What is Assembly Language?

- Assembly language is essentially the native language of your computer. Technically the processor of your machine understands *machine code* (consisting of ones and zeroes). But in order to write such a machine code program, you first write it in assembly language and then use an *assembler* to convert it to machine code.
- However nothing is lost when the assembler does its conversion, since assembly language simply consists of mnemonic codes which are easy to remember (they are similar to words in the English language), which stand for each of the different machine code instructions that the machine is capable of executing.

By: Dalbir Singh, Computer Science Dep't

Here is an example of a short excerpt from an assembly language program:

```
MOV EAX,1
SHL EAX,5
MOV ECX,17
SUB EAX,ECX ….
```

An assembler would convert this set of instructions into a series of ones and zeros (i.e. an executable program) that the machine could understand.

What is it good for?

- Because it is extremely *low level*, assembly language can be optimized extremely well. Therefore assembly language is used where the utmost performance is required for applications.
- Assembly language is also useful for communicating with the machine at a hardware level. For this reason, it is often used for writing device drivers.
- A third benefit of assembly language is the size of the resulting programs. Because no conversion from a higher level by a compiler is required, the resulting programs can be exceedingly small.

Getting Started

- There are only two things required to get started with assembly language. An ordinary text editor to enable you to write your assembly language programs and an assembler.
- Technically you need a program called a *linker* as well. However, most decent assemblers come with a linker as part of the package.
- Basically a linker is used for large projects where there is more than one file to be assembled. Each of the assembly language files might contain references to code elements in the other files. A linker is the program that ties all the loose ends together and makes a single program out of the pieces.
- an assembler converts assembly language files to *object files* (basically machine code with some loose ends), and a linker connects all the object files together and makes a single executable program out of them.

Registers

- Assembly language programming is actually quite primitive. You'll soon realize that about all the computer's processor can do is interact with data in the computer's memory and with the data in its own internal data stores, which are called registers. Your job as a computer programmer is to write code that tells the CPU what data to move where.
- For the time being, we'll just be doing 32 bit coding, so the internal registers in the CPU, for us, will be 32 bits. Each bit is capable of storing a 1 or a 0.

- The advantage of registers over computer memory is that they are extremely fast. Time critical programming applications often try to maximize the amount of computation that can be done in the CPU registers, instead of in the computer's memory.
- The CPU has four general purpose programming registers, EAX, EBX, ECX and EDX and a number of other specialised registers.

Each of the four general purpose programming registers is 32 bits wide, but we can access the lower 16 bits of EAX (called AX) if we want to.
Similarly the lower 16 bits of EBX is called BX, etc.
Furthermore, we can access both the upper and lower 8 bits of AX (called AH and AL respectively), too, and similarly for BX, CX and DX. Refer to Diagram 1 below to see how EAX is laid out.

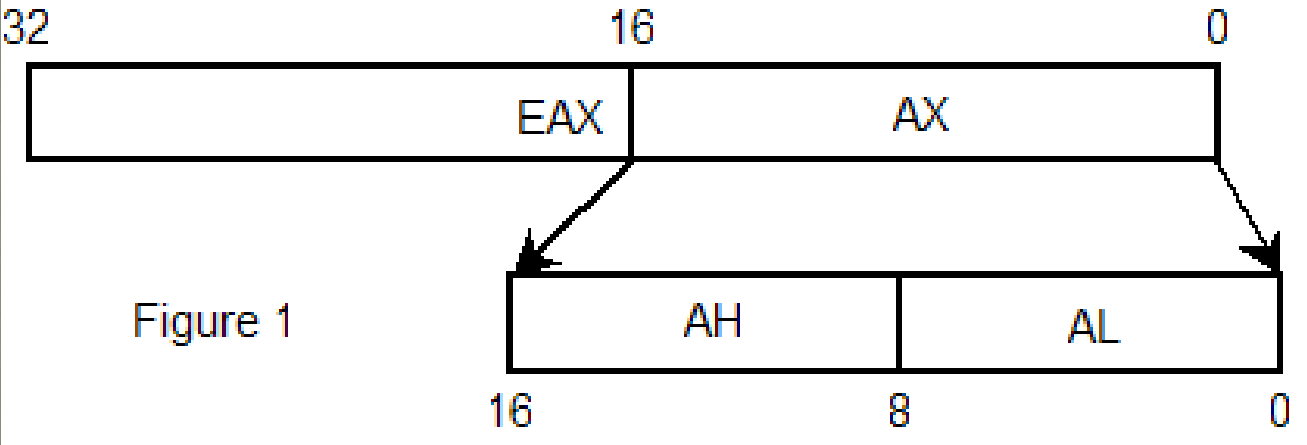

Figure 1

Moving Data into Registers

- The first thing we will want to learn about programming at the level of the CPU, is how to put data into registers. For this we use the MOV instruction. (This is the first real assembly programming instruction we've met.)
- Here is a code fragment which moves the number 47 into the EDX register:

```
MOV  EDX, 47
```

- Or, suppose that we wanted to move the hexadecimal number A4C9 into the 16 bit AX register:

```
MOV  AX, 0A4C9h
```

(Note the leading zero, and the trailing h, to denote that the number is hexadecimal.)

- Similarly, if we wished to move the binary number 01101110 into the 8 bit BH register we would type the following code:

```
MOV  BH, 01101110b
```

- To write a binary number in assembly language, we append a b to the number to indicate that it is binary, e.g. 11010110b. This corresponds to the decimal number 214. So now we have two ways to move the number 214 into a register, e.g.

  MOV AH, 11010110b

  MOV BH, 214

- To write hexadecimal numbers in assembly language, we append the number with an h and prepend it with a 0. So now we have a third way of writing the number 214 in assembly language. It is 0D6h.

  MOV CH,0D6h

Data

- Here are some examples of declaring and initializing some variables.

```
myvar1 DB 3

anothervar DW 03FAh

someval DD 721099

repeatvar DB 7 dup(12,28)

string1 DB 'This is a string'
```

The first line sets aside a single byte of memory and initialises it to the value 3. This byte of memory can then be referred to in the program by the name myvar1. Essentially the word myvar1 represents the address of the memory location. If we want to refer to the actual value stored at that address (i.e. the value 3 until something changes it), we must write[myvar1].

The second line sets aside a *word* of data (two consecutive bytes) containing the value corresponding to the given hexadecimal number.

```
myvar1 DB 3

anothervar DW 03FAh

someval DD 721099

repeatvar DB 7 dup(12,28)

string1 DB 'This is a string'
```

The third line declares and initialises a *double word* (four bytes of data).

The next line makes use of the dup operator to set aside 14 bytes of data and to initialise it to seven copies of the two bytes 12, 28. This operator is quite useful for declaring arrays of bytes or words, etc, that are initialised to zero,
e.g: `myarr DD 100 dup 0`

The next line above sets aside 16 bytes of data and sets their contents to be equal to the ASCII values corresponding to the letters of the given string. This is how we can declare strings in assembly language.

Uninitialized Data

- Sometimes we want to set aside some data without actually initialising it to any particular values.

myval DB ?

thisval DD ?

array1 RB 32

array2 RW 1000

- The first two lines declare a byte and a double word respectively, without initialising them to anything.
- The second last line reserves 32 bytes of space, but doesn't set them to anything. The final line sets aside 1000 words of data without initialising them.

Cont.... Uninitialized Data

- Note that for large arrays it is best to reserve space for them and not initialise them in the data definition, but to write a routine in code that actually initialises the values in the array (if necessary). This can make your program smaller since the initialisation is done by a small piece of code instead of a long set of explicit values contained within your program.
- Since initialised and uninitialised data are treated differently, it is sensible to separate the two into different sections. Of course sometimes you want certain pieces of data to occur in a given order, in which case this can't be done, but otherwise, separating the two different kinds of data out gives the assembler a chance to make use of the distinction.

Moving Data Between Registers

- We can move data from one register to another, so long as they are of the same size.
  - For example, to move the contents of DH into CL we write:

```
MOV CL,DH
```

Note that the source register goes on the right and the destination register goes on the left.

Movement to and From Memory

- We can use the same instruction to move data from memory to registers and vice versa. Note that we cannot move data from memory to memory with the MOV instruction.
  - If we want to move the data at a byte memory location called myvar into AH say, we would write

    MOV AH,[myvar]

  Note that the square brackets tell the machine to move the actual data into AH, not the address of the data.
  - The assembler requires that the source and destination are of matching sizes. For example, you can't move data from a variable which was declared as a byte of data, into a 16 or 32 bit register.

- However, this functionality can be easily overridden. Suppose we have a byte variable called myvar1 and we wish to move the word of data starting at that location, into the AX register. We simply type

  MOV word AX,[myvar1]

  - Of course this will take both the byte of data stored in the variable myvar1 and the byte that just happens to follow it in memory, and place the two bytes as a single word into AX.
  - The similar overrides for moving a byte and a double word of data are denoted byte and dword respectively.
- Obviously if we want to move data from a register into memory, we just put the operands in the reverse order, e.g.

  MOV [myvar1],CH

Moving Addresses into Registers

- Sometimes we actually want to move the address of a variable into a register. Since addresses are 32 bits, we can only move addresses into the 32 bit registers.
  - For example, suppose that we wished to move the address of the variable myvar2 into the EAX register. We simply type

    MOV EAX,myvar2
  - The EAX register is now a *pointer* to myvar2. It does not contain the contents of myvar2 (which may not even be a double word), but it contains the address of myvar2.

Variables as Pointers

- Once we have moved an address into a 32 bit register, we are then free to move it into a double word variable for storage.
  - For example, suppose that EAX has been loaded with the address of some memory location storing a byte of data and suppose that we have a double word variable mypoint, say, that we want to store this address in. We simply write

    MOV [mypoint],EAX
  - Now here comes the tricky part. Let's suppose we want to load the contents of the memory location now pointed to by mypoint, into the CH register. Firstly we have to retrieve the address from storage:

    MOV EBX,[mypoint]
  - Now EBX points to the location in question. Now to retrieve the byte of data at that location, we write

    MOV CH,[EBX]
  - Here the square brackets do not denote the contents of EBX itself (for which we would just write EBX), but rather, they denote the contents of the location *pointed to* by EBX.
- Although this usage of the square brackets may seem different to the earlier usage, it is in reality the same thing, since the thing inside the brackets is basically a pointer in both cases.

Loops

- There are multiple ways of creating loops. This page will describe the simplest way
  - Firstly, you need to specify how many times your code is going to loop. This is done by loading the loop count into the ECX register. Next, you start your code block with a label. This label indicates the point that your loop will return to after it has finished each iteration of the loop. Then comes the main code which you want to execute in your loop. Finally you finish your code with the LOOP instruction, giving the label name that you specified, as a parameter.

```
MOV ECX,100
mylabel:
        ;Main code block goes here.
LOOP mylabel
```