

**NAME:** MUGECHI VICTOR MAINA

**REGISTRATION NUMBER:** CT201/0037/18

**COURSE:** COMPUTER SCIENCE

**SCHOOL:** COMPUTING AND INFORMATICS

**UNIT:** ASSEMBLY LANGUAGE PROGRAMMING

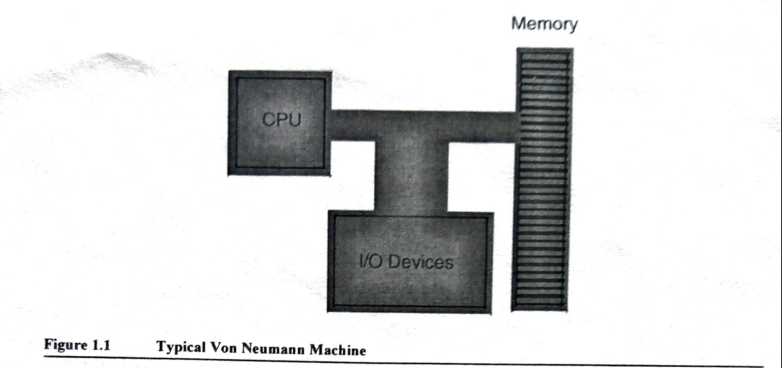
**CONTENT**: INTRODUCTION TO MACHINE ARCHITECTURE

**INTRODUCTION TO MACHINE ARCHITECTURE**

**Architecture** is the basic operation of a computer system

John Von Neumann, a pioneer in computer design, is given credit for architecture of most computers used today for example: the 80x86 family uses **Von Neumann** **Architecture** which has three major components:

* CPU (Central Processing Unit)
* Memory
* I/O(Input/ Output)



In Von Neumann Architecture machine like 80x86 families, CPU is where all actions take place like computations. Data and machines instructions reside in memory until required by CPU. To the CPU, most I/O devices look like memory because the CPU can store data to an output device and reads data from an input device. The major difference between memory and I/O locations is that I/O locations are generally associated with external devices in outside world.

**System Bus**

It connects various components of the Von Neumann Architecture machine.

There are three major buses in a Von Neumann Architecture machine:

* Address bus
* Data bus
* Control bus

**Bus** is a connection of wires on which electrical signals pass between components in the system. These buses vary from processor to processor. A typical 80x86 system component uses standard **TTL logic levels**. This means each wire on a bus uses a standard voltage level to represent 0 and 1.We will always specify 0 and 1 rather than the electrical levels because these levels vary on different processors

1. **Data Bus**

The 80x86 processor uses this bus to shuffle data between various components in a computer system**.** This bus defines the size of the processor. The size of this bus varies widely in 80x86 family.

Actually newer members of the family tend to use lower voltage signals but these remain compatible with TTL signals. TTL logic signals represent value 0 with a voltage in range 0.0V to 0.8V. It represents a value 1 with a voltage in the range of between 2.4V and 5V. If the signal on the bus line between 0.8V and 2.4V its value is indeterminate. Such a condition should only exist when a bus line is changing from one state to another.

Most people now agree that the minimum of either the number of data lines on the processor or the size of the largest general purpose register determines the processor size. Anything you can do with the smaller data bus can be done with the larger data bus as well; the larger data bus however can access data faster and in larger chunks in one memory operation.

1. **Address Bus**

The data bus on an 80x86 family processor transfer information between a particular memory location or I/O device and CPU. The only question is “What memory location or I/O device?” The address bus answers this question

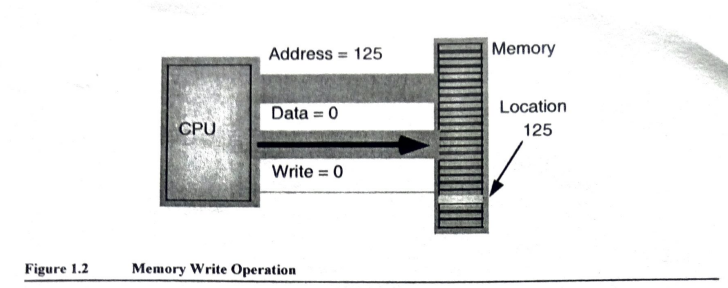
When a software wants to access some particular memory location or I/O device, it places the corresponding address on address bus. Circuitry associated with memory or I/O devices recognizes this address and instructs the memory or I/O device to read the data from or place data on data bus. In either case all other memory locations ignore the request. Only the device whose address matches the value on address bus responds. With a single address line a processor could create exactly two unique addresses: 0 and 1. With **n** address lines, the processor can provide **2n**unique addresses. Therefore the number of bits on address bus will determine maximum number of addressable memory and I/O locations. Larger address buses can address more memory.

1. **Control Bus**

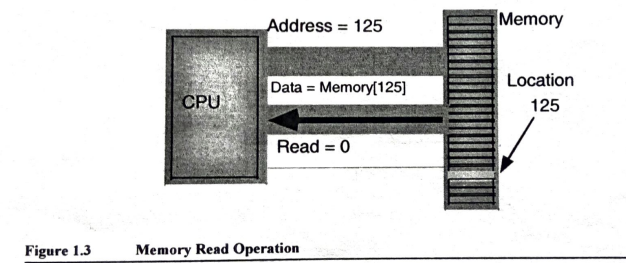
It is a collection of signals that control how a processor communicates with the rest of the system. There are two lines on the control bus read and write which specify direction of data flow. Other signals include system clock interrupt lines, status lines and so on. The read and write control lines control direction of data on data bus. When both contain logic 1, the CPU, memory and I/O are not communicating to each other. If the read line is 0 (low) then the CPU is reading data from memory that is the system is transferring data from the memory to the CPU. If the write line is 0 (low), the system transfers data from CPU to memory. The logic enable lines are another set of important control lines. These lines allow 16, 32 and 64 bit processors to deal with smaller chunks of data.

**MEMORY SUBSYSTEM**

The 80x86 processor addresses a maximum of **2n** different memory locations, where **n** is the number of bits on address bus. 80x86 processors have 20, 24, 32, 36 bit address buses with 64 bits; of course the question you should ask is “What exactly is the memory location?” The 80x86 supports byte addressable memory. Therefore the basic memory unit is a byte. So with 24, 20, 32 and 36 address lines, the 80x86 processors can address 1MB, 16MB, 4 GB, and 64 GB respectively. Think of a memory as a linear array of bytes. The address of the first byte is 0 and the address of the last byte is **2n -1**. To execute the equivalent of Pascal statement **“Memory [125]:=0;”** the CPU places value 0 on the data bus, the address 125 on address bus and asserts the write line (since the CPU is writing data to memory)



To execute the equivalent of Pascal statement **“CPU=Memory [125];”** the CPU places address 125 on the address bus asserts the read line (since the CPU is reading data from memory) and then reads resulting data from data bus.



**NOTE: The above discussion applies when accessing a single byte in memory.**

So what happens when the memory tries to access a word or double word? Since memory consists of an array of bytes how can we possibly deal with values larger than 8 bits? Different computer systems have different solutions to this problem:

* The 80x86 family deals with this problem by storing the low order byte of a word at the address specified and high order byte at the next location. Therefore a word consumes two consecutive memory addresses since a word is two bytes. Similarly a word consumes four consecutive memory locations. The address for the double word is the address of the low order byte. The remaining three bytes follow this low order byte with the high order byte appearing at the address of the double word plus three. Bytes, words and double words may begin at any valid address at the memory. However starting larger objects at an arbitrary address is not a good idea. **Note** that it is quite possible for a byte, word and double word values to overlap in memory for example: you could have a word variable beginning at address 193, a byte variable at address 194, and a double word value beginning at address 192. Those variables would all overlap.

A processor with 8 bits bus can transfer 8 bits of data at a time. Since each memory address corresponds to an 8-bit byte, this turns out to be the most convenient arrangement. The term **byte addressable memory array** means that the CPU can address memory in chunks as small as a single byte. It also means that this is the smallest unit of memory you can access at once with the processor that is: if the processor wants to access a 4 bit value it must read 8 bits then ignore the extra 4 bits. **NOTE:** Addresses are integers you cannot specify an address 125.5 for example, to fetch fewer than 8 bits.

* CPU’s with an 8 bit bus can manipulate word and double word values even though data bus is 8 bits wide. However this requires multiple memory operations because these processors can only move 8 bits of data once. To load a word you require 2 memory operations and to load a double word you require 4 memory operations. Some older x86 CPUs have 16 bit data bus. This allows these processors to access twice as much memory in same amount of time as their 8 bit brethren. These processors organize memory into two banks: **even and odd bank.**
* The 16 bit members of 80x86 family can load a word from any arbitrary address. The processor fetches low order byte of the values from address specified and high order byte from the next consecutive address. This creates a subtle problem for instance what happens when you want to access a word on an odd address? Suppose you want to read a word from location 125, the low order byte of the word comes from 125 and high order byte comes from location 126.

It turns out there are 2 problems to this approach:

* + Data bus lines 8 through 15 (high order byte) connect to the odd bank and data bus lines 0 through 7 (low order byte) connect to the even bank. Accessing the memory location 125 will transfer data to the CPU in high order byte of data bus yet we want this data in low order byte. Fortunately 80x86 CPU’s recognize this situation and automatically transfer data on D8-D15 to low order byte.
  + When accessing words we are accessing 2 separate bytes each of which has its own byte address. So the question arises “What address appears on the address bus?” The 16 bit 80x86 CPU’s always place even addresses on the bus. Even bytes always appear on data lines D0-D7 and odd bytes always appear on data lines D8-D15. If you access a word at even address, the CPU can bring the entire 16 bit chunk in one memory operation. Likewise if you access a single byte, the CPU activates the appropriate bank (using a byte enable control line). If the byte appeared at an odd address the CPU will automatically move it to the high order byte on the bus to the low order byte.

So what happens when CPU accesses a word at an odd address, the CPU cannot place the address 125 onto the address bus and read the 16 bits from memory. There are no odd addresses coming out of a 16-bit 80x86 CPU. The addresses are always even. So if you try to put 125 on the address bus, this will put 124 onto the address bus. Were you to read 16 bits at this address you would get the word at address 124(Low order byte) and 125 (High order byte) not what you would expect.

Accessing a word at an odd address requires 2 memory operations:

* + - * First the CPU must read the byte at address 125 and then it needs to read the byte at address 126.
      * Finally it needs to swap the positions of these bytes internally since both entered the CPU on the wrong half of the data bus.

Therefore accessing words at odd addresses on 16 bit processors is slower than accessing words at even addresses. By carefully arranging how you use memory, you can improve speed of your programs on CPU’s

Accessing 32 bit quantities always takes at least 2 memory operations on 16 bit processors. If you access a 32 bit quantity at an odd address, a 16 bit will require 3 memory operations to access data. The 80x86 processor with a 32 bit data bus use 4 banks of memory connected to 32 bit data bus. The address placed on the address bus is always some multiple of 4. Using various byte enable lines, the CPU can select which of the 4 byte at that address the software wants to access. As with the 16 bit processor, the CPU will automatically rearrange the bits as necessary.

With 32 bit memory interface, the 80x86 CPU can access any byte with one memory operation. If (address MOD 4) does not equal to 3, then a 32 bit CPU can access a word at that address using a single memory operation. However if the remainder is 3, then it will take 2 memory operations to access that word. This is the same problem encountered with the 16 bit processor except that it occurs half as often.

A 32 bit CPU can access a double word in a single memory operation if the address of that value is evenly divisible by 4 if not the CPU will require 2 memory operations.

However there is a performance benefit to proper data alignment. As a general rule you should always place word values at even addresses and double word values at addresses that are evenly divisible by 4. This will speed up your program. The Pentium and later processors provide 64 bits data bus and special cache memory that reduces impact of non-aligned data access. Although there may be still a penalty for accessing data at inappropriate address modern x86 CPU’s suffer from the problem less frequently than earlier CPU’s.

**I/O SUBSYSTEM**

Besides the 20, 24, 32 address lines which access the memory, the 80x86 family provides a 16 bit I/O address bus. This gives the 80x86 CPU’s 2 separate address spaces: one for memory and one for I/O operations. Lines on control bus differentiate between memory and I/O addresses. Other than separate control lines and smaller bus, I/O addressing behaves exactly like memory addressing. Memory and I/O devices share the same data bus and low order 16 lines on address bus.

There are **3 limitations to I/O subsystem on PC**:

* 80X86 CPU’s require special instructions to access I/O devices.
* Designers of PC used the best I/O locations for their own purposes forcing third party developers to use less accessible locations.
* 80x86 system can address no more than 65536(216) I/O addresses.

**HLA SUPPORT FOR DATA ALIGNMENT**

In order to write the fastest running programs you need to ensure that your data objects are properly aligned in memory. Data becomes misaligned whenever you allocate storage for different sized objects in adjacent memory locations. Since it is nearly impossible to write a (large) program that uses objects that are all the same size, some other facility is necessary in order to realign data than would normally be unaligned in memory.

Consider the following HLA variable declarations:

static

dw: dword;

b: byte;

w: word;

dw2: dword;

w2: word;

b2: byte;

dw3: dword;

The first static declaration in a program places its variables at an address that is an even multiple of 4096 bytes. Since 4096 is a power of 2 whatever variable first appears in static declaration is guaranteed to be aligned on a reasonable address. Each successive variable is allocated at an address that is the sum of sizes of all preceding variables plus the starting address. Therefore assuming the above variables are allocated at the starting address of 4096, then each variable will be allocated at the following addresses:

Start address Length

dw: dword; 4096 4

b: byte; 4100 1

w: word; 4101 2

dw2: dword; 4103 4

w2: word; 4107 2

b2: byte; 4109 1

dw3: dword; 4110 4

With the exception of the first variable (which is aligned on a 4k boundary) and the byte variables (whose alignment doesn’t matter) all those variables are misaligned in memory. The w, w2 and dw2 variables are aligned on odd addresses and dw3 variable is aligned on an even address that is not an even multiple of 4.

An easy way to guarantee that your variables are aligned on appropriate address is to put all the dword variables first, word variables second and byte variables last in declaration:

static

dw: dword;

dw2: dword;

dw3: dword;

w: word;

w2: word;

b: byte;

b2: byte;

The organization produces the following addresses in memory (again assuming the first variable is allocated at address 4096)

Start address Length

dw: dword; 4096 4

dw2: dword; 4100 4

dw3: dword; 4104 4

w: word; 4108 2

w2: word; 4110 2

b: byte; 4112 1

b2: byte; 4113 1

As you can see these variables are aligned at reasonable addresses. Unfortunately it is very possible to arrange your variables in this manner. While there are lots of technical reasons that make this alignment impossible, a good practical reason for not doing this is because it doesn’t let you organize your variable declarations by logical function that is you probably want to keep related variables next to one another regardless of size. To resolve this problem HLA provide 2 solutions;

* The first is an alignment option whenever you encounter a static section. If you follow the static keyword by integer constant inside the parentheses, HLA will align the very next variable declaration at an address that is an even multiple of specified constant. While the alignment parameter to the static directive is useful on occasion thus there are 2 problems with it:
* The first problem is that inserting so many static directives into the middle of your variable declarations tends to disrupt readability of your variable declarations. Part of this problem can be overcomed by placing a directive before every variable declaration.
* Variables appearing in separate static sections are not guaranteed to be allocated in adjacent memory locations
* Use of align directive which contains the following syntax:

**align(integer-constant);**