KICKING OFF A COURSE IN COMPUTER ORGANIZATION AND ASSEMBLY/MACHINE LANGUAGE PROGRAMMING

Richard R. Eckert
Department of Computer Science
State University of New York
Binghamton, NY 13901

INTRODUCTION

For several years I have been teaching a sophomore/junior level course entitled Computer Organization and Programming. The objectives of the course are to introduce machine organization and teach students to program in assembly language. Although there are many fine text books on assembly language programming and several good books on computer organization, I have not been able to find one that combines the two subjects so that they are meaningful to beginning students. It has been particularly difficult to start off the course in an interesting and challenging way without losing these students, who, for the most part, have been exposed only to introductory high level language programming. Recently I have tried an approach that shows signs of being successful. The heart of the approach is the use of a simple fictitious computer to illustrate the basic concepts. Although this approach is not completely new, I feel that the simplicity of the computer used along with the fact that the material is presented so early in the semester make it rather unique. In this paper I describe the machine and how it is used in the first class of the semester.

BACKGROUND MATERIAL

The course begins with a review of the component parts of a Von Neumann stored-program computer: the CPU, the memory, and the input-Output units. Figure 1 is presented to the students as the function of each part is briefly described. The three-bus architecture and the fetch-decode-execute cycle are then discussed with Figure 2 in front of the class. At this time the basic control signals: MEMR (Memory Read), MEMW (Memory Write), IOR (Input-Output Read) and IOW (Input-Output Write) are introduced and a description given of how the three buses work together with control signals from the CPU to effect the transfer of information between the CPU and memory or I/O units. After this background material is given, we turn our attention to a simple, fictitious computer.

THE SIMPLE (FICTITIOUS) COMPUTER

Figure 3 is a diagram of the computer we use to illustrate the internal organization, functioning, and programming of a real digital computer. The machine is about as simple as could be imagined. Its control section contains a Program Counter (PC) register (sometimes called an instruction pointer) which always holds the memory address from which the next instruction is to be fetched. An Instruction Register (IR) receives each instruction as it comes in from memory on the data bus. Decoder and controller/sequencer circuits output the correct series of control signals necessary to execute the instruction contained in the IR.

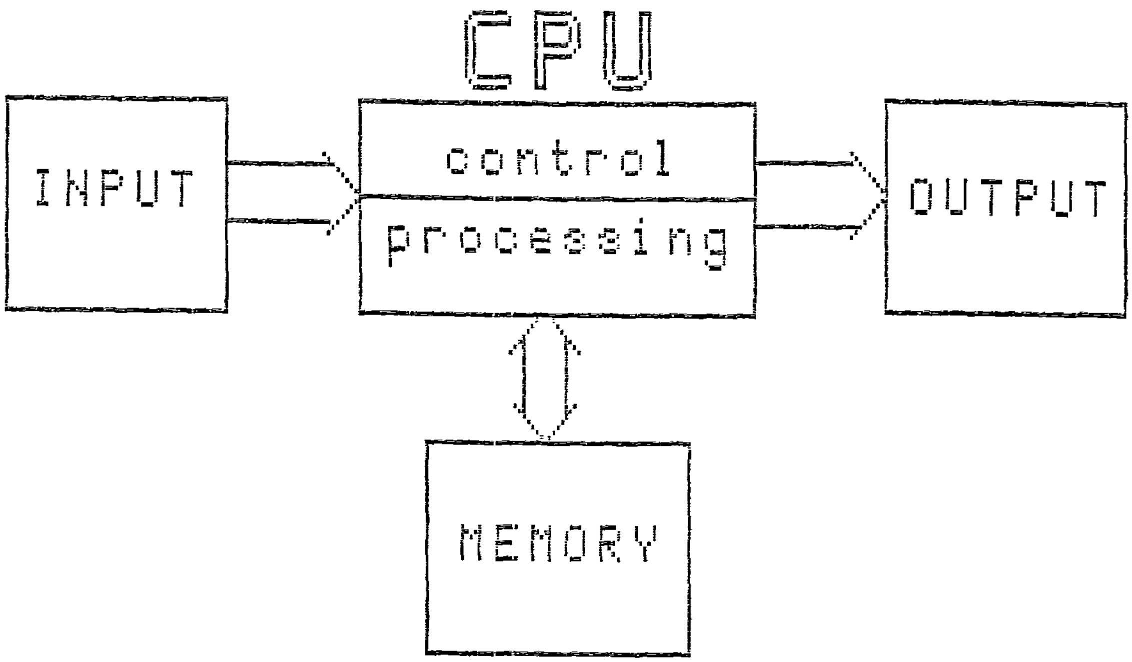

Figure 1. A Von Neumann Computer

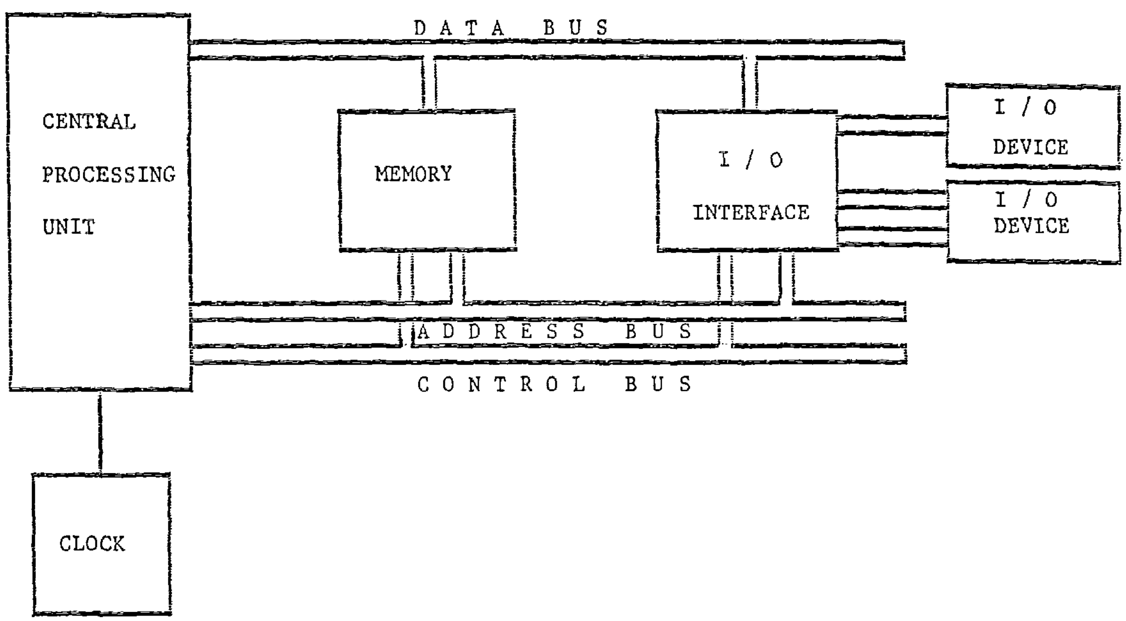

Figure 2. A 3-Bus Computer System

The processing section of the CPU contains an arithmetic-logic unit (ALU), which can perform operations on data contained in its two input registers: the accumulator (ACC) and register B. This very simple ALU is capable only of adding or subtracting. The ADD and SUBTRACT signals are provided by the control section of the CPU. For each of these two functions, the ACC will contain the result of the operation after execution.

INSTRUCTION FORMATS (FICTITIOUS) FOR THE SIMPLE PROCESSOR

Instructions for this machine consist of three-digit-long decimal numbers. The first digit (opcode field) specifies a code (0-9) for some basic instruction in the set of instructions that the machine can perform. The second two digits (operand field) specify an address (memory or I/O). For each type of instruction, the data contained in the memory cell or I/O port whose address is given in the instruction's operand field will be manipulated by the CPU. The op-code field specifies WHAT is to be done; the operand field specifies WHERE the data to be operated on is located.

```
 OPCODE     OPERAND
------------------------
/      /      /      /    <----- A 3-digit instruction
------------------------
```

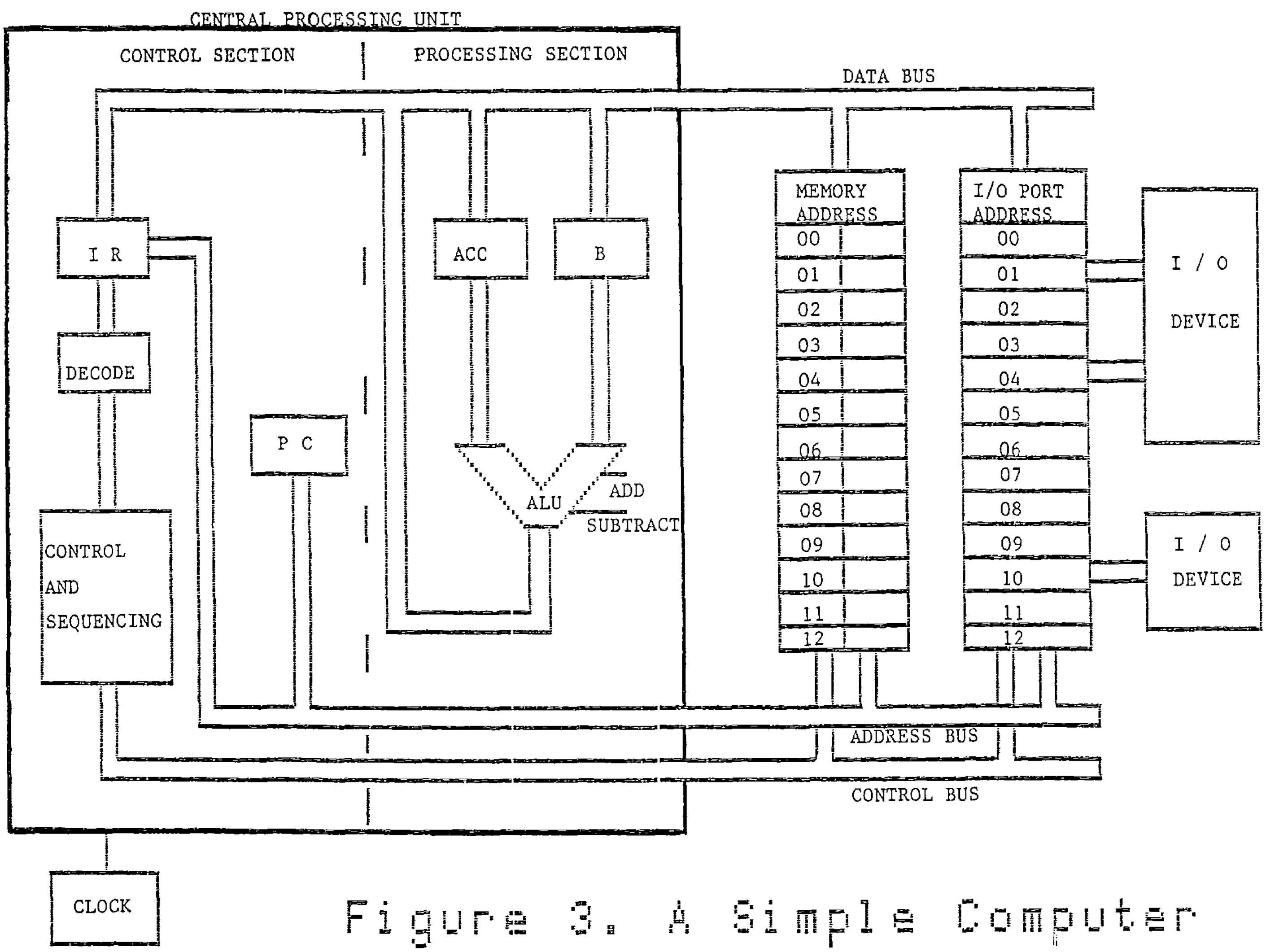

Figure 3. A Simple Computer

A SAMPLE (FICTITIOUS) INSTRUCTION SET

In order to introduce the basic ideas involved in machine language and assembly language programming, a set of instructions that may be executed on the simple computer is now given. Table I shows the instruction set. For each, the op-code number, a memory-aiding "mnemonic", and a description of what the instruction does are specified in the table.

TABLE I

AN INSTRUCTION SET FOR THE SIMPLE COMPUTER

OPCODE	MNEMONIC	MEANING
0	NOP	No operation occurs
1	LDA	The data word contained in the memory address specified in the operand field of the instruction is copied into the ACC register
2	STA	Data in the ACC is copied to the memory cell whose address is contained in the instruction's operand field
3	ADD	The word contained in the memory cell whose address is specified in the operand field is added to the contents of the ACC; the result replaces the former contents of the ACC
4	SUB	The word contained in the memory address specified in the operand field is subtracted from the ACC; the result replaces the former contents of the ACC
5	IN	Data contained in the input port whose address is given in the operand field is copied to the ACC
6	OUT	Data in the ACC is copied to the output port whose address is contained in the operand field
7	JMP	Fetch the next instruction from the memory address contained in the operand field of the current instruction (unconditional jump)
8	JN	The next instruction is to be fetched from the address in the operand field if the ACC contains a negative number (conditional jump)
9	HLT	Clock stops

WHAT HAPPENS DURING THE FETCH-DECODE-EXECUTE CYCLE

Table II shows what occurs inside our simple computer during the fetch cycle and several instruction execution cycles. From this table students observe what data is carried on the various system buses and what is contained in the CPU registers as each clock pulse is issued.

A SAMPLE PROGRAM

To see how our computer is programmed, we next give an example of an assembly language program that will add the number stored at memory location 34 to that stored at location 35 and output the result to the device connected to port 7.

```
Mnemonic   Operand    Comment

LDA         34        ;GET FIRST NUMBER
ADD         35        ;ADD SECOND NUMBER
OUT         07        ;OUTPUT RESULT TO PORT 7
HLT                   ;ALL DONE--STOP CLOCK
```

Assembling the program would then produce something like this:

```
ASSEMBLER OUTPUT            SOURCE PROGRAM
Address  Code         Mnemonic  Operand  Comment

00       134             LDA       34     ;GET FIRST NUMBER
01       335             ADD       35     ;ADD SECOND NUMBER
02       607             OUT       07     ;OUTPUT RESULT TO PORT 7
03       900             HLT              ;ALL DONE--STOP CLOCK
```

To pull everything together we now trace execution of this program, assuming that it has already been loaded into memory starting at address 0 and that the PC contains a 00. (Some of the complexities involved in starting up a real system and loading the first program into memory are mentioned, but dismissed rapidly!) We assume that memory location 34 contains a 27 and location 35 a 41. Starting the clock commences the fetch-decode-execute cycle. Table III indicates what will be on the various buses and in the various CPU registers as each instruction is fetched and executed. Each line of the table represents the state of the machine after a new clock pulse arrives. Considerable time is spent in carefully describing this table to the class.

A MORE COMPLICATED EXAMPLE

Finally, to illustrate non-sequential execution, we develop a more complicated program that will subtract two numbers that are available at input ports 3 and 4 and output the difference to output port 5 if it is non-negative; a zero is to be output to the same port if the difference is negative. Listing 1 shows an assembly language program that will perform the required task. For this program Listing 2 shows how the output of the assembler would look. (The comment field has not been reproduced in the listing.) The class then goes through the interesting exercise of tracing execution of the program on our simple machine. They again assume that it already has been loaded into memory starting at location 00 and that the two input devices contain valid data. A table similar to Table III is constructed for each of the two possible cases.

Additional more complicated programs may be assigned to students. Although our machine's instruction set is quite primitive, it is flexible enough to

TABLE II

DETAILS OF THE FETCH AND EXECUTE CYCLES OF VARIOUS INSTRUCTIONS

```
*** FETCH CYCLE ***
-1st clock pulse: The contents of the PC are placed onto the
                  address bus.
-2nd clock pulse: The MEMR line of the control bus is activated.
                  This causes memory to respond by placing the
                  data contained in the location whose address is
                  on the address bus onto the data bus.
-3rd clock pulse: The IR receives the data that is on the data
                  bus. At the same time the PC is incremented.
                  The instruction is now safely in the IR, where
                  it can be decoded; the PC points to the next
                  instruction in memory. The execute portion
                  of the cycle will now occur. What happens next
                  depends upon what the instruction is.

*** OPCODE 0 -- NOP ***
-1st clk: Nothing occurs; the fetch cycle resumes on the next
          clock pulse.
*** OPCODE 1 -- LDA ***
-1st clk: The least significant 2 digits (operand field) of the
          IR are sent out on the address bus.
-2nd clk: MEMR becomes active on the control bus. This causes
          the contents of the selected memory cell to be placed
          on the data bus.
-3rd clk: The contents of the data bus are received by the ACC.
*** OPCODE 3 -- ADD ***
-1st clk: The least significant 2 digits of the IR go out on
          address bus.
-2nd clk: MEMR becomes active on the control bus causing the
          contents of the selected memory location to be deposited
          on the data bus.
-3rd clk: The contents of the data bus are received by register B
          inside the cpu.
-4th clk: The ALU's ADD signal is activated causing the contents
          of register B to be added to the ACC. The result of the
          addition is stored back into the ACC.
*** OPCODE 6 -- OUT ***
-1st clk: The least significant 2 digits of the IR go out on the
          address bus.
-2nd clk: The contents of the ACC go out on the data bus.
-3rd clk: IOW becomes active on the control bus causing the
          contents of the data bus to be received by the selected
          output port.
*** OPCODE 8 -- JN ***
-1st clk: If the contents of the ACC are negative, copy the least
          significant 2 digits of the IR to the PC; if not, do
          nothing. The result is that, if the ACC was negative,
          the PC will now be pointing to the jump address. The
          next fetch will then bring in the instruction stored at
          that address. If, on the otherhand, the ACC was
          non-negative, the next fetch will bring in the next
          instruction in sequence, since, in that case, the PC was
          not altered.
*** OPCODE 9 -- HLT ***
-1st clk: Stop the clock. The fetch-decide-exeute cycle terminates.
```

permit programs containing combinations of any of the three basic program control structures: sequence, iteration, and selection.

STUDENT REACTION

After starting off the semester with this approach the past two times I have taught the course, unsolicited comments from students have been very positive. Several of them even went to the extreme of saying that for the first time they had gained some idea of "what really goes on inside a computer." Use of the simple fictitious machine described in this article also seems to lead very naturally into real machines and instruction sets. We have been emphasizing the Intel 8088-based, IBM-PC microcomputer and the IBM System/370 in the course. Although these machines are considerably more complex than the fictitious computer, students, after having been exposed to the latter, seem to "catch on" rapidly to how the the real machines work and are programmed.

```
Label   Mnemonic   Operand    Comment

        IN         3          ;GET 1ST NUMBER
        STA        99         ;STORE IT IN MEMORY
        IN         4          ;GET 2ND NUMBER
        STA        98         ;STORE IN MEMORY
        LDA        99         ;GET 1ST NUMBER INTO ACC
        SUB        98         ;SUBTRACT 2ND NUMBER--RESULT IN ACC
        JN         NEG        ;IF RESULT IS NEGATIVE, GO TO 'NEG'
        JMP        DONE       ;IF NOT, GO TO END OF PROGRAM
NEG:    STA        97         ;STORE RESULT
        SUB        97         ;RESULT-RESULT=ZERO
DONE:   OUT        5          ;OUTPUT THE RESULT OR THE ZERO
        HLT                   ;ALL DONE
```

Listing 1. An assembly language program that inputs two numbers, finds their difference, and outputs it if it is non-negative. A zero is output if the result is negative.

```
Address   Code    Label   Mnemonic   Operand

00        503             IN         3
01        299             STA        99
02        504             IN         4
03        298             STA        98
04        199             LDA        99
05        498             SUB        98
06        808             JN         NEG
07        710             JMP        DONE
08        297     NEG:    STA        97
09        497             SUB        97
10        605     DONE:   OUT        5
11        900             HLT
```

Listing 2. The assembled version of Listing 1. (The comment field has been omitted.)

TABLE III

THE STATE OF THE COMPUTER DURING EXECUTION OF THE PROGRAM:

00	134	LDA	34
01	335	ADD	35
02	607	OUT	07
03	900	HLT	

CPU REGISTERS				SYSTEM BUSES			WHAT'S HAPPENING
PC	IR	ACC	B	Address	Control	Data	
00							
							Clock starts
00				00			Begin 1st fetch
00					MEMR	134	
01	134						End 1st fetch
01	134			34			Begin 1st execute
01	134				MEMR	27	
01	134	27					End 1st execute
01	134	27		01			Begin 2nd fetch
01	134	27			MEMR	335	
02	335	27					End 2nd fetch
02	335	27		35			Begin 2nd execute
02	335	27			MEMR	41	
02	335	27	41				
02	335	68	41		ADD to ALU		End 2nd execute
02	335	68	41	02			Begin 3rd fetch
02	335	68	41		MEMR	607	
03	607	68	41				End 3rd fetch
03	607	68	41	07			Begin 3rd execute
03	607	68	41			68	
03	607	68	41		IOW		(Result received by output port) End 3rd execute
03	607	68	41	03			Begin 4th fetch
03	607	68	41		MEMR	900	
04	900	68	41				End 4th fetch
04	900	68	41				Begin 4th execute Clock stops