Diyala Journal of Engineering Sciences

ISSN 1999-8716
Printed in Iraq

Vol. 02 , No. ٠١ , pp. 13-34 , June 2009

EMULATION OF THE MICROPROCESSOR INTEL 80386

SAMEERA A'AMER ABDUL-KADER

Computer Department, College of Education, Diyala University
(Received:19/8/2008 ; Accepted:24/1/2009)

ABSTRACT - Microprocessor simulation is one of the recent applications of computer design. It is used for emulating the microprocessors for some purposes such as; learning microprocessor structure and assembly language in laboratories in universities. In this research simulation for the microprocessor Intel 80386 was suggested, designed and implemented. Implementation was verified for some data transfer instructions like MOV instructions in deferent addressing modes. The designed simulation program was implemented using Visual Basic programming. Examples were tested successfully for some MOV instructions.
Keywords: ceramic, edge radius, finite element..

1. INTRODUCTION

In microcomputers and microprocessors instruction the student's job is to write programs for controlling equipment connected to a microcomputer. Instead of real devices we propose control devices simulated on the computer screen. Of course, it is important that the simulated environment is invisible for the students. Control programs, written by students, should work in the same way in the simulated environment as in a real environment.

Because of the time, expense, and complexity of constructing hardware prototypes, the computer architecture research community relies heavily on simulation to evaluate new ideas. As a result of the increasing complexity in modern computer systems, and with the corresponding difficulties of building and maintaining software tools that reflect that complexity, simulation infrastructure is now widely shared among both academic and industry researchers.

When microcomputers were invented, the most common microprocessor communication with the outside world has been through input/output ports that are located in the I/O or memory space. This facilitates constructing more advanced and more complicated microcomputer peripherals such as parallel or serial ports, timers or counters, DMA controllers and so on. Generally, internal ports of these peripherals are designed for the following functions:

- DATA ports
- CONTROL ports
- STATUS ports.

The communication through input/output ports is also applied for devices such as printers, plotters, floppy or hard discs, etc. More complicated devices, such as display graphic controllers, contain ports in addition to memories. Complex external devices have complicated electric schemes. From the programmers point of view electric details of controlled devices are not essential. For proper control it is enough to know what information to send to the CONTROL/DATA ports. The situation is the same even if external devices are causing interrupts.

In the teaching of microprocessor programming the above idea is very common: students control external real devices, connected to the microcomputers, by writing programs in a specified language (assembler, C, Basic). These programs are sending sequences of controls through input/output ports. Instead of real devices we consider the situation of controlling simulated devices displayed on the computer screen.(1)

In this research a simulation program for the microprocessor 80386 was suggested, designed and implemented. The simulation program was written in Visual Basic software. MOV instructions were implemented successfully.

2. CASE STUDIES

The last studies reflect the practical expertise that implemented of international in subject of this research. So according to importance of learning system and simulation programs, the researchers dealt with designing programs. This research deals with simulation for the microprocessor 80386, so there were studies in this subject as will reviewed bellow:

- Jacek Majewiski: In this paper control programs, written by students worked in the same way in the simulated environment as in a real environment. The paper considers as an example the preparation of a control program for the model of a plotter. The control plotter programs are written in C and 8086 assembler and compiled by real compilers: Borland C and TASM.(1)

- **Nancy A. Day**: This research presents a technique for doing symbolic simulation of microprocessor models in the functional programming language Haskell. Polymorphism and the type class system, a unique feature of Haskell are used, to write models that work over both concrete and symbolic data. It offers this approach as an alternative to using uninterpreted constants. When the full generality of rewriting is not needed, the performance of symbolic simulation by evaluation is much faster than previously reported symbolic.(2)
- **Zaatar, W. Nasr, G.E.:** In this paper a general method for defining a microprocessor emulator, applicable in any high level programming language, is presented. A complete set of communication rules is defined to describe all calls between different modules of the emulation software. The construction methodology used allows easy modifications in the emulator structure to fit different processors. Although most analysis is written in pseudo code, an actual implementation is done in Visual BASIC. In addition, the proposed emulator is implemented and compared to standard emulators while running a typical execution sequence.(3)
- **Werstein, P. Cooper, C.:** The paper describes the use of Java to develop a software based emulator for a microprocessor. This emulator is used in a second year computer architecture course to teach the basics of assembly language programming. The implementation of the emulator in Java is described in detail.(4)

3. ADVANCED MICROPROCESSORS

The 80386 microprocessor is a full 32-bit version of the earlier 8086 / 80286 16-bit microprocessors and represents a major advancement in the architecture a switch from a 16-bit architecture to a 32-bit Architecture. Along with this larger word size are many improvements and additional features, the 80386 microprocessor features multitasking memory management, Virtual memory (with or without paging) software protection and a large memory system. All software written for the early 8086/8088 and the 80286 are upward-compatible to the 80386 microprocessor. The amount of memory addressable by the 80386 is increased from the 1M bytes found in the 8086/8088 and the 16M bytes found in the 80286, to 4G bytes in the 80386. The 80386 can switch between protected mode and real mode without resetting the microprocessor. Switching from protected mode to real mode was a problem on the 80286 microprocessor because it required a hardware reset.

The 80486 microprocessor is an enhanced version of the 80386 microprocessor that executes many of its instructions in one clocking period. (5)

4. THE 80386 MICROPROCESSOR STRUCTURE

Figure (1) illustrates the pin-out of the 80386DX microprocessor. The 80386DX is packaged in a 132-pin PGA (pin grid array). Two versions of the 80386 are commonly available: the 80386DX, the other is the 80386SX, which is a reduced bus version of the 80386. Anew version of the 80386 Ex-incorporates the AT bus system, dynamic RAM controller, programmable chip selection logic, 26 address pin, 16 data pins, and 24 I/0 pins .

The 80386 DX addresses 4G bytes of memory through its 32-bit data 32-bit address. The 80386 SK. more like the 80286, addresses 16M bytes of memory with its 24-bit address bus via its 16-bit data bus.

The 80386 SK was developed after the 80386DX for applications that didn't require the full 32-bit bus version. The 80386SK is found in many personal computers that use the same basic mother board design as the 80286. At the time that the 80386SX was popular, most applications including windows required fewer than 16M bytes of memory, so the 80386SK is a popular and a less costly version of the 80386 microprocessor. Even though the 80486 has become a less expensive upgrade path for newer system, the 80386 still can be used for many applications. For example, the 80386 EX does not appear in computer systems, but it is becoming very popular in embedded applications.(5)

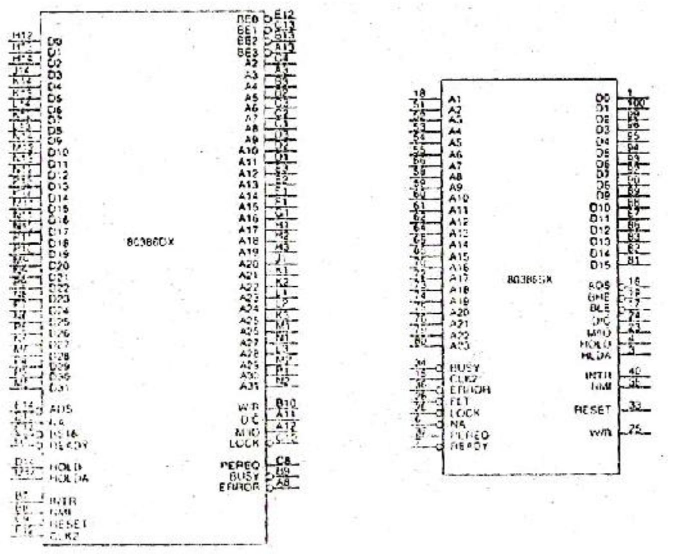


Fig. (1): the pin-outs of the 80386DX and 80386SX microprocessors.

4.1. The Memory System

The physical memory system of the 80386DX is 4G bytes in size and is addressed as such. If virtual addressing is used 64Tbytes are mapped in to the 4G bytes of physical space by the memory management unit and descriptors. (Note that virtual addressing allows a program to be larger than 4G bytes if a method of swapping with a very larger hard disk drive exists.) Figure (2) shows the organization of the 80386DX physical memory system.

The memory is divided into four 8-bit wide memory banks each containing up to 1Gbytes of memory. This 32-bit wide memory organization allows bytes, words or double words of memory data to be accessed directly.

The 80386DX transfers up to a32-bit wide number in a single memory cycle, whereas the early 8088 requires four cycles to accomplish the same transfer, and the 80286 and 80386SX require two cycles. Today, the data width is important, especially with single-precision floating-point numbers that are 32bits wide, High-level software normally uses floating-point numbers for data storage, so 32-bit memory locations speed the execution of high-level software when it is written to take advantage of this wider memory. [5]

4.2. The Input /Output System

The 80386 input / output system is the same as that found in the any Intel 8086 family microprocessors–based systems. There are 64K different bytes of I/O space available if isolated I/O is implemented. With isolated I/O the IN and OUT instruction are used to transfer I/O data between the microprocessor and I/O devices.

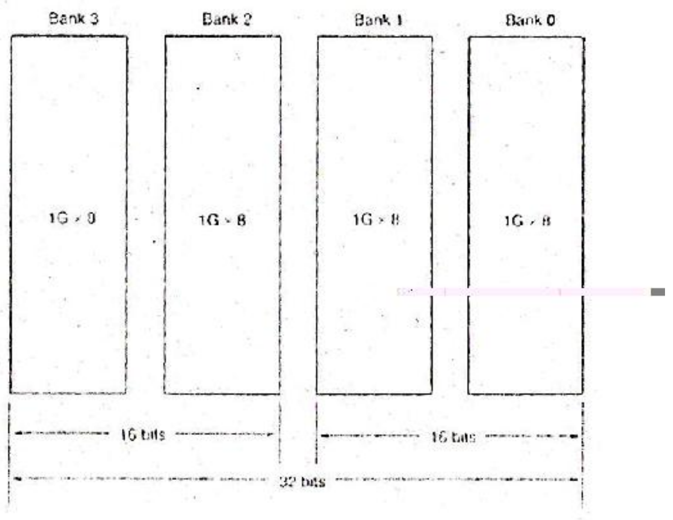


Fig. (2): the memory system for the 80386 microprocessor.

The I/O port address appears on address bus connections A15-A2. With BE3-BE0 used to select a byte, word or double word of I/O data. If memory-mapped I/O is implemented, then the number of I/O locations can be any amount up to 4G bytes. With memory-mapped I/O any instruction that transfers data between the microprocessor and memory system can be used for I/O transfers because the I/O device is treated as a memory device. Almost all 80386 system use isolated I/O because of the I/O protection scheme provided by the 80386 in protected mode operation.

Figure (3) shows the I/O map for the 80386 microprocessor. Unlike the I/O map of earlier Intel microprocessors which were 16-bits wide. The 80386 uses a full 32-bit wide I/O system divided into four banks. This is identical to the memory system, which is also divided into four banks. Most I/O transfers are 8-bits wide because we often use ASCII code (a 7-bit code) for transferring alphanumeric data between the microprocessor and printers and keyboards. This may change if Unicode a 16-bit alphanumeric code becomes common and replaces ASCII code.

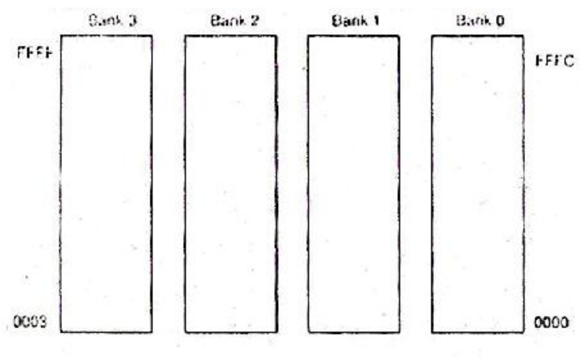


Fig. (3): The isolated I/O map for the 80386 microprocessor.

The only new feature that was added to the 80386 with respect to I/O is the I/O privilege information added to the tail end of the Tss when the 80386 is operated in protected mode.[5]

4.3. Special 80386 Registers

A new series of registers not found in earlier Intel microprocessors appears in the 80386 as control debug and test registers. Control registers CR0-CR3 control various features. DR0-DR7 facilitates debugging and registers TR6 and TR7 are used to test paging and caching.

4.3.1. Control Registers

In addition to the EFLAGS and EIP earlier there are other control registers found in the 80386. Control register 0 (CR0 is identical to the MSW (machine status word) found in the 80286 microprocessor except that it is 32-bits wide instead of 16-bits wide, additional control registers are CR1, CR2, and CR3.

Figure (4) illustrates the control register structure of the 80386. Control register CR1 is not used in the 80386 but is reserved for future products. Control register CR2 hold the linear page address of the last page accessed before a page fault interrupt. Finally Control register CR3 holds the base address of the page directory. The rightmost 12-bits of the 32-bit page table address contain zeros and combine with the remainder of the register to locate the start of the 4k-long page table. Register CR0 contains a number of special control bits in 80386.

PG selects page table translation of linear addresses in to physical addresses when PG = 1 page table translation allows any linear address to be assigned any physical memory location.

ET selects the 80387 coprocessor when ET=0 or the 80387 coprocessor when ET=1, this bit was installed because there was no 80387 available when the 80386 first appeared. In most system ET is set to indicate that an 80387 is present in the system.

TS Indicates that the 80386 has switched tasks (in protected mode changing the contents of TR places a 1 in to TS). If TS=1, a numeric coprocessor instruction causes a type 7 (coprocessor not available) interrupt.

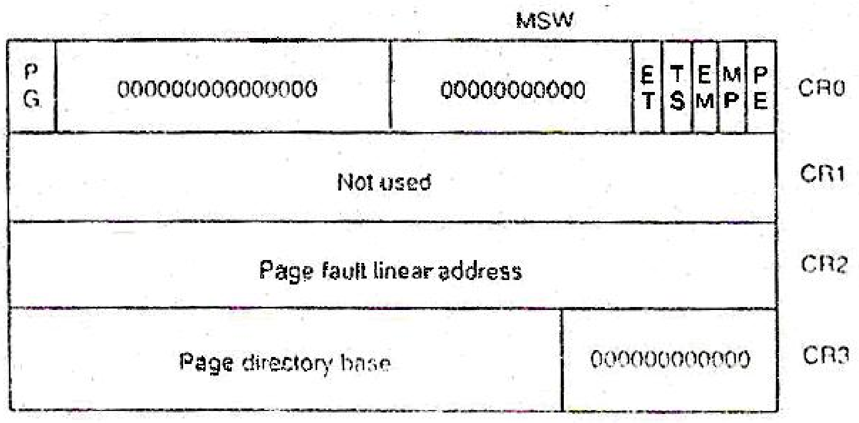


Fig. (4): The control register structure of the 80386 microprocessor.

EM Is set to cause a type 7 interrupt for each ESC instruction (Escape instructions are used to encode instructions for the 80387 coprocessor) we often use this interrupt to emulate with software.

MP Is set to indicate that the arithmetic coprocessor is present in the system.

PE Is set to select the protected mode of operation for the 80386. It may also be cleared to re-enter the real mode. This is bit can only be set in the 80286. (5)

5. The 80386 MICROPROCESSOR INSTRUCTION SET and ADDRESSING MODEL

The instruction set of a microprocessor defines the basic operations that a programmer can make the device perform. The 80386DX microprocessor provides a powerful instruction set that contains more than 150 basic instructions. The wide range of operands and addressing modes permitted for use with these instructions further expands the instruction set into many more executable instructions at the machine code level. For instance, the basic MOV instruction expands into more than 30 different machine level instructions.

The instruction set of the 8086 and 8088 microprocessors, called the basic instruction set, was enhanced in the 80286 microprocessor to implement what is known as the extended instruction set. This extended instruction set includes several new instructions and implement additional addressing modes for a law of the instructions already available in the basic Instruction set. (6)

5.1. Addressing Modes of The 80386DX Microprocessor

When the 80386DX executes an instruction, it performs the specified function on data. These data called operands, may be part of the instruction, may reside in one of the internal registers of the microprocessor may be stored at an address in memory, or may be held at an I/O port. To access these different types of operands, the 80386DX is provided with various addressing modes. An addressing mode is a method of specifying an operand. The addressing modes are categories into three types register operand addressing, immediate operand addressing, and memory operand addressing. Let us now consider in detail the addressing modes in each of these categories. (6)

5.1.1. Register Operand Addressing Mode

With the register addressing mode the operand to be accessed is specified as residing in an internal register of the80386DX. Figure (5) lists the internal registers that can be used as a source or destination operand. Notice that only the data registers can be accessed in byte, word, or double-word sizes.

An example of an instruction that uses this addressing mode is:

MOV AX, BX

Register	**Operand size**		
	Byte(Reg8)	**Word(Reg16)**	**Double Word(Reg32)**
Accumulator	AL, AH	AX	EAX
Base	BL, BH	BX	EBX
Count	CL, CH	CX	ECX
Data	DL, DH	DX	EDX
Stack pointer	-	SP	ESP
Base pointer	-	BP	EBP
Source index	-	SI	ESI
Destination index	-	DI	EDI
Code segment	-	CS	-
Data segment	-	DS	-
Stack segment	-	SS	-
E data segment	-	ES	-
F data segment	-	FS	-
G data segment	-	GS	-

Fig. (5): Direct addressing register and operand sizes.

This stands for move the word-wide contents of EBX, which is the source operand BX, to the word location in EAX. This is identified by the destination operand AX. Both the source and destination operands have been specified as the contents of internal registers of the 80386DX.(6)

5.1.2. Immediate Operand Addressing Mode

If an operand is part of the instruction instead of the contents of a register or memory location, it represents what is called an immediate operand and is accessed using the immediate addressing mode. Figure (6) shows that the operand, which can be 8 bits (Imm8), 16bits (Imml6), or 32 bits (Imm32) in length, is encoded as part of the instruction. Since the data are encoded directly into the instruction, immediate operands normally represent constant data. This addressing mode can be used only to specify a source operand.

In the instruction: MOV AL, 15 H

The source operand 15H (1516) is an example of a byte-wide immediate source operand. The destination operand, which is the contents of AL, uses register addressing. Thus this instruction employs both the immediate and registers addressing modes.(6)

Opcode	Immediate operand

Fig. (6): Instruction encoded with an immediate operand

5.2. 16-Bit Memory Operand Addressing Modes

To reference an operand in memory, the 80386DX must calculate the physical address (PA) of the operand and then initiate a read or write operation for this storage location .The 80386DX MPU is provided with a group of addressing modes known as the memory operand addressing for the purpose. The capabilities of these addressing modes have been enhanced significantly in the 80386DX compared to how they operated in the 16-bit members of the 80x86 families. In our examination of register operand addressing and immediate operand addressing, we found that the new 32-bit extensions could not be used if the objective is to write 16-bit compatible software.

The value of the EA can be specified in a variety of ways. One way is to encode the effective address of the operand directly in the instruction. This represents the simplest type of memory addressing, known as the direct addressing mode.

Figure (7) shows that an effective address can be made up from as many as three elements: the base, index, and displacement. Using these elements, the effective address calculation is made by the general formula

EA = base + index + displacement

PA= SBA: EA

PA= Segment base: Base +index + displacement

$$PA = \begin{Bmatrix} CS \\ SS \\ DS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ BP \end{Bmatrix} + \begin{Bmatrix} SI \\ DI \end{Bmatrix} + \begin{Bmatrix} 8-bit-displacement \\ 16-bit-displacement \end{Bmatrix}$$

Fig. (7): Real-mode physical and effective address computation for Memory Operand

Figure (7) also identifies the registers that can be used to hold the values of the segment base and index. For example, it tells us that any of the six-segment registers can be the source of the segment base for the physical address calculation and that the value of base for the effective address can be in either the base register (BX) or base pointer register (BP). Also identified in Figure (7) are the sizes permitted for the displacement . [6]

5.3. Direct Addressing Mode

Direct addressing mode is similar to immediate addressing in that information is encoded directly into the instruction. However, in this case the instruction opcode is followed by an effective address instead of the data. As shown in Figure (8). We find that the offset is stored in the two byte locations that follow the instruction's opcode. As the instruction is executed, the 80386DX combines 1234_{16} with 0200_{16} to get the physical address of the source operand as follows:

$$\begin{aligned} PA &= 02000_{16} + 1234_{16} \\ &= 03234_{16} \end{aligned}$$

Then it reads the word of data starting at this address, which is $BEED_{16}$ and load it into the CX register.[6]

PA = Segment base-Direct address

$$PA = \left\{ \begin{bmatrix} CS \\ DS \\ SS \\ ES \\ FS \\ GS \end{bmatrix} \right\} \quad : \quad \{Directed - address\}$$

Fig. (8): Computation of a direct memory address.

5.4. Register Indirect Addressing Mode

Register indirect addressing mode is similar to the direct addressing we just described, in that an effective address is combined with the contents of DS to obtain a physical address. However, it differs in the way the offset is specified. Figure (9) shows that this time the 16-bit EA resides in either a base register or an index register within the 80386DX. The base register can be either base register BX or base pointer register BP, and the index register can be source index register SI or destination index register DI. Another segment register can be referenced by using a segment-override prefix.

If SI contain 123416 and DS contain 020016 the result produced by executing the instruction is that the contents of the memory location at address

$$PA = 02000_{16} + 1234_{16} = 03234_{16}$$

are moved into the AX register.

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \\ FS \\ GS \end{Bmatrix} : \begin{Bmatrix} \\ BS \\ BP \\ SI \\ DI \\ \end{Bmatrix}$$

Fig. (9): Computation of an indirect memory address.

The result produced by executing this instruction, However, they differ in the way in which the physical address was generated. [(6)]

5.5. Based Addressing Mode

In the based addressing mode, the effective address of the operand is obtained by adding a direct or indirect displacement to the contents of either base register BX or base pointer register BP. As shown in Figure (10) we see that the value in the base register defines the beginning of a data structure, such as a record, in memory and the displacement selects an element of data within this structure. To access a different element in the record, the programmer simply changes the value of the displacement.

A move instruction that uses based addressing to specify the location of its destination operand is as follows:

$$\text{MOV} \quad (BX) + 1234\text{H, AL}$$

This instruction uses base register BX and direct displacement 1234H to derive the EA of the destination operand. The base addressing mode is implemented by specifying the base register in brackets followed by a + sign and direct displacement. The fetch and execution of this instruction causes the 80386DX to calculate the physical address of the destination operand from the contents of DS, BX, and the direct displacement. The result is:

$$PA = 02000_{16} + 1000_{16} + 1234_{16}$$
$$= 04234_{16}$$

Then it writes the contents of source operand AL into the storage location at 04234_{16}. The result is that ED_{16} is written into the destination memory location. Again, the default segment register for this physical address calculation is DS, but it can be changed to another segment with the segment-override prefix.

If BP is instead of BX, the calculation of the physical address is performed using the contents of the stack segment (SS) register instead of DS. This permits access to data in the stack segment of memory.(4)

$$\text{PA} = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \\ FS \\ GS \end{Bmatrix} : \begin{Bmatrix} BS \\ BP \\ SI \\ DI \end{Bmatrix} + \begin{Bmatrix} 8-bit-displacement \\ 16-bit-displacemen \end{Bmatrix}$$

Fig. (10): Computation of based address

5.6. Indexed Addressing Mode

Indexed addressing mode works in a manner similar to that of the based addressing mode. Indexed addressing mode uses the value of the displacement as a pointer to the starting point of an array of data in memory and the contents of the specified register as an index that selects the specific element in the array that is to be accessed. For instance, for the byte-size element array in figure (11) the index register holds the value n. In this way, it selects data element n in the array. The physical address is obtained from the value in a segment register, an index in the SI or DI register, and a displacement. (6)

PA = Segment base: index + displacement

$$\text{PA} = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \\ FS \\ GS \end{Bmatrix} : \begin{Bmatrix} SI \\ DI \end{Bmatrix} + \begin{Bmatrix} 8-bit-displacement \\ 16-bit-displacemen \end{Bmatrix}$$

Fig. (11): Computation of indexed address

6. EMULATORS

An emulator is a software program which enables one computer to act like another. It'll emulate the graphics, peripherals, sound, timing, etc. Ideally, the emulator will run pretty much everything the emulated computer could run, and provide a similar "experience."

First contribution to old computer emulation is Sensible Keyboard. Old computers had keyboards which weren’t like the PC keyboards used today. Maybe the left parenthesis was shift 8 and the right shift 9; and maybe the double quote "was shift 2. Most emulators map the keyboard to the PC keyboard, ignoring the PC keyboard labeling for the sake of the emulation.

Second contribution to old computer emulation is Quicktype. Even using Sensible keyboard, it may be inconvenient to type on the emulated computer. So, you’d type a text document using something like Notepad. Then you’d use my emulator’s Quicktype feature to read that document, and, Quicktype will enter the program from the file into the emulated computer, just as if you were typing it from the emulated computer’s keyboard, only quicker. It’s very useful.

Today, we may replace old computer system with a new one, because of upgrading software that were used in the old computer, and if they may not run in the new computer. The existing software is too important to discard. We easily make out this point by reflecting of Intel's microprocessors 8086, 80286, 80386 and the add-on card of IBM PC which can execute the software of IBM 360, 370.

If we give up existing software, we might suffer for long time until perfect replacement. But, sometimes the replacement which cannot provide the compatibility would be unavoidable, then we are eager to reduce labors dedicated to replacement. The better choice of new computer provides us with the better computing environment, and the smooth replacement of two computers would be desired.

To do this efficiency, it is recommended to employ the real-time emulator that has new operating environment which is compatible with the anticipate computer, and has special hardware which can execute the instructions of the old computer perfectly. We can execute the instructions of the old computer perfectly. We can convert the programs of old computer to the new one's without time consuming with this real-time emulator. As shown in figure (12).[7]

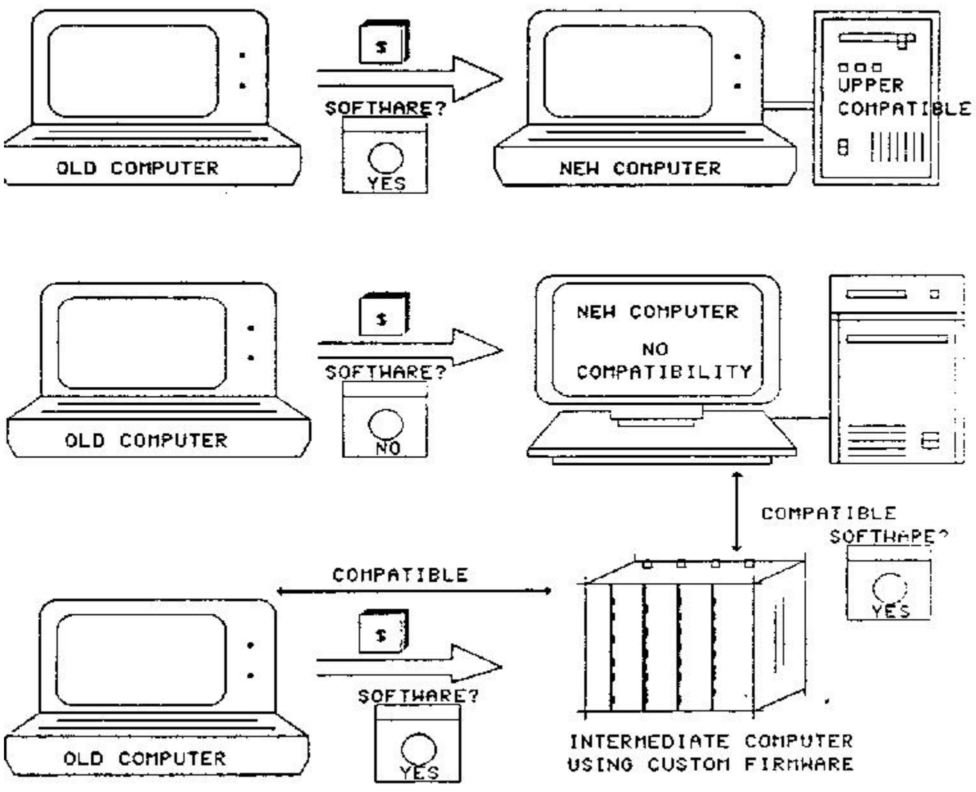


Fig. (12): Replacement of computer

6.1 Microprocessor Emulation

Programmers have used microprocessor emulation for many years as a software development vehicle. It allows programmers to write and test code on a development platform before testing on target hardware. This same concept can be practical when the target microprocessor architecture does not lend itself to efficient implementation. (8)

Microprocessors weren't always designed with in-circuit emulation in mind. But as the microprocessors evolved, the need to support in-circuit emulation within the microprocessors became obvious. Without microprocessor support, it would be very difficult, if not impossible, to halt the microprocessor anywhere on a specified breakpoint event, let alone reconstruct an instruction disassembly trace. As time went on, many more emulation features were built into the microprocessor. On the 80186 a few pins were implemented. On the 80286, a few pins and a few instructions were implemented. The 80386 expanded these support pins, added a few more instructions, some debug registers, and a few special bus cycles. The 80486 refined these same features. (9)

7. SIMULATION PROGRAM

Visual Basic software was used to design suggested simulation program for the microprocessor Intel 80386. Designed program consists of two main screens; program editor screen and Emulation and execution screen. 16-bit manipulation operations for real mode are used in design.

A brief flowchart for designed emulation program is shown in figure (13).

As shown in the flowchart, the assembly language program written in the Intel 80386 instructions is written in Edit screen. The written program is checked for errors, so if there any error it must be corrected manually. After error correcting or if there is no error, the program can be switched to Emulator screen. On emulator screen the program can be run and the results can be seen. The internal registers of the microprocessor Intel 80386 and memory locations can be seen on Emulator screen. Results then can be examined in registers and memory locations. After program running, it can be returned to Editor screen in order to write another assembly language program for the microprocessor Intel 80386.

A memory of 1 MByte is used and expected as ROM and RAM started at location 00000 H, 20 bits of address bus are used to address it.

Examples of some instructions and programs were written on edit screen and executed in Emulator screen successfully.

MOV instructions are tested successfully by writing instructions and executing them. Samples of different MOV instructions are illustrated as follows.

Direct addressing and immediate addressing MOV instructions were written in program editor screen as shown in figure (14).

Emulation and execution for them are shown in figure (15). Contents of memory locations and CPU registers are illustrated.

Register addressing and immediate addressing MOV are also used as another example. The instructions were written in program editor screen as shown in figure (16).

Their Emulation and execution for them are shown in figure (17). Also content of memory locations and registers are illustrated

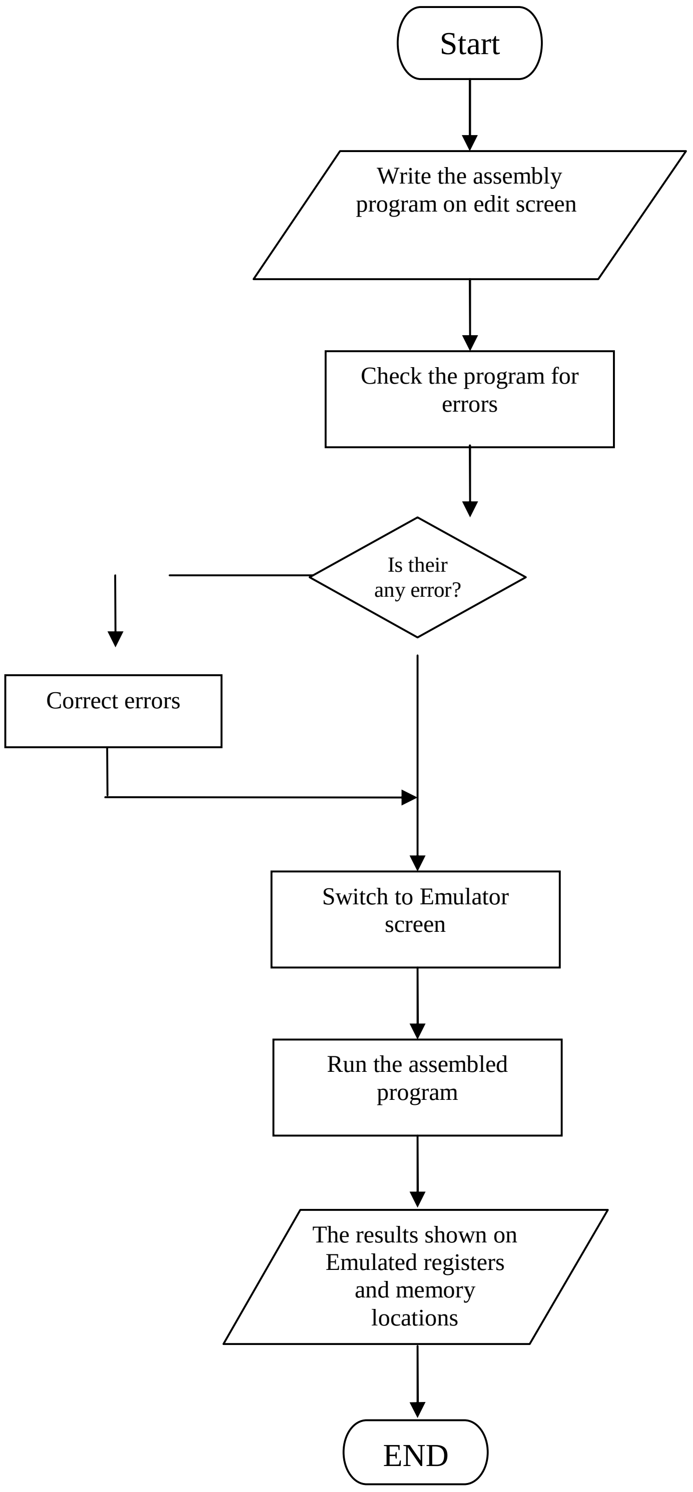
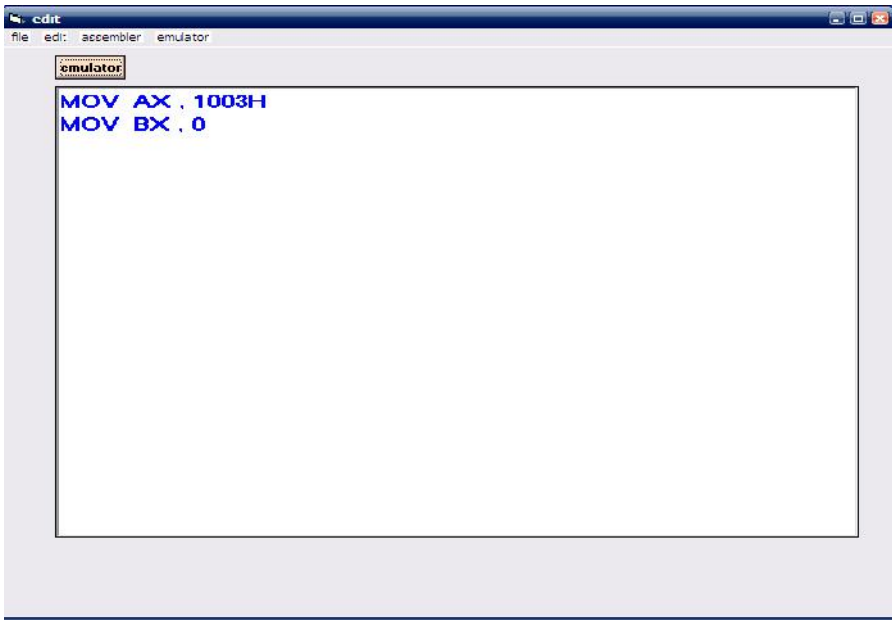
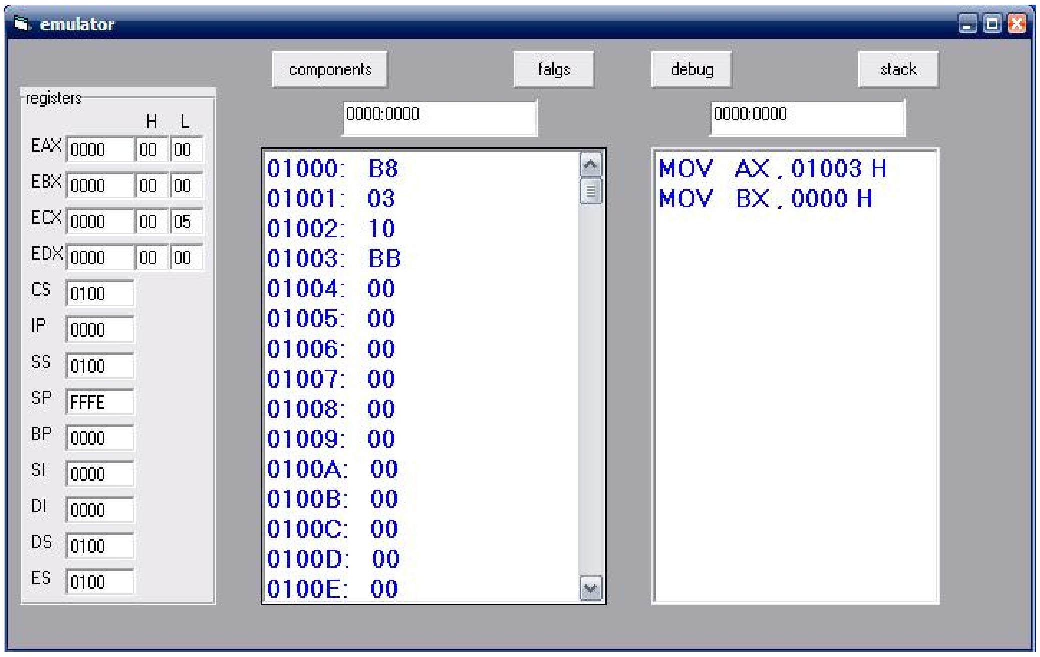

Start
Write the assembly program on edit screen
Check the program for errors
Is their any error?
Correct errors
Switch to Emulator screen
Run the assembled program
The results shown on Emulated registers and memory locations
END

Fig. (13): Flowchart of Emulation program operation.



```
MOV  AX , 1003H
MOV  BX , 0
```

Fig. (14): Direct addressing and immediate addressing MOV instructions written on Editor screen.



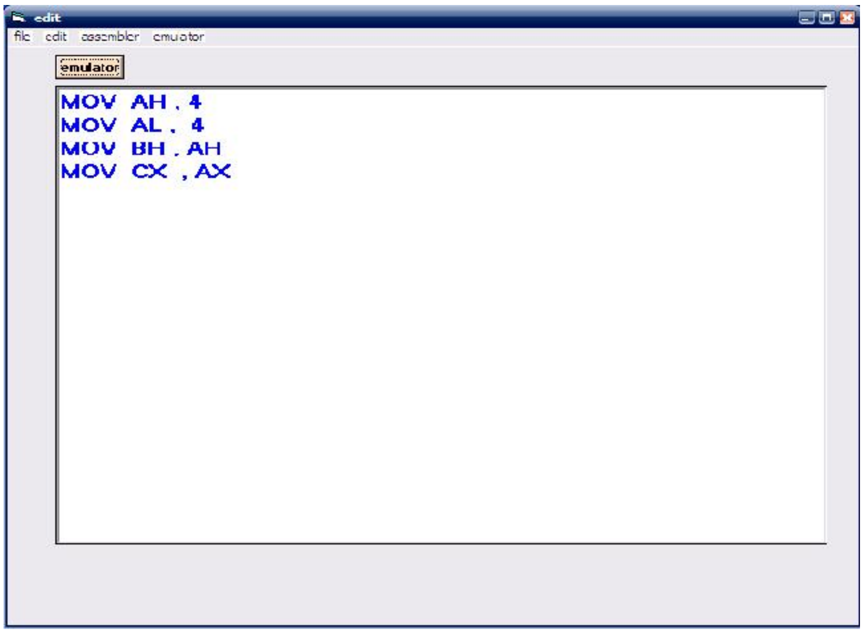
registers

		H	L
EAX	0000	00	00
EBX	0000	00	00
ECX	0000	00	05
EDX	0000	00	00
CS	0100		
IP	0000		
SS	0100		
SP	FFFE		
BP	0000		
SI	0000		
DI	0000		
DS	0100		
ES	0100		

```
01000:  B8
01001:  03
01002:  10
01003:  BB
01004:  00
01005:  00
01006:  00
01007:  00
01008:  00
01009:  00
0100A:  00
0100B:  00
0100C:  00
0100D:  00
0100E:  00
```

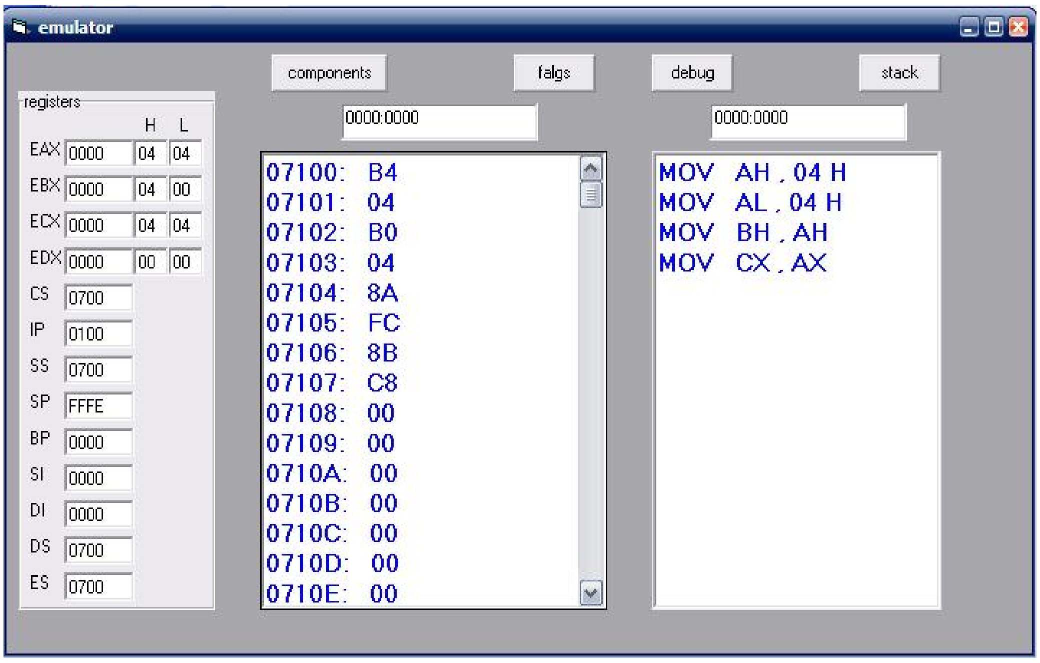
```
MOV   AX , 01003 H
MOV   BX , 0000 H
```

Fig. (15): Direct addressing and immediate addressing MOV instructions execution on Emulation and execution screen.



```
MOV AH , 4
MOV AL , 4
MOV BH , AH
MOV CX , AX
```

Fig. (16): Register addressing and immediate addressing MOV instructions written on Editor screen.



		H	L
EAX	0000	04	04
EBX	0000	04	00
ECX	0000	04	04
EDX	0000	00	00
CS	0700		
IP	0100		
SS	0700		
SP	FFFE		
BP	0000		
SI	0000		
DI	0000		
DS	0700		
ES	0700		

```
0000:0000
07100:  B4
07101:  04
07102:  B0
07103:  04
07104:  8A
07105:  FC
07106:  8B
07107:  C8
07108:  00
07109:  00
0710A:  00
0710B:  00
0710C:  00
0710D:  00
0710E:  00
```

```
0000:0000
MOV  AH , 04 H
MOV  AL , 04 H
MOV  BH , AH
MOV  CX , AX
```

Fig. (17): Register addressing and immediate addressing MOV instructions execution on Emulation and execution screen.

8. CONCLUSION

1. The designed emulation program is not the first but it is extended to recent emulation programs for Intel processors.
2. The designed program can be used for executing 32-bit operations and can be developed to do this easily, while the Intel 8086 Emulator can execute 16-bit operations.
3. The designed program can be used in university laboratories to learn students because it is designed to be easy for using by any on.

9. SUGGESTIONS

1. It is suggested to use emulation programs for advanced microprocessors learning in university laboratories.
2. Designing a complete emulator for the microprocessor 80386 to use it in universities laboratories.
3. Designing emulation programs for advanced processors like 80x86 Intel processors series other than 80386.

REFERENCES

1. Jacek Majwiski, 1992, "Functional Simulation in Microprocessors Applications Teaching", Cybernetics Institute, University of Wroclaw, Wroclaw, Poland, p (1-3).
2. Nancy A. Day, 1999, "Symbolic Simulation of Microprocessor Models Using Type Classes in Haskell", Conference on Correct Hardware Design and Verification Methods, Penn state and NEC, p (1-2).
3. Zaatar, W. Nasr, G.E., 2002, "An Implementation Scheme for a Microprocessor Emulator", Lebanese American Univ., Byblos, IEEE, (abstract).
4. Werstein, P. Cooper, C., 2002, "The use of Java to develop a microprocessor emulator", Otago Univ., Dunedin, IEEE, (abstract).
5. Barry B. Bray, 2000, "The Intel Microprocessors, Architecture, Programming, and Interfacing", 5th Edition, Prentice Hall, New Jersey, USA, p (673-689).
6. Walter A. Triebel, 1998, "The 80386, 80486 and Pentium Processors, Hardware, Software and Interfacing", Prentice Hall, New Jersey, USA, p (62-74).
7. Nachyuck Chang, 1989, "Development of a Microprogrammed Real-Time Emulator for the WestingHouse 2500 Computer, Part I Hardware", Seol National University.

8. Roman-Jones, 2003, "Emulate 8051 Microprocessor in PicoBlaze IP Core", Roman-Jones, Inc.
9. Robert R. Collins, 1999, "In-Circuit Emulation: How the Microprocessor Evolved Over Time", Intel Secrets Web Site and Robert Collins.

محاكاة للمعالج الدقيق إنتل ٨٠٣٨٦

سميرة عامر عبد القادر

مدرس مساعد

كلية التربية – جامعة ديالى

الخلاصة

تعد محاكاة المعالج الدقيق من التطبيقات الحديثة في تصميم الحاسوب. حيث تستخدم المحاكاة في تحويل أو مضاهاة المعالج الدقيق إلى برنامج لأستخدامه في أغراض معينة مثل: تدريس تركيب المعالج الدقيق و لغة التجميع في مختبرات الجامعات. هذا البحث يقترح تصميم و تنفيذ برنامج محاكاة للمعالج الدقيق انتل ٨٠٣٨٦. التنفيذ تم لبعض ايعازات نقل البيانات مثل ايعازات MOV و بمختلف أنماط العنونة. البرنامج صمم باستخدام الفيجوال بيسك. كما تم اختبار بعض ايعازات MOV بنجاح.