Why Study Assembly Language?

This depends on the decade in which you studied assembly language.

1940's You cannot study assembly language. It does not exist yet.

1950's You study assembly language because, other than raw binary machine language, it is the only way to program a computer.

1960's & 1970's

You study assembly language in order to recode time–critical parts of your code generated by a compiler.

In 1972, on a PDP–9, I could write assembly code that executed at least twice as fast as the equivalent compiled FORTRAN code.

1980's You study assembly language in order to maintain the large base of legacy code, written in assembly language.

Today Legacy code is still an issue, though a minor one.

We study assembly language in order to understand the architecture of the computer and the nature of the software services provided.

Focus on compilers and the run–time system.

Assembly Language: Traditional Goals

These focus on the assumption that the student will soon be writing programs in assembly language. Today, this is rarely true.

Here is a partial list of traditional goals for an assembly language course.

1. The binary representations used for character, integer, floating point and other decimal data. Floating point is usually given little attention.
2. Conversions from any one of these formats to any of the other formats.
3. Organization of program data into fields, records and files.
   The assembler declaratives that support record definition.
4. The basic functions of a two–pass Assembler in producing object code.
5. The basic functions of a Link Editor in producing an executable module.
6. Addressing modes in the computer, including the use of base registers.
7. How to write simple assembler programs that process character and decimal data. Here the student will write and test actual programs.
8. How to link separately assembled programs and pass data among them.

Assembly Language: Additional Goals

Study of an assembly language directs our attention to the services provided by the compiler and run–time system of a modern high–level language.

If we have to “do it ourselves”, we shall understand these services better.

Compiler and Loader Services:

Allocation of memory for variables.

Register allocation for more efficient program execution.

Resolution of external references in independently compiled programs.

Adjustment of memory addresses to reflect placement by the loader.

Run–Time Services

Creation and maintenance of dynamic data structures, such as stacks and linked lists.

Support for recursion.

Why Study IBM Mainframe Assembler?

Didactically speaking:

It is important for a student to study some assembly language, in order to understand the ISA (Instruction Set Architecture) of a sample stored–program computer.

It is desirable to study an older and simpler ISA, one without the accretions seen in the Intel series of computers.

Geographically Speaking:

Assembly language is not much used these days. For the Columbus GA area the assembler of choice is IBM 370 mainframe assembler, as it is still in use by some of the larger companies in the Columbus area.

Other Issues:

This assembly language is an older design (late 1950's and early 1960's). As such it is fairly "primitive", requiring that much be done explicitly.

Having to do things explicitly is a learning opportunity.

The Standard Model of Computation

The structure of an assembly language is dictated by the Instruction Set Architecture of the target computer.

The ISA is defined as the view of that computer's organization as it directly impacts the programmer.

The standard model for all computers today, including the IBM Mainframe, is the **stored program computer**, also called a "von Neumann machine".

Programs and data are stored in an addressable memory.

Computation is done in a separate unit, called the Central Processing Unit.

Program instructions are fetched from memory into the CPU and executed. This process is called the **"fetch/execute cycle"**.

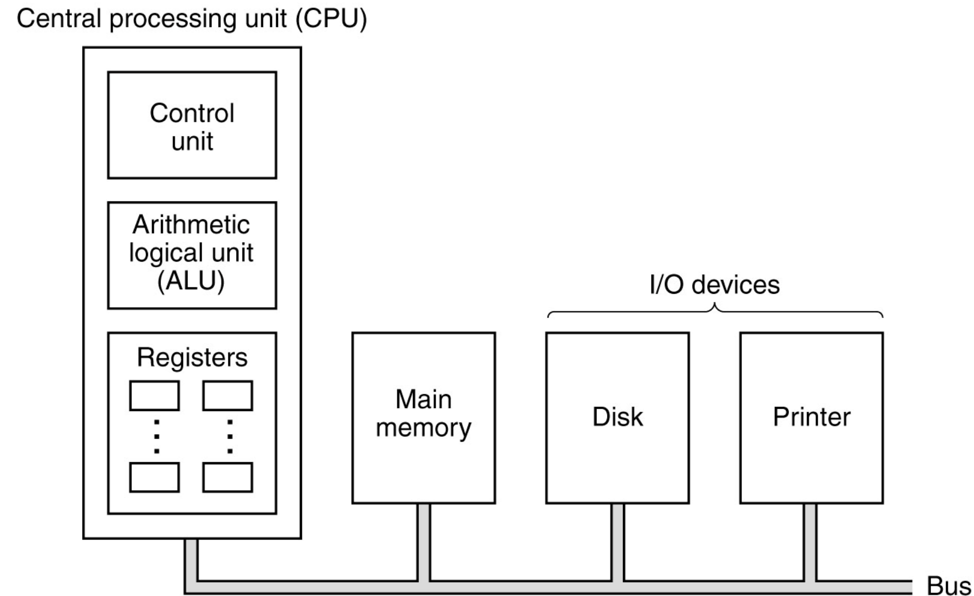
The program achieves its effect by reading data values from memory, making computations on those data, and writing new values into the memory.

All modern computers include Input/Output units to read user data into the computing process and allow output of results for human inspection.

Computer Basics and Organization

The computer has four top–level components.

1. The CPU (Central Processing Unit)
2. The Main Memory
3. Input/Output Devices, including a Hard Disk
4. A Bus Structure to facilitate communications between the other components.



Major Components Defined

The **system memory** (of which this computer has 512 MB) is used for transient storage of programs and data. This is accessed much like an array, with the **memory address** serving the function of an array index.

The **Input / Output system (I/O System)** is used for the computer to save data and programs and for it to accept input data and communicate output data. Technically the **hard drive** is an I/O device.

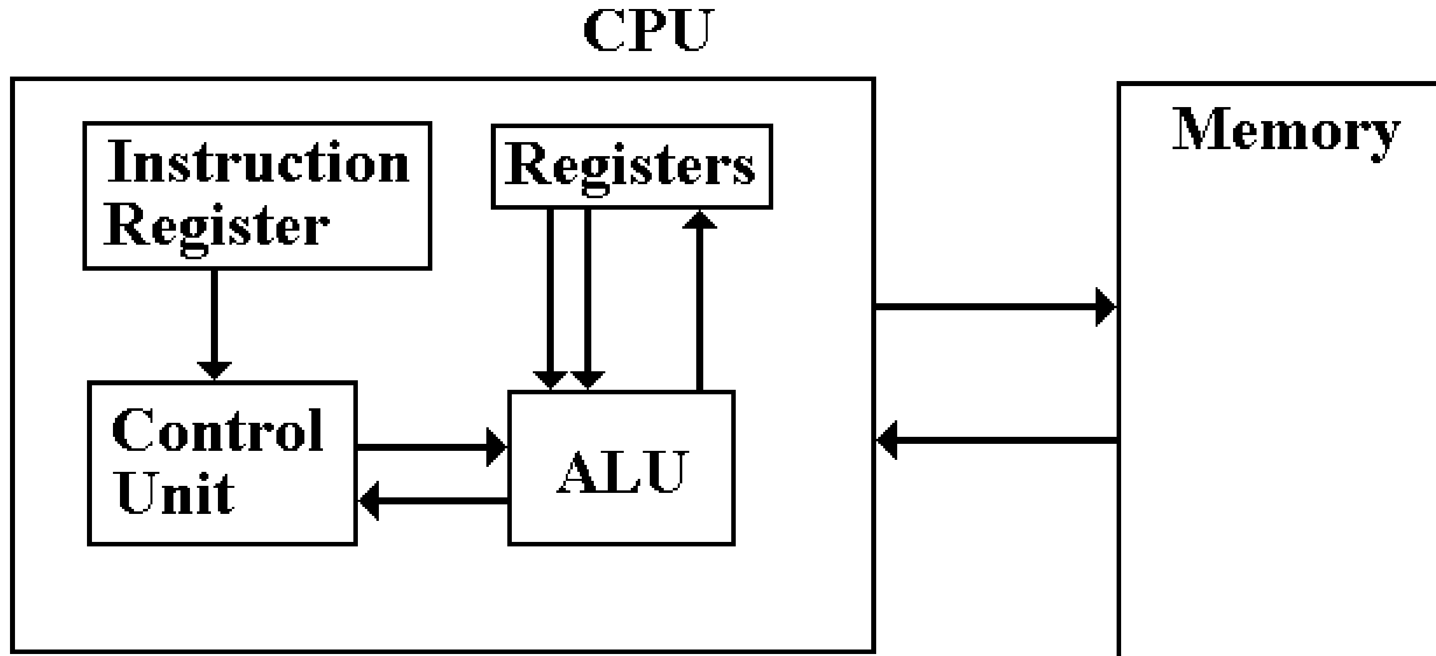
The **Central Processing Unit (CPU)** handles execution of the program.
It has four main components:

1. The **ALU (Arithmetic Logic Unit)**, which performs all of the arithmetic and logical operations of the CPU, including logic tests for branching.
2. The **Control Unit**, which causes the CPU to follow the instructions found in the assembly language program being executed.
3. The register file, which stores data internally in the CPU. There are user registers and special purpose registers used by the Control Unit.
4. A set of internal busses to allow the CPU units to communicate.

A **System Level Bus**, which allows the top–level components to communicate.

The CPU and Its Register Set

For the moment, we shall focus on two components: the CPU and memory.



The circuits used to implement the register set are faster and more costly than the circuits used to implement the main memory.

Our computational model depends heavily on the use of CPU registers.

The Memory Organization

Memory can be considered as an array of addressable items.

As in modern programming languages, this memory has 0–based indexing, so that a 4096–word memory would have addresses 0 through 4095.

The IBM Mainframe organization of memory is fairly typical, though the terminology is not standard.

Byte	8 binary bits	(this is standard)
Half Word	16 binary bits	(2 bytes)
Word	32 binary bits	(4 bytes).

NOTE: This is a memory size, not a variable type.

A word may hold several distinct data types:

32–bit signed integer,
32–bit floating point number (unlikely)
packed decimal number (also unlikely).

Declaring Variables

In higher–level programming languages, we declare variables by type and then use them. Assuming 32–bit integers, we might have:

```
int x = 0 ;
int y = 0 ;
int z = 0 ;
// More code here.
```

The standard is "declare first and then use".

In a two–pass assembler, such as the IBM Mainframe assembly language, storage for variables is usually declared at the end of the program (more later).

```
X  DS    F        // Declare a full 32–bit word
Y  DS    F        // Each full word holds 4 bytes.
Z  DS    F
```

Notes: The storage area is not initialized to any particular value.
Each of these storage locations should have an address that is a multiple of 4, to facilitate execution.

Overloading Operators

Consider the following fragment of Java code.

```
int x ;
float y, z ;
// Define x and y
z = x + y ;     // Add integer to a float.
```

This sort of type casting (integer to float) does not happen in assembly language.

In assembly language, one must do something like the following.

1. Define a new variable, call it “TEMP”.
2. Convert the integer value stored in X to its equivalent float and store the value in TEMP.
3. Add Temp to Y, giving Z.

NOTE: Variables, keywords, etc. in assembler tend to be written in UPPERCASE LETTERS. This goes back to the days of card punches, which lacked lower case characters.

Passing Arguments to Functions

Consider the following Java declaration

```
public static int theSum (int w, int x, int y, int z)
```

How are the arguments passed to the function?

How is the value returned to the calling program?

The IBM Mainframe Assembler uses one of the more common non–recursive methods for passing parameters. We study it in detail later.

Basically, create a block of memory containing the arguments and pass the address of that block.

Select a given register to be used to return function values.

Adjusting Addresses on Program Load

Consider the following assembly language program, written for the Marie.
It is given with its listing on left and its assembly on right.

000	Load	X	000	1004	// Address 004
001	Add	Y	001	3005	// Address 005
002	Store	Z	002	2006	// Address 006
003	Halt		003	7000	
X:	Hex	0	004	0000	
Y:	Hex	0	005	0000	
Z:	Hex	0	006	0000	

Were this code relocated to address 200, it would have to be changed.

200	1204	
201	3205	
202	2206	
203	7000	// No address here.

Two options: Use a linking loader that resolves addresses
Use base register addressing. IBM assembler does this.

Register Allocation

One of the bigger problems in compiler design is the allocation of registers to hold temporary results from computations.

Study of assembly language will give a greater appreciation of the use of CPU general purpose registers and how good compilers allocate them.

The one issue in the design of the compiler is the number of general purpose registers that can be allocated to hold intermediate results.

The issue arises in the evaluation of expressions of moderate complexity, for which the compiler may have to write intermediate results back to memory.

$$W = (X1 + X2) \bullet (Y1 + Y2) \bullet (Z1 + Z2 + Z3)$$

The Marie (considered next) has only one register, the Accumulator. This considerably inhibits good compiler design.

Register Allocation Example

Consider the following high–level language assignment statement.

$$W = (X1 + X2) \bullet (Y1 + Y2) \bullet (Z1 + Z2 + Z3)$$

Here is the code in the style of the Marie, which has only one register.

```
Load    Z1
Add     Z2
Add     Z3
Store   T1        // Write a temporary result back to memory.
Load    Y1
Add     Y2
Store   T2        // Write another result
Load    X1
Add     X2
Mult    T1
Mult    T2
Store   W
```