Experiment on 80x86 registers and memory architecture

Objectives: To visualize the memory model and register usage in the 80x86 microprocessor.

Software: The DEBUG monitor program (to be run under MSDOS).


Introduction

I.1 The 8086 basic architecture

The architecture of a CPU is its logical internal structure and resources, seen from the programmer's point of view. Inside the 8086 family CPUs are a number of registers each of which can hold a 16 bit pattern. In assembly language, each of the registers is denoted by a two letter name. There are 14 registers, and their names are:

AX BX CX DX
SP BP SI DI
CS DS SS ES
PC ST

Each of the registers are a little different and have different intended uses, but they can be grouped into some broad classes.

The general purpose registers (AX BX CX DX) are meant for general use. These are registers which hold patterns pulled in from memory, and to hold temporary results of computations. We can use these registers to hold just about anything we want.

Each of the general purpose registers can be broken down into two 8-bit registers, which have names of their own. Thus, the CX register is broken down into the CH and CL registers. The "H" and "L" stand for the high and low bytes respectively. Each general purpose register breaks down into a high/low pair.

The AX register, and it's 8 bit low half, the AL register, are somewhat special. Mainly for historical reasons, these registers are referred to as the 16 bit and 8 bit accumulators. Some operations of the CPU can only be carried out on the contents of the accumulators, and many other operations are faster when applied onto these accumulators.

Another group of registers are the segment registers (CS DS SS ES). These registers hold segment values for use in generating memory addresses. The CS, or code segment register, is used every time the 80x86 accesses memory to read an instruction pattern. The DS, or data segment register, is used for bringing data patterns in. The SS register is used to access the stack (more about the stack later). The ES is the extra segment register. Only very few special instructions use the ES register to access memory.

The pointer registers (SP BP) and index registers (DI SI) registers are used to provide indirect addressing, which is an very powerful technique for accessing memory. The SP register is used to
implement a stack in memory. Besides their special function, the BP, DI and SI registers can be used as additional general purpose registers. Although it's physically possible to directly manipulate the value in the SP register, it's best to leave it alone, since you could wipe out the stack and cause the machine to hang-up (lost control).

Finally, there are two registers which are relatively inaccessible by direct manipulation. The first is the **program counter** PC. This register always contains the offset part of the address of the next instruction to be executed. Although it is not allowed to just move values into this register, you can indirectly affect it's contents, and hence the next instruction to be executed, using control-transfer instructions such as branch, jump, call . etc. Occasionally, you will see the PC referred to as the IP, which stands for instruction pointer.

The last register is also relatively inaccessible. This is the **status register**, ST. This one has two nicknames, so watch for FL (flag register) and PSW (program status word). The status register consists of a series of single-bit flags which can affect how the 8086 works. There are special instructions which allow you to set or clear each of these flags. In addition, many instructions affect the state of the flags, depending on the outcome of the instruction. For example, one of the bits of the status register is called the Zero flag. Any operation which ends up generating a bit pattern of all 0's automatically sets the Zero flag on. Setting the flags doesn't seem to do much, until you know that there a whole set of conditional branching instructions which changes the sequence of instruction execution if the particular flags pattern they look for is matched. In assembly language, the only way to make a decision and branch accordingly is via this flag testing mechanism.

### I.2 The 80x86 real-mode memory model and memory addressing.

The CPU inside your computer can manipulate the bit patterns which make up the computer's memory. Some of the possible manipulations are copying patterns from one place to another, turning on or turning off certain bits, or interpreting the patterns as numbers and performing arithmetic operations on them. To perform any of these actions, the CPU has to know what part of memory is to be worked on. A specific location in memory is identified by it's **address**. An address is a pointer into memory. Each address points to the beginning of a byte long chunk of memory. The 80x86 CPU, in its inherent mode, has the capability to distinguish 1,048,576 different bytes of memory. It is quite clumsy to write 20 bits to get a total of 1,048,576 different addresses, and thus a memory address may be written down as a series of Hexadecimal digits. For example, the address 00410 in hexadecimal is equivalent to 00000000010000010000 in binary.

The 80x86 doesn't very happy handling 20 bits at a time. The biggest chunk that's convenient for it to use is a 16 bit word. The 80x86 actually generates 20 bit addresses as the combination of two address words, a segment word and an offset word. The combination process involves interpreting the two patterns as hexadecimal numbers and adding them. The way that two 16 bit patterns can be combined to give one 20 bit pattern is that the two patterns are added, but out of alignment by one hex digit (four bits), as in the example below:

\[
\begin{align*}
0 & 0 & 4 & 0 & \quad \text{16-bit segment address} \\
0 & 0 & 1 & 0 & \quad \text{16-bit offset address} \\
\hline
0 & 0 & 4 & 1 & 0 & \quad \text{20-bit memory address}
\end{align*}
\]

Because of this mechanism for calculating addresses, they will often be written down in what may be called **segment:offset** form. Thus, the address in above calculation could be written as:

\[
0040:0010 = 00410
\]

The contents of memory may be broken down into two broad classes. The first is data, just raw patterns of bits for the CPU to work on. The significance of the patterns is determined by what the computer is being used for at any given time. The second class of memory contents are instruction codes, or machine codes. The CPU can look at memory and interpret a bit pattern it sees there as specifying one of the 200 some fundamental operations it knows how to do. The set of code patterns
which maps to operations is called the **machine language** of the 8088. A **machine language program** consists of a series of code patterns located in consecutive memory locations, whose corresponding sequence of operations perform some useful task.

Note that there is no way for the CPU to know whether a given bit pattern is meant to be an instruction, or it is a piece of data to operate on. It is quite possible for the CPU to accidentally begin reading what was intended to be data, and interpret the patterns as instruction codes. The CPU can then become out of control, and is often described as “crashed” or “hanged”.

### I.3 Use of assembly language to represent CPU instructions

Unless you happen to be the CPU, the bit patterns which make up a machine language program can be pretty incomprehensible. For example, the instruction pattern which tells the 8088 to invert all the bits in the byte at memory address 5555H is:

```
F6 16 55 55
```

which is not very informative, although you can see the 5555H memory address in there. In the early days, the old computers were programmed by laboriously figuring out bit patterns which represented the series of instructions desired. Needless to say, this technique was incredibly tedious, and very prone to making errors. The task of figuring out the proper bit patterns for each instruction is now done by the computer itself, and **assembly language** programming was born.

Assembly language represents each of the many operations that the computer can do with a **mnemonic**, a short, easy to remember series of letters. For example, in Boolean algebra, the logical operation which inverts the state of a bit is called "not", and hence the assembly language equivalent of the preceding machine language pattern is:

```
NOTB [5555]
```

The brackets around the 5555 roughly mean "the memory location addressed by". The "B" at the end of "NOTB" indicates that we want to operate on a byte of memory, not a word.

Unfortunately, the CPU cannot understand the string of characters "NOTB". What is needed is a special program which converts the string "NOTB" into the pattern F6 16. This program is called an **assembler**. A good analogy is that an assembler program is like a machine which takes in assembly language and gives out machine language.

Typically, an assembler reads a file of assembly language and translates it one line at a time, outputting a file of machine language. Often times the input file is called the **source file** and the output file is called the **object file**. The machine language patterns produced are called the **object code**.

Also produced during the assembly process is a **listing**, which summarizes the results of the assembly process. The listing shows each line from the source file, along with the shorthand “hex code” representation of the object code produced. In the event that the assembler was unable to understand any of the source lines, it inserts error messages in the listing, pointing out the problem.

An assembler program is not used in this experiment. However, the DEBUG can perform the reverse. It can **unassembly** machine code by showing the corresponding assembly language equivalent.
The DEBUG program to monitor/change CPU registers and memory contents

The DEBUG program can be started simply by typing DEBUG at the DOS prompt, that is:

C:\> DEBUG

When the DEBUG program is started, the contents of the CPU registers are initialised to have specific values. You can display the contents of the registers by typing R at the prompt >. The starting addresses of the data segment, the code segment and the stack segment are, respectively, given by DS:0000, CS:0000 and SS:0000. You can dump their memory contents by typing D DS:0000, D CS:0000 and D SS:0000 at the prompt >.

The DEBUG program is now waiting for you to issue a command. For the time being, you only need to know the brief meanings of the commands that the DEBUG program supports:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assemble</td>
<td>A [address]</td>
</tr>
<tr>
<td>compare</td>
<td>C range address</td>
</tr>
<tr>
<td>dump</td>
<td>D [range]</td>
</tr>
<tr>
<td>enter</td>
<td>E address [list]</td>
</tr>
<tr>
<td>fill</td>
<td>F range list</td>
</tr>
<tr>
<td>go</td>
<td>G [=address] [addresses]</td>
</tr>
<tr>
<td>hex</td>
<td>M value1 value2</td>
</tr>
<tr>
<td>input</td>
<td>I port</td>
</tr>
<tr>
<td>load</td>
<td>L [address] [drive] [firstsector] [number]</td>
</tr>
<tr>
<td>move</td>
<td>M range address</td>
</tr>
<tr>
<td>name</td>
<td>N [pathname] [arglist]</td>
</tr>
<tr>
<td>output</td>
<td>O port byte</td>
</tr>
<tr>
<td>proceed</td>
<td>P [-address] [number]</td>
</tr>
<tr>
<td>quit</td>
<td>Q</td>
</tr>
<tr>
<td>register</td>
<td>R [register]</td>
</tr>
<tr>
<td>search</td>
<td>S range list</td>
</tr>
<tr>
<td>trace</td>
<td>T [-address] [value]</td>
</tr>
<tr>
<td>unassemble</td>
<td>U [range]</td>
</tr>
<tr>
<td>write</td>
<td>W [address] [drive] [firstsector] [number]</td>
</tr>
<tr>
<td>allocate expanded memory</td>
<td>XA [#pages]</td>
</tr>
<tr>
<td>deallocate expanded memory</td>
<td>XD [handle]</td>
</tr>
<tr>
<td>map expanded memory pages</td>
<td>XM [Ipg] [Fpg] [handle]</td>
</tr>
<tr>
<td>display expanded memory status</td>
<td>XS</td>
</tr>
</tbody>
</table>

Because the DEBUG program allows you to look anywhere in memory and change any memory location you want, you may sometimes "crash" the system, or the computer "hangs" and does not respond to the keyboard anymore. In such cases you may need to restart DOS by simultaneously pressing Ctrl, Alt and Delete keys, or press the hardware RESET button on the front panel of the PC.

Should you forget the DEBUG commands, you can always type ? at the prompt to display them on the screen. Detailed descriptions of the DEBUG commands can be found in the Windows manual.
A. **Registers and Memory.**

Start the DEBUG program. Type “Q” to exit. Start DEBUG again.

Type "R" to display the current status of the CPU and the current contents of the registers.

Type “R AX”. Enter “1234”. The content of register ____ is changed to 1234.

Type “R DS”. Enter “2000”. The content of the ____ register is changed to 2000.

Type "D DS:1000," to dump 80H bytes in the Data segment. What are they?

Type "F DS:1000 100 77" and then dump the 80H bytes again. Figure out what "F DS:1000 100 77" does.

Type "D 2000:0000" to dump 80H bytes of data.

Type "D 1900:0100" and then type "D 1ff0:0100" to dump the contents of 2 segments of memory. Compare them with those in the segment starting from 2000:0000. Do they match with each other? Based on your observation, conclude if one can refer to the same memory location with different combinations of segment address and offset address.

B. **Entering machine code, Single stepping, and Break point.**

Type “Q" to exit first if you are still working in DEBUG. Start DEBUG again.

Type "R" to display the current status of the CPU. What would be the next instruction to be executed? How can you know?

You may input a program for execution in two ways. In the first approach, you input the machine code of the program to the code segment directly.

Type "U" to display a sequence of the instructions in the code segment. Then type "D CS:100," to dump the data in code segment. See if the contents displayed with different commands are the same.

Input a stream of bit patterns to CS:100 by typing "E CS:100," and then type "D 1ff0:0100," to dump the contents of 2 segments of memory. Compare them with those in the segment starting from 2000:0000. Do they match with each other? Based on your observation, conclude if one can refer to the same memory location with different combinations of segment address and offset address.

Type "R" to display the current status of the CPU for future reference. Trace the program by typing "T" repeatedly until the final instruction you entered is executed. Make sure that the content of IP is 100H before tracing the program. Keep tracking the contents of the registers especially AX and IP. Also keep your eyes on the status of the flags.

Type “Q" to exit and start DEBUG again.

Now we enter a program with the second approach. This is done by typing the following assembly program after typing "A CS:100,\(\text{\textdagger}\)"

```
MOV AX,FF00\(\text{\textdagger}\)
ADD AX,01F0\(\text{\textdagger}\)
MOV BX,AX\(\text{\textdagger}\)
NEG BX\(\text{\textdagger}\)
ADD AX,BX\(\text{\textdagger}\)
```
Type "U CS:100 10A:" to get the machine code corresponding to the entered assembly program. Record the corresponding machine code. Reset the content of IP to 100H by typing "R IP:" and then "100:". Type "R:" to display the current status of the CPU for reference. Then trace the program by typing "T:" repeatedly until the final instruction you entered is executed. Keep tracking the contents of the registers and the status flags. Explain why some of the flags change. Tell what the function of the program is.

Reset the content of IP to 100H again. This time we don't trace the program step by step. We specify where the program should stop before we execute the program. This is done by typing "G 10C:".

We can also set a break point in the program. This is done by adding an instruction "INT 3" at the end of the program. After doing this, you can run the program by just typing "G:" without specifying the end of the program as before. The program will automatically stop at the break point. Try it. Don't forget to reset the IP before running the program.

The instruction "INT 3" is a special instruction and its effect is to return control to the DEBUG. Placing this instruction at a particular location will effectively prevent code beyond that point to be executed. By doing so, a breakpoint for the program under test can be set.

C. ASCII character codes, Input from Keyboard, and Output to Display.

(i) The ASCII (American Standard Code for Information Interchange) coding standard uses 7-bit binary patterns to represent characters. Textual information is normally coded in ASCII.

Type “A CS:100:" and enter the following program.

```
MOV BX,0
MOV [BX],BL
INC BX
CMP BH,1
JNE 103
INT 3
```

Type "D DS:0 100:" to dump the data in the corresponding memory segment before running the program.

Run the program to fill DS:0000-00FF. Type "D DS:0 100:" to dump the filled data. On the right side of the window, you can find the characters of the corresponding ASCII codes. Control characters which are not displayable are displayed as dots in the window. Now, can you tell how the string "How are you?" is stored in a computer?

(ii) The BIOS (Basic Input / Output System) program of the computer has a number of subroutines to handle input and output through the standard hardware, such as the keyboard and the display.

The BIOS routines can be called via special instructions called Software Interrupts. Some examples are:

- INT 16H returns in AL the ASCII code of the next key input
- INT 10H with BH = 0 moves the cursor to the row and column positions as specified by DH and DL
- INT 21H with AH = 2 prints a character according to the ASCII code in DL
- INT 21H with AH = 9 prints a string (a sequence of ASCII codes terminated by the $ sign) located by DS:DX.

(ii)(a) Enter the following code sequence into CS:0100 :-

```
MOV  AH,0
INT  16H
INT  3
JMP  100

Unassemble the program segment with the "U CS:100 110" command. You will find that the machine code for instruction "JMP 100" is "EBF9". Relative addressing is used to refer to the starting address of the first instruction. Can you figure out the meaning of "F9" in the machine code? In your opinion, is there any advantage to use relative addressing instead of absolute addressing?

Reset IP to 100H and type "G" to execute the program.

Press the "B" key. Note the content of AL: __________

Repeat typing "G" and then pressing any alphabet key once and record the content of AL. Figure out what the program does.

(ii)(b) Enter the following code sequence into CS:200.

MOV  BH,0
MOV  DL,0
MOV  DH,0
MOV  AH,2
INT  10
MOV  DX,0
MOV  AH,9
INT  21
INT  3

Enter the ASCII code of the string "Hello$" into DS:0000 by typing "E DS:0" and then entering "68 65 6C 6C 6F 21 24".

Set IP to 200 and then execute the program by typing "G".

Note what you see at the top left of the display window: __________________________________________________________________________

What does the program do? __________________________________________________________________________

D. Memory addressing.

Enter the following code sequence in memory starting at CS:0000.

MOV  AX,33
MOV  AX,[SI]
MOV  AX,[SI+2]
MOV  AX,[SI+2]
MOV  AX,[25]
MOV  AX,[BX+SI]
MOV  AX,[BX+SI-3]
MOV  AX,20[BX+SI-10]
INT  3

Various addressing modes are used in these instructions. Determine which addressing mode is used in which instruction.
Unassemble the programs. You will find that some of the instructions are interpreted as identical in the CPU. What are they? Any conclusion can you draw based on your observation?

Align the data segment with the code segment. Dump the first 40H bytes in the data segment by typing "D DS:0 40,1" and record them.

Assume the contents of BX and SI are, respectively, 12 and 17. Use what you learnt in the class to determine the results of executing the instructions.

Set IP, BX and SI to 0, 12 and 17 respectively. Trace the program and record the content of AX. Does it match the result you expected?

E. The Stack.

(i) Enter the following code sequence in memory starting at CS:0000

```
MOV BX, 22  
MOV CX, 33  
PUSH AX  
PUSH BX  
MOV AX, 0  
MOV BX, 0  
POP AX  
POP BX  
INT 3
```

Trace the program and record any changes observed. Don't forget to set the content of IP to 0.

<table>
<thead>
<tr>
<th>Step</th>
<th>AX</th>
<th>BX</th>
<th>SP</th>
<th>Stack content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Modify the program such that the modified program can restore the contents of AX and BX after being executed.

(ii) Quit DEBUG and restart DEBUG.

Enter instruction "INT 3" at CS:100.

Type "R". Record the parameters of the stack segments and the code segments:

SS:__________ SP:__________ CS:__________ IP:__________

Dump a segment of the stack segment "D SS:FFE0 FFFF,1". Record the valid data in the stack segment.

Address: ____________________________________________________________
Content: ____________________________________________________________
Executing instruction "INT 3" causes a software interrupt to the CPU. The corresponding interrupt service routine is then invoked. The last instruction of an interrupt service routine is "IRET". It informs the CPU that the service for the interrupt is completed.

Trace the program "INT 3" step by step until the instruction "IRET" is executed. Keep your eyes on the content of register SP and dump the stack segment after each step. Record what are pushed into the stack when it goes to the interrupt service routine and what are popped out from the stack when it goes out of the interrupt service routine. Why does it have to do so?

Based on the observation you have had, can you tell the interrupt vector of INT 3 and the starting address of the corresponding interrupt service routine?

Now you are an expert in assembly programming. You may try the following tasks:
1. Write a program to compute 1+2+4+8..+256 and store the result in register AX.
2. Write a program to read a character from the keyboard and then display it on the screen.
3. Write a program which can provide you the square value of the content of AH in register BX.

End of experiment.