchronous dynamic access memory.

References

Edward L. Bosworth, Ph.D., Design and Architecture of Digital Computers:An Introduction, TSYS School of Computer Science