Why Learn Assembly Language? If you’re still not convinced that you should learn assembly

language, consider the following points:

• If you study computer engineering, you may likely be asked to write embedded programs.

They are short programs stored in a small amount of memory in single-purpose devices such

as telephones, automobile fuel and ignition systems, air-conditioning control systems, security systems, data acquisition instruments, video cards, sound cards, hard drives, modems,

and printers. Assembly language is an ideal tool for writing embedded programs because of

its economical use of memory.

• Real-time applications dealing with simulation and hardware monitoring require precise

timing and responses. High-level languages do not give programmers exact control over

machine code generated by compilers. Assembly language permits you to precisely specify a

program’s executable code.

• Computer game consoles require their software to be highly optimized for small code size and fast

execution. Game programmers are experts at writing code that takes full advantage of hardware

features in a target system. They often use assembly language as their tool of choice because it

permits direct access to computer hardware, and code can be hand optimized for speed.

• Assembly language helps you to gain an overall understanding of the interaction between

computer hardware, operating systems, and application programs. Using assembly language,

you can apply and test theoretical information you are given in computer architecture and

operating systems courses.

• Some high-level languages abstract their data representation to the point that it becomes awk

ward to perform low-level tasks such as bit manipulation. In such an environment, programmers will often call subroutines written in assembly language to accomplish their goal.

• Hardware manufacturers create device drivers for the equipment they sell. Device drivers

are programs that translate general operating system commands into specifific references to

hardware details. Printer manufacturers, for example, create a different MS-Windows device

driver for each model they sell. Often these device drivers contain signifificant amounts of

assembly language code.

**Accumulator**

**There is a central register in every processor called the accumulator.**

**Traditionally all mathematical and logical operations are performed on the**

**accumulator. The word size of a processor is defined by the width of its**

**accumulator. A 32bit processor has an accumulator of 32 bits.**

**General Registers (AX, BX, CX, and DX)**

**The registers AX, BX, CX, and DX behave as general purpose registers in**

**Intel architecture and do some specific functions in addition to it. X in their**

**names stand for extended meaning 16bit registers. For example AX means**

**we are referring to the extended 16bit “A” register. Its upper and lower byte**

**are separately accessible as AH (A high byte) and AL (A low byte). All general**

**purpose registers can be accessed as one 16bit register or as two 8bit**

**registers. The two registers AH and AL are part of the big whole AX. Any**

**change in AH or AL is reflected in AX as well. AX is a composite or extended**

**register formed by gluing together the two parts AH and AL.**

**The A of AX stands for Accumulator. Even though all general purpose**

**registers can act as accumulator in most instructions there are some specific**

**variations which can only work on AX which is why it is named the**

**accumulator. The B of BX stands for Base because of its role in memory**

**addressing as discussed in the next chapter. The C of CX stands for Counter**

**as there are certain instructions that work with an automatic count in the**

**CX register. The D of DX stands for Destination as it acts as the destination**

**in I/O operations. The A, B, C, and D are in letter sequence as well as depict**

**some special functionality of the register.**

**Bus Structure**A system bus consists, typically, of from about fifty to hundreds of separate lines.  
Each line is assigned a particular meaning or function. Although there are many  
different bus designs, on any bus the lines can be classified into three functional  
groups (Figure 3.16): data, address, and control lines. In addition, there may be  
power distribution lines that supply power to the attached modules.  
The **data lines**provide a path for moving data among system modules. These  
lines, collectively, are called the **data bus**. The data bus may consist of 32, 64, 128, or  
even more separate lines, the number of lines being referred to as the width of the  
data bus. Because each line can carry only 1 bit at a time, the number of lines determines how many bits can be transferred at a time. The width of the data bus is a key  
factor in determining overall system performance. For example, if the data bus is  
32 bits wide and each instruction is 64 bits long, then the processor must access the  
memory module twice during each instruction cycle.  
The **address lines**are used to designate the source or destination of the data on  
the data bus. For example, if the processor wishes to read a word (8, 16, or 32 bits)  
of data from memory, it puts the address of the desired word on the address lines.  
Clearly, the width of the **address bus**determines the maximum possible memory  
capacity of the system. Furthermore, the address lines are generally also used to  
address I/O ports. Typically, the higher-order bits are used to select a particular  
module on the bus, and the lower-order bits select a memory location or I/O port  
within the module. For example, on an 8-bit address bus, address 01111111 and  
below might reference locations in a memory module (module 0) with 128 words  
of memory, and address 10000000 and above refer to devices attached to an I/O  
module (module 1).  
The **control lines**are used to control the access to and the use of the data and  
address lines. Because the data and address lines are shared by all components,  
there must be a means of controlling their use. Control signals transmit both command and timing information among system modules. Timing signals indicate the  
validity of data and address information. Command signals specify operations to be  
performed. Typical control lines include:

• **Memory write:**causes data on the bus to be written into the addressed location  
• **Memory read:**causes data from the addressed location to be placed on the  
bus  
• **I/O write:**causes data on the bus to be output to the addressed I/O port  
• **I/O read:**causes data from the addressed I/O port to be placed on the bus  
• **Transfer ACK:**indicates that data have been accepted from or placed on the  
bus  
• **Bus request:**indicates that a module needs to gain control of the bus  
• **Bus grant:**indicates that a requesting module has been granted control of the  
bus  
• **Interrupt request:**indicates that an interrupt is pending  
• **Interrupt ACK:**acknowledges that the pending interrupt has been recognized  
• **Clock:**is used to synchronize operations  
• **Reset:**initializes all modules.

**Index Registers (SI and DI)**

**SI and DI stand for source index and destination index respectively. These**

**are the index registers of the Intel architecture which hold address of data**

**and used in memory access. Being an open and flexible architecture, Intel**

**allows many mathematical and logical operations on these registers as well**

**like the general registers. The source and destination are named because of**

**their implied functionality as the source or the destination in a special class**

**of instructions called the string instructions. However their use is not at all**

**restricted to string instructions. SI and DI are 16bit and cannot be used as**

**8bit register pairs like AX, BX, CX, and DX.**

**Instruction Pointer (IP)**

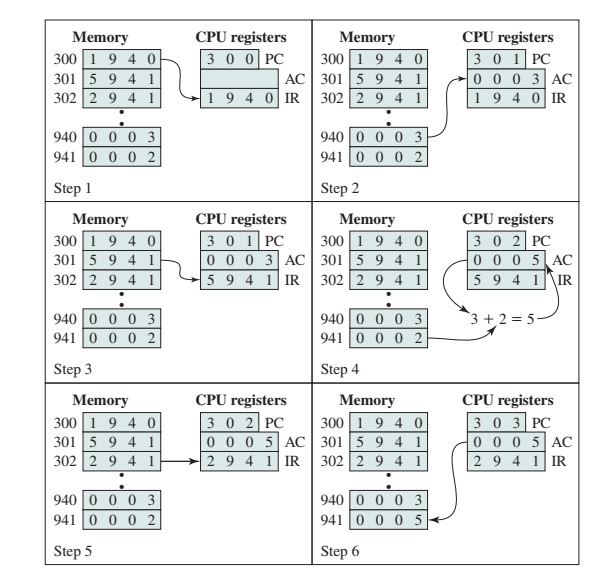
**This is the special register containing the address of the next instruction to**

**be executed. No mathematics or memory access can be done through this**

**register. It is out of our direct control and is automatically used. Playing with**

**it is dangerous and needs special care. Program control instructions change**

**the IP register.**



**Flags Register**

**The flags register as previously discussed is not meaningful as a unit**

**rather it is bit wise significant and accordingly each bit is named separately.**

**The bits not named are unused. The Intel FLAGS register has its bits**

**organized as follows:**

**C Carry**

**When two 16bit numbers are added the answer can be**

**17 bits long or when two 8bit numbers are added the**

**answer can be 9 bits long. This extra bit that won’t fit**

**in the target register is placed in the carry flag where it**

**can be used and tested.**

**P Parity**

**Parity is the number of “one” bits in a binary number.**

**Parity is either odd or even. This information is**

**normally used in communications to verify the integrity**

**of data sent from the sender to the receiver.**

**A Auxiliary Carry**

**A number in base 16 is called a hex number and can be**

**represented by 4 bits. The collection of 4 bits is called a**

**nibble. During addition or subtraction if a carry goes**

**from one nibble to the next this flag is set. Carry flag is**

**for the carry from the whole addition while auxiliary**

**carry is the carry from the first nibble to the second.**

**Sign Flag**

**A signed number is represented in its two’s complement**

**form in the computer. The most significant bit (MSB) of**

**a negative number in this representation is 1 and for a**

**positive number it is zero. The sign bit of the last**

**mathematical or logical operation’s destination is**

**copied into the sign flag.**

**Interrupt Flag**

**It tells whether the processor can be interrupted from**

**outside or not. Sometimes the programmer doesn’t**

**want a particular task to be interrupted so the**

**Interrupt flag can be zeroed for this time. The**

**programmer rather than the processor sets this flag**

**since the programmer knows when interruption is okay**

**and when it is not. Interruption can be disabled or**

**enabled by making this bit zero or one, respectively,**

**using special instructions.**

**Overflow Flag**

**The overflow flag is set during signed arithmetic, e.g.**

**addition or subtraction, when the sign of the**

**destination changes unexpectedly. The actual process**

**sets the overflow flag whenever the carry into the MSB**

**is different from the carry out of the MSB**

**Zero Flag**

**The Zero flag is set if the last mathematical or logical instruction has produced a zero in its destination.**

**Imp=> when counter register (cx ) value become zero , zero flag is automatically set)**

**If counter become zero , zero flag tells us about it**

**E.g. when loop terminate**

**Overflow Flag**

**The overflow flag is set during signed arithmetic, e.g.**

**addition or subtraction, when the sign of the**

**destination changes unexpectedly. The actual process**

**sets the overflow flag whenever the carry into the MSB**

**is different from the carry out of the MSB**

Example 2.1 (p=18)

Addtwonumber.asm

; a program to add three numbers using memory variables

[org 0x0100]

mov ax, [num1] ; load first number in ax

**mov bx, [num2] ;load second number in bx**

**add ax, bx ; accumulate sum in ax**

**mov bx, [num3] ; load third number in bx**

**add ax, bx ; accumulate sum in ax**

**mov [num4], ax ; store sum in num4**

**mov ax, 0x4c00 ; terminate program**

**int 0x21**