# Memory Systems

This chapter begins the discussion of memory systems from the implementation of a single bit. The architecture of memory chips is then constructed using arrays of bit implementations coupled with familiar digital logic components such as decoders, transmission gates and multiplexors to access groups of bits such as words. Several alternative memory chip architectures are explored and motivated. These chips in turn are used to construct memory systems: these are systems that are designed to return specific multibit quantities such as bytes and words using memory chips of various sizes. Finally, we conclude with view of a memory system from a programmers perspective which is essentially that of a linear array of storage locations. Our programming view is now based a thorough understanding of the implementation and leads naturally to the first set of programmer (assembly language) directives that we will explore: directives for data storage.

## *Static Random Access Memory (SRAM) Cell*

The storage of a single bit of information using CMOS SRAM technology is shown in Figure 1. The basic storage component is constructed with a pair of cross coupled inverters, These inverters are embedded in a larger circuit that provides controlled connections to the inverter loop through two n-type switches. When the word line is driven high the bit lines carry the value of the bit and the value of the complement of the bit stored in the cell.

As we will see in the following sections, the term random refers to the fact that any bit in a RAM array can be accessed directly without having to first read through a sequence of other bits. Static RAM cells

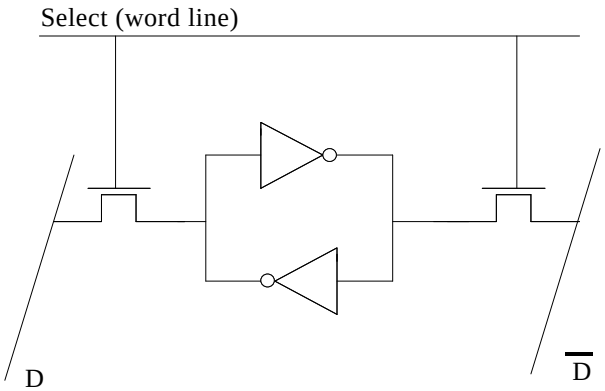



**FIGURE 1. Structure of an SRAM cell**

are capable of retaining the stored value as long as power is supplied to the circuit. This design is often referred to as a “6T design” since 6 transistors are used: two for each inverter and one each for access to the value of the bit and its complement.

## *Dynamic Random Access Memory (DRAM) Cell*

The structure a single DRAM cell is shown in Figure 2. In this design a bit of information is stored as a charge on the internal capacitor. To read the value stored in the cell the word line is driven high. The n-type transistor is turned on and the charge on the capacitor is sensed on the bit line. This implementation uses one transistor and is therefore often referred to as a “1T design”.

The process of reading the value of the bit by discharging the capacitor is a destructive operation. Therefore the bit value must be written back in amplified form following a read operation. Even if the value is not read, physical cells experience leakage of the stored charge over time. Therefore periodically the bit values stored in the cell must be read and written back. This process is referred to as *refresh* and is periodically and automatically performed by special circuits. It is this behavior that leads to the label dynamic.

Now that we have an idea how individual bits are stored we can move up the level of memory chips and study the internal architecture and operation of some standard memory components.

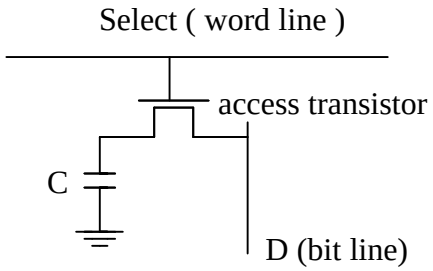



**FIGURE 2. Structure of a DRAM cell**

## *Architecture of Memory Chips*

Now let us consider the architecture and operation of memory chips. The architecture we will discuss is applicable to both SRAM and DRAM based designs. At the core of this architecture is a two-dimensional array of bits where each bit may be implemented as an SRAM or DRAM cell. A single bit in this array can be selected or *addressed* by providing the row and column indices of the location of the cell. This bit value stored in this cell can be read into a buffer from which it can be read off-chip. Let us make this sequence of steps more precise using an example and with reference to Figure 3.

Imagine we have a memory chip that stores 1 Mbits. To access any cell in this chip we must be able to identify this cell. The simplest approach would be to number all of the cells and provide the integer value or number of the cell you wish to read. Such a number is the *memory address* of the cell and providing this number is referred to as the process of addressing a cell. How many bits do we need to address 1 Mbits? We have $2^{20}$ = 1 M and therefore we need a 20 bit number which is referred to as the *address* of the cell.

However these memory cells (that store the individual bits) are not stored in a linear array that can be addressed in such a simple manner. The single bit cells are arranged in a two-dimensional array of 1024x1024 cells. All of the cells in a row share the same word line or *select* signal. Thus when a word line is asserted all of the cells in the row will drive their bit values onto the bit lines. Similarly all of the cells in a column share the same bit line. Therefore we conclude that at any given time only one cell in a column should be placing a value on this shared bit line. Addressing a row or column now requires 10 bits ($2^{10}$ = 1024). While strictly speaking a cell holds a bit value we will use the term bit and cell interchangeably.

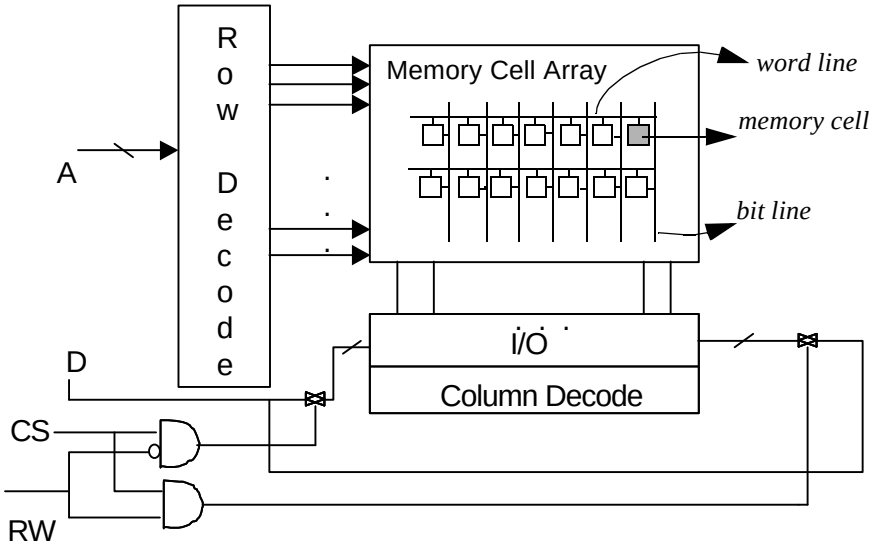



**FIGURE 3. Typical organization of a memory chi**

Now when a 20 bit address is provided we may see the following sequence of events. The most significant 10 bits if the address are used as a row address. This address is provided to a 10 bit row decoder. Each of the 1024 outputs of the row decoder are connected to one of the 1024 word lines in the array. The selected row will drive their bit values onto the corresponding 1024 bit lines. The 1024 bit values are latched into a row buffer that can be graphically thought of as residing at the bottom of the array in Figure 3. Now the 10 bit column address is used to select one of the 1024 bits from this row buffer and provide the value to the chip output. The access of data from the chip is controlled by two signals. The first is the chip select (CS) which enables the chip. The second is a signal that controls whether data is being written to the array (RW=0) or whether data is being read from the array (RW=1). The combination of CS and RW control the the tristate devices on the data signals D. The organization shown in Figure 3 uses bidirectional signals for data into and out of the chip rather than having separate input data signals and output data signals.

Now imagine that we had four identical memory planes shown in Figure 3. Each plane operating concurrently on the same 20 bit address. The result is 4 bits for every access and we would have 4 Mbit chip. Such a chip would be described as a 1Mx4 chip since we have $2^{20}$ addresses and at each

address the chip delivers 4 bits. We can conceive of alternative similar data organizations for 1 Mbit chips such as 256Kx4, 1Mx1 and 128Kx8. While the total number of bits within the chip may be the same the key distinguishing features is the number of distinct addresses that are supported and the number of bits stored at each address. With a fixed number of total bits on the chip, the number of distinct addresses provided by the chip determines the number of bits at each address.

## *Building Memory Systems*

To design a memory system we will aggregate several memory chips and combinational components. The design is determined by the types of chips available, for example 1Mx4 or 4Mx1, and the number of distinct addresses that are to be provided. Solutions are illustrated through the following examples.

### Example 1

Imagine we wish to design a 1Mx8 memory system using 1Mx4 memory chips. The total number of bits to be provided by the memory system is 8M bits. Each available 1Mx4 chip provides 4M bits and thus we will need two chips. Further we have 1M addresses but need to provided 8 bits at each address while each chip can only provide 4 bits at each address. We can organize the two chips as shown in Figure 4. One chip provides the least significant 4 bits at each address and the second chip provides the most significant four bits at the same address. The 20 bit address is provided to both chips as are the chip select and read/write control signals.

### Example 2

Now let us modify the previous example by asking for a memory system with 2M addresses and four bits at each address and using the same 1Mx4 memory chips. The total number of bits in the memory system is still the same: 8 Mbits. We observe that each chip can provide a 4 bit quantity and 1 M addresses. Thus our organization can be as shown in Figure 5. One chip services the first 1M addresses, that is addresses from 0 to $2^{20}$-1. The second chip provides the data at the remaining addresses, that is addresses from $2^{20}$ to $2^{21}$-1. However we need additional control. Only one of the two memory chips must be enabled depending on the value of the address, We can achieve this control by using the most significant bit of the address and a 1:2 decoder. The most significant bit of an address determines which of the two halves of the address range is being accessed. The outputs of the decoder are connected to the individual chip select signals and the corresponding chip will be enabled. The remaining address lines and the read/write control signal are connected to the corresponding inputs of each chip. Note that each chip still is provided with exactly 20 address bits. The memory select signal (*Msel*) is used as an enable to the memory system.

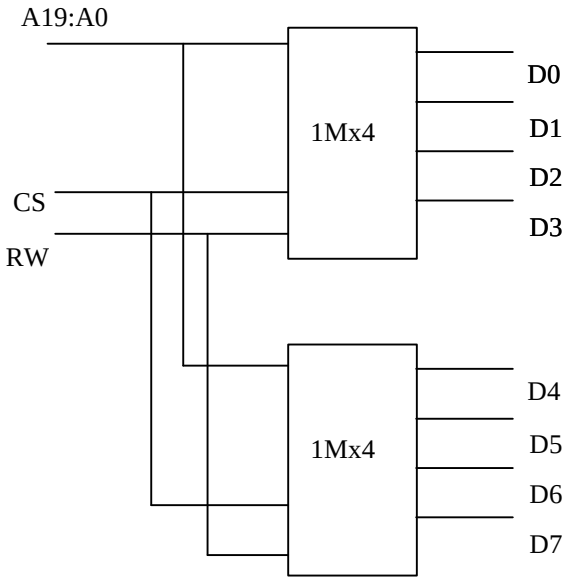



**FIGURE 4. A 1Mx8 memory system**

This approach represents a common theme. We first organize the memory chips to determine how addresses will be serviced. Some bits of the address as necessary will be used to determine which set of chips will deliver data at a specific address. These bits of the address are decoded to enable to "correct" set of memory chips. A few more examples will help clarify this approach.

### Example 3

Now we consider a memory system with 4 M addresses and four bits at each address for a total of 16 Mbits. As before using 1Mx4 chips we will need four chips. Based on the previous example the most straightforward design is one wherein each chip serves exactly one fourth of the addresses. The first chip will service addresses in the range 0 to $2^{20}$-1. The second chip will serve addresses in the range $2^{20}$ to $2^{21}$-1 and so on. The total address range is 0 to $2^{22}$-1. The two most significant bits of the address can be used to select the chip. The remaining 20 bits of address are provided to each chip as is the common read/write control. A 2:4 decoder operating on the two most significant bits of the address is used to select the chip. As before *Msel* is used as a memory system enable by serving as an enable signal for the decoder which in turn generates the chip select signals. The organization is shown in Figure 6.

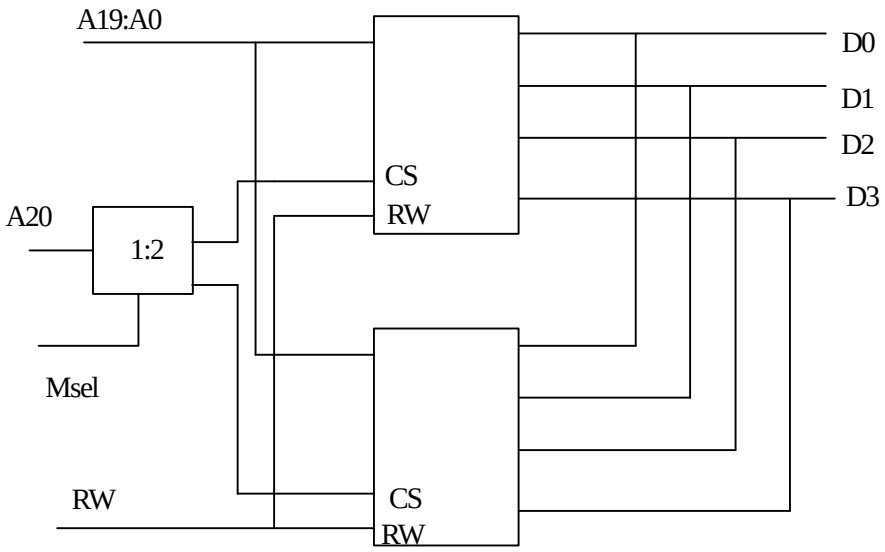



**FIGURE 5. A 2Mx4 memory system**

Note that only one chip is active at any given time. However we may have organizations where this is not necessarily the case as in the next example.

### Example 4

In this example we are interested in a memory system that will provide 2M addresses with 8 bits at each address. Using the same 1Mx4 chips we need at least two chips to provide an 8 bit quantity at any given address. We also need four chips to provide the necessary total of 16 Mbits (2M addresses and 8 bits at each address). These chips are organized in pairs as shown in Figure 7. The first two chips provide four bits each for the first 1M addresses. The second pair of chips does so similarly for the second 1 M addresses. A decoder uses the most significant bit of the address to determine which of the two pairs of chips will be selected for any specific address.

The preceding examples have illustrated several common ways for constructing memory systems of a given wordwidth using memory chips that provide multibit quantities. While this is the view that memory system designers may have programmers certainly do not think of memory in these terms. When we write programs our view of memory is quite a bit more abstract. Programming tools such as assemblers, linkers and loaders hide the details of the preceding examples from the application programmers. The most common and arguably simplest logical model of memory is that of a liner array of storage locations. This view along with programmer directives to utilize this view are discussed next.

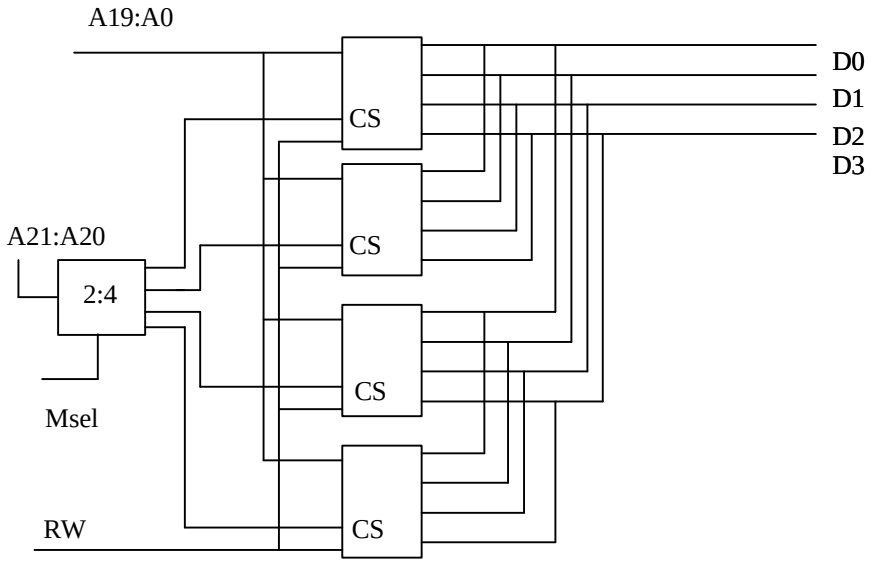



**FIGURE 6. A 4Mx4 memory system**

## *A Logical/Programmers View of Memory*

From the previous section we have seen that memory systems can be designed such that when reading from memory addresses return values stored at that address. As a user or programmer we are interested in having available a sequence of memory addresses at which we can store values. From this perspective we really do not care how this particular sequence of addresses are realized, that is, what memory chip types are used and how address bits are decoded. There is a *logical view* of memory that programmers and compilers have while a *physical implementation* is the realm of the memory systems designer. This section describes a logical view of memory.

From the construction of a memory system we know that memory is viewed as a sequence of addresses and at each address or location is stored a value. In digital systems information is stored in binary form. Therefore the first question we ask is how many bits can be stored at each memory address? We have seen examples of how to construct memory systems that return 8, 16 or 32 bit values for each address. A memory system designed to store 8-bit numbers at each address is

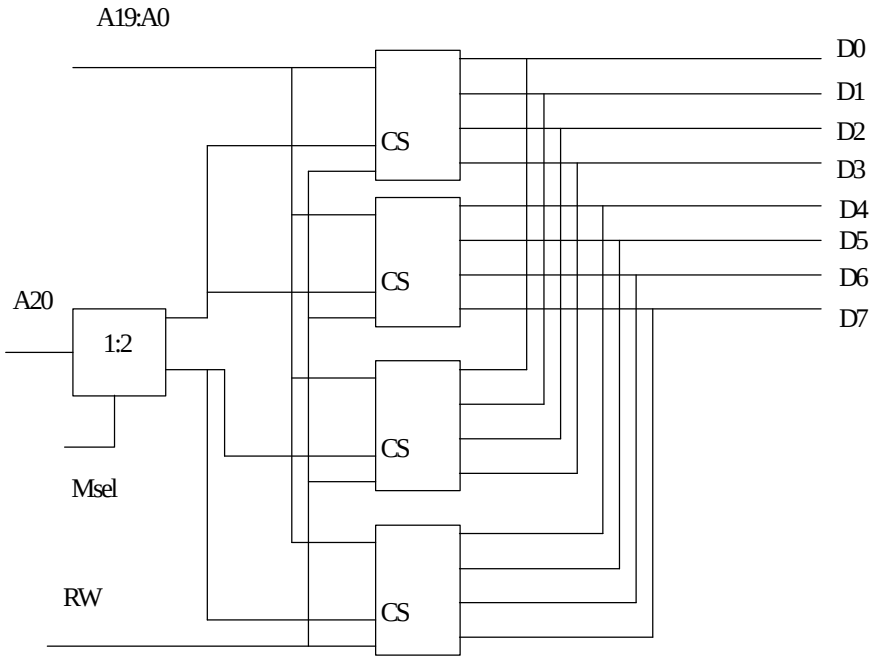



**FIGURE 7. A 2M x 8 memory system**

referred to as *byte addressed memory*. Similarly one that is designed to return a 32-bit word from each address is referred as *word addressed* memory. Although we must be careful to remember that modern microprocessors are 64-bit machines and word addressed memory implies accessing 64-bit quantities. However, it is quite common for microprocessor systems to provide byte addressed memory even though the word size may be 32 or 64 bits. Therefore we will consistently use a byte addressed memory in our examples which can be viewed as shown in Figure 8.

The contents of each memory location in the figure is an 8-bit value that shown in hexadecimal notation. The address of each memory location is shown adjacent to the location. Addresses are assumed to be 32-bit values and are also shown in hexadecimal notation. Note the distinction between addresses and the values stored at those addresses! You should keep in mind that in other models you may see each address return 16, 32 or 64 bits rather than 8 bits.

We can think of several reasons why memory should addressed in bytes. Images are organized as arrays of pixels which in black and white images can often be stored as 8-bit values. The ASCII code uses an 8-bit code and storage of character strings typically uses a sequence of byte locations. However the majority of modern high performance processors internally operate on 32 and 64 bits and therefore will

| memory contents | memory address |
|---|---|
| 0x00 | 0x10010000 |
| 0x33 | 0x10010001 |
| 0x00 | 0x10010002 |
| 0x18 | 0x10010003 |
| 0x22 | 0x10010004 |
| 0xab | 0x10010005 |
| 0xd4 | 0x100100006 |
| 0x44 | 0x100100007 |
| 0x2c | 0x100100008 |
| 0x22 | 0x100100009 |
| 0x01 | 0x10010000a |

**FIGURE 8. A logical view of byte addressed memory**

be storing and retrieving 32 and 64 bit quantities. Considering that we have a byte addressed memory what are the issues if our microprocessor performs accesses to 32-bit words?

The first issue is how are these words stored? For example, consider the need to store the 32-bit quantity 0x4400667a at address 0x10010000. The address refers to a single byte in memory however we wish to store 4 bytes at this location. The straightforward solution is to use the 4 bytes starting at address 0x10010000. After storage the memory will appear as shown in Figure 9. Note the most significant byte of the word is stored at memory location 0x10010000 and the least significant byte of the word is stored at memory location 0x10010003. This type of storage convention is referred to as *big endian* referred to in this ways since the “big end” or most significant byte of the word is stored first. We could just as easily have stored the bytes of the word in memory in the reverse order, that is, the contents of memory location 0x10010000 would have been 0x7a which is the least significant byte or “little end” of the word. Not surprisingly this storage convention is referred to as *little endian.* Different microprocessor vendors will adopt one convention or the other in the way in which words are stored. For example, Intel x86 architectures are little endian while Sun and Apple architectures are big endian. As you might imagine this places a bit of a burden on communication software that transfers data between machines that use different storage conventions since the order of bytes with each word must be reversed. In general, unless stated otherwise little endian storage convention will be used.

| | |
|---|---|
| 0x44 | 0x10010000 |
| 0x00 | 0x10010001 |
| 0x66 | 0x10010002 |
| 0x7a | 0x10010003 |

**FIGURE 9. Storage of 32-bit words in byte addressed memory**

If the word size is 32 bits, in a byte addressed memory every fourth address will be the start of a new word. Such addresses are referred to *word boundaries.* Alternatively if the word size is 64 bits each word is comprised of 8 bytes. Therefore every eighth byte will correspond to a word boundary. In general we can think of $2^k$ byte boundaries where $0 \leq k \leq n$ where $n$ is the number of bits in the address.

## *Assembler Directives and Data Layout*

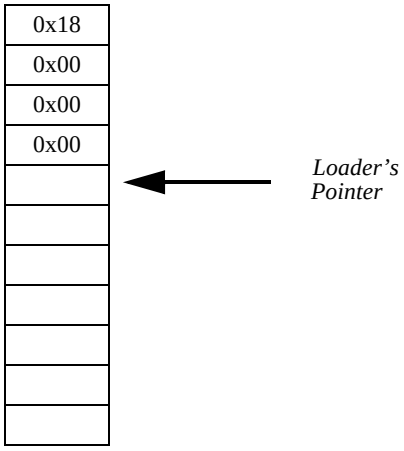
We now have view of memory as a sequence of addresses and values stored at those addresses. The discussion in this section adopts a programmers perspective in describing how we use such a logical model of memory. For our purposes we can also consider this to be equivalent to the compilers view of memory since the majority of programming in modern machines is performed in high-level languages and it is the compiler that is responsible for data storage and management.

First a few definitions. In the following examples a word is 32 bits or four bytes. All addresses are 32 bits and memory is byte addressed. From this perspective how do we describe the placement of data in memory? One approach might be the following. Imagine we have a robot that reads storage commands and places data in memory. The robot or *loader* (an operating system program in real life) starts at some fixed location in memory and continues to follow your commands for storing data. What type of commands might make sense? Consider the following commands or *directives*. Note the '.' in front of each command to signify a storage directive. We assume that the loader starts at memory location 0x10010000 although in general it can start anywhere. Finally little endian storage is assumed. Now consider the following sequence of data directives.

- .word 24: This command directs the loader to place the value 24 (or equivalently 0x18) in memory at the current word. Note that a word is 4 bytes. Therefore the value that is actually stored is the 32-

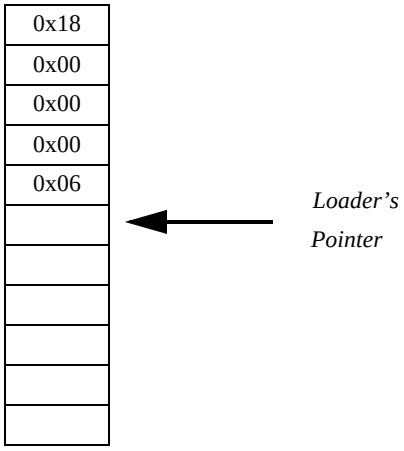
bit number 0x00000018. The loader's view of memory is now as follows (assuming little endian storage).

**MEMORY**



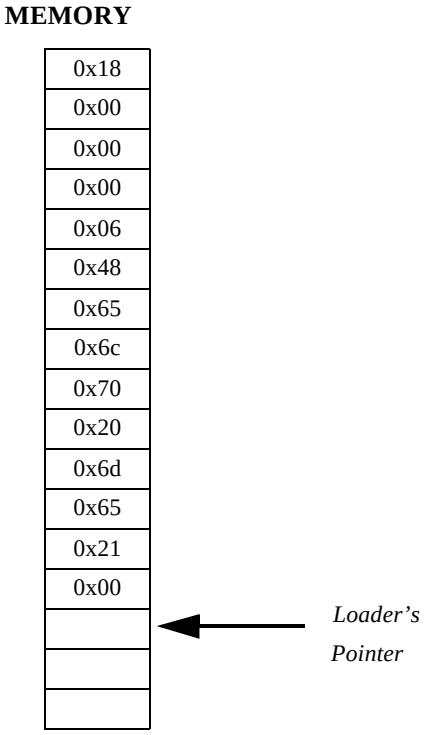

- .byte 6: This command directs the loader to store the value of 6 in the next byte in memory. The contents of memory now appears as follows.

**MEMORY**




- .asciiz "Help me!": This directive instructs the loader to store the sequence of characters "Help me" in memory. Each character is stored in a byte according the ASCII code. Remember spaces between words are a character! The quotation marks are not stored. They are used to delimit the characters. Thus the preceding character string will require 8bytes of storage. Finally there is

one more character called the null termination character that is automatically stored after the string to identify the end of the string bringing the total storage space to 9 bytes. The memory now appears as follows.




- .space n: Simply skip the next n bytes.

Consider the example sequence of directives and the resulting storage shown in Figure 10. Assume that the loader starts at address 0x10010000. The .data directive signifies the start of a sequence of data directives. The directives shown lead to the pattern of storage shown in the figure. However now suppose we wish to follow the last directive with .word 26. This will cause the storage of a word to start at location in memory which is not a word boundary. This is the issue of memory alignment, We will simplify the memory model here and require that words start on a word boundary. We can enforce this requirement with one last directive, namely .align k. This command forces the loader to move to the next $2^k$ byte boundary. In order to force the loader to move to the next word boundary we simply place the directive .align 2 after the last command. The storage map that results from this new sequence of commands is shown in Figure 11.

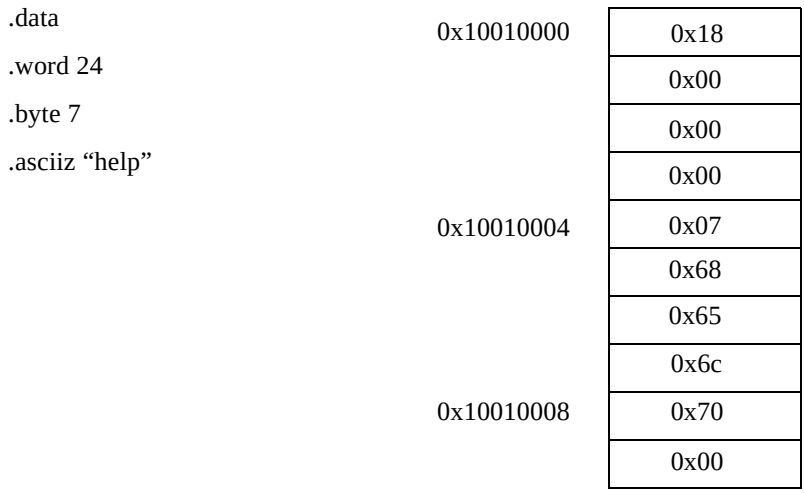



**FIGURE 10. Data directives and resulting storage pattern**

We can think now of programs that are comprised of a sequence of data directives and which are used to store data structures in memory. For example, we can envision a sequence of .word directives that would store the elements of a vector in successive memory locations. Think for a moment about the storage of a two dimensional array such as matrix. Memory is a linear array of locations. Thus two dimensional data structures must be translated to a linear array for storage purposes. One straightforward approach is store the elements of the first row in a sequence of memory locations, followed by the elements of the second row and so on. We could also store the data by columns. Such *row-major* or *column-major* ordering of data structures are automatically performed by compilers. Data directives such as those described in this section are used to describe such storage patterns.

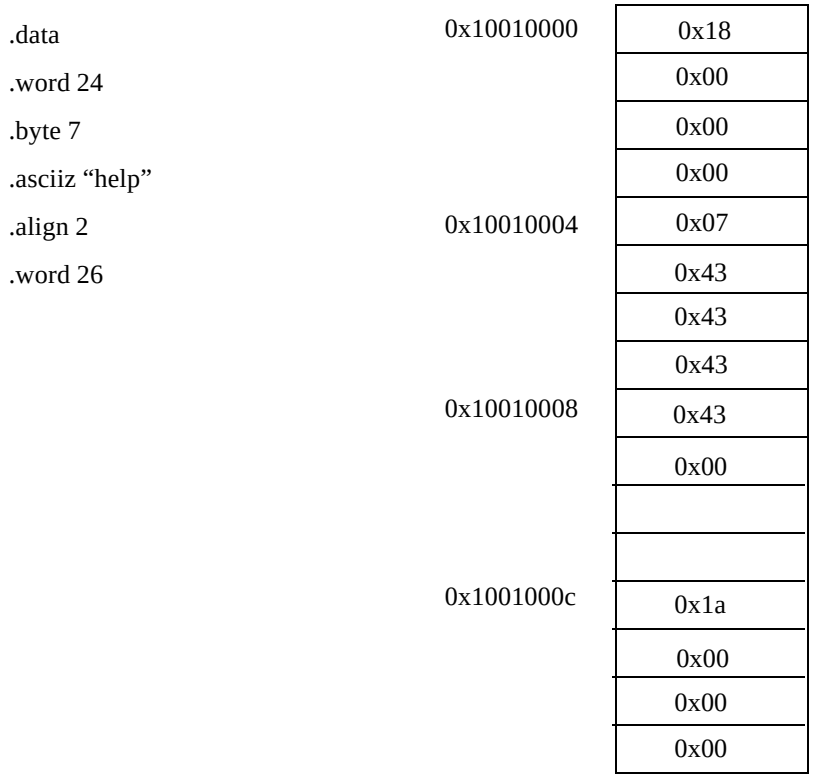



**FIGURE 11. Data directives and resulting storage pattern: a second example**