Let Us Build a Computer (A)

We now have learned how to build simple circuits that compute functions and store information. The next task is to study the assembly of such circuits into a computer.

Most computer architectures that have been built up to the present day have two major subparts, the machine central processing unit (CPU) and the memory. The CPU executes the calculations and thus has computational registers wired to do the arithmetic operations and other manipulations on data. It contains the Instruction Register introduced in chapter 6, and its associated decoding circuitry. It contains all the mechanisms for retrieving instructions to be executed, sequencing them through the Instruction Register, and getting the individual instructions executed.

The memory has no ability to compute anything. It is simply a huge array of very low-cost registers where information can be stored when needed. The registers are numbered, usually from 0 to \( n \), where each location may hold perhaps 8, 16, or some other number of bits, and the values of \( n \) may be large, say, from 32,767 to many millions.

The central processor needs two kinds of information that are stored in the memory: its own instructions and the data that it works on. Its instructions are stored in the memory as a sequence of binary coded machine instructions, and they are sequentially brought into the central processor for execution. The resulting computations utilize the other kind of information stored in the memory, the data. These data contain the character sequences and the numbers that the user wants to manipulate to obtain the desired answers.

A typical instruction for the machine in this chapter is \( COPY AX,X \) as will be described in a later section. When such an instruction is stored in the memory or in a register, it is usually coded as a binary number such as 001011000010100. However, we will not use these binary codes in the discussions but will write \( *COPY AX,X * \) where the surrounding asterisks indicate that the
instruction has been translated into binary code. That is, COPY AX, X is an instruction to the machine, and *COPY AX, X* represents its binary code:

```
10  *COPY  AX, X  *
11  *ADD  AX, Y  *
12  *COPY  CN1, AX*
13  *COPY  AX, CN1*
14  *COPY  Z, AX  *
20  7 (X)
21  4 (Y)
22  0 (Z)
23  0 (CN1)
```

The computation involves continuously bringing into the CPU data items, which are then combined and modified according to the instructions and returned to the memory.

A huge variety of digital machines exists in today’s world, and it is not important to enumerate them here. There are machines with only one computational register in the CPU and a few instructions. There are also machines with dozens of such registers and hundreds of instructions. An example architecture is the CPU for the IBM Personal Computer, which uses the Intel 8088 VLSI chip as its central processor. It has 12 registers and 92 instructions, which do a variety of arithmetic, logical, and character operations. This processor performs primitive internal operations at the rate of one every 210 nanoseconds (.000000210 second). Many basic instructions require two such cycles, and a few complex instructions require many such cycles. For example, 16-bit division requires 206 cycles.

Another way of varying the architecture is to have several copies of the CPU-memory configuration shown above. This is called a parallel architecture, and it enables the programmer to divide a computation into several different parts, which can all be worked on simultaneously. Parallel architectures and their uses are described in chapter 12.

Regardless of the details, most digital computers use the same fundamental mechanisms, and our understanding will be sufficient if we can comprehend the workings of only a very simple machine. The machine described here will have only one register for manipulating data and only 12 instructions. It is called the P88 Machine, which stands for “part of an Intel 8088.” Its instructions are nearly identical to some of the instructions of the Intel 8088 for the sake of achieving some degree of realism. A programmer for a real Intel 8088 needs to learn to use 12 computational registers instead of one and 92 instructions instead of 12.
An Example Architecture: the P88 Machine (B)

The P88 computer contains the components shown below: an Instruction Pointer Register IP, an Instruction Register IR, a Condition Flag CF, the Computational Register AX, and the memory.

The functioning of the computer proceeds by repeatedly executing the following two steps:

Repeat without end:
1. (Fetch) Find the instruction in memory at the address given by IP and put that instruction into IR. Increment IP to give the address of the next instruction.
2. (Execute) Execute the instruction in IR.

These two steps comprise what is called the fetch-execute cycle, and they can best be understood by going over an illustration. Suppose that the machine instruction codes

\[
\begin{align*}
\text{*COPY AX, X} & \\
\text{*ADD AX, Y} & \\
\text{*COPY CN1, AX} & \\
\text{*COPY AX, CN1} & \\
\text{*COPY Z, AX} &
\end{align*}
\]

reside in the memory at locations 10 through 14 and that the machine is about to execute these codes. Then the address (10) of the first such code \text{*COPY AX, X} will be in IP:
The first step of the fetch-execute cycle involves loading IR from the location given by IP. Then IP is incremented to give the address of the next instruction:

![Diagram of fetch and execute steps]

The second step is the execute cycle, which carries out the instruction in the Instruction Register IR. In this case, `*COPY AX,X` is executed. As is explained below, the contents of memory location X will be copied into the register AX. The circuitry doing this task is that explained in Chapter 6, the instruction decoding and execution (D&E) circuits:

![Diagram of D&E execution steps]

The fetch-execute cycle continues without stopping. On the next cycle, the instruction is fetched from location 11 in the memory. This is an add instruction, `*ADD AX,Y`. It adds the contents of memory location Y to the contents of register AX and leaves the result in register AX. Here is the result of this fetch and execute:
Another pass through the fetch-execute results in the third instruction's being executed. This instruction causes the contents of the AX register to be copied into the memory location CNI:

Thus the machine sequentially loads the string of commands in memory and carries them out. Occasionally a "jump" instruction will load a new value into IP, which begins a new sequence of commands. For example, in the above case, it is possible that some instruction would cause 10 to be loaded into IP again and result in another execution of this sequence.

The fundamental operation of every digital computer is built around the fetch-execute cycle as described here. This is, in fact, the only thing that modern computers are designed to do—to fetch instructions and execute them. There is no need to be impressed with the intelligence of any modern computer, no matter how large, with its blinking lights, complex displays, and many workers huddled over their terminals. The giant machine is doing nothing more than fetching instructions and executing them. It never has in the past, cannot now, and never in the future will be able to do more or less than this!
Exercises
1. Trace the execution of the final two instructions in the above example—those instructions in memory locations 13 and 14. Show every detail of the fetch-execute cycle.
2. In the example above, the instruction pointer IP increases one on each cycle and will apparently soon reach 20. An error will occur if it does. Explain the nature of the error. What instruction must be included after location 14 but before location 20 to avoid this error?

Programming the P88 Machine (B)

The P88 has 12 instructions, as shown below. Each individual instruction is simple in its operation, as described by its action entry. Thus, the first instruction is called a “copy” and is written \textit{COPY AX,.mem} where \textit{mem} refers to a location in memory. If this instruction is executed, the contents of the memory at location \textit{mem} is copied into register AX. Another copy instruction \textit{COPY mem,AX} copies the contents of AX to memory location \textit{mem}. As an illustration, one can use these two instructions to move data from memory location \text{A} to memory location \text{B}.

\begin{verbatim}
COPY AX,A
COPY B,AX
\end{verbatim}

Here are the instructions.

\begin{center}
\begin{tabular}{lll}
Instruction & Format & Action \\
copy from mem & \texttt{COPY AX,.mem} & AX := mem \\
copy to mem & \texttt{COPY mem,AX} & mem := AX \\
add & \texttt{ADD AX,.mem} & AX := AX+mem \\
subtract & \texttt{SUB AX,.mem} & AX := AX-mem \\
multiply & \texttt{MUL AX,.mem} & AX := AX*mem \\
divide & \texttt{DIV AX,.mem} & AX := AX/\text{mem} \\
compare & \texttt{CMP AX,.mem} & If AX < mem then \\
 & & CF := 1 \\
 & & else \\
 & & CF := NB \\
jump & \texttt{JMP labl} & Go to the instruction \\
 & & with label labl. \\
jump if not below & \texttt{JNB labl} & Go to the instruction \\
 & & with label labl if \\
 & & CF=NB. Otherwise go to next instruction. \\
jump if below & \texttt{JB labl} & Go to the instruction \\
 & & with label labl if \\
 & & CF=B. Otherwise go to the next instruction.
\end{tabular}
\end{center}
Machine Architecture

input \hspace{1cm} \text{IN AX} \hspace{1cm} \text{Input an integer into register AX.}

output \hspace{1cm} \text{OUT AX} \hspace{1cm} \text{Output an integer from register AX.}

The P88 has four arithmetic instructions—ADD, SUB, MUL, and DIV—which perform their respective operations on AX and a memory location and leave the result in AX. Thus ADD AX,B adds the contents of memory location B to AX and leaves the result in AX.

It also has instructions for reading and printing integers, IN and OUT. These instructions can be used for writing some P88 programs. Here is a P88 program for reading an integer into register AX and then printing it:

\text{IN AX}
\text{OUT AX}

Or one could read an integer, square it, and print the result:

\text{IN AX}
\text{COPY M1,AX}
\text{MUL AX,M1}
\text{OUT AX}

As another example, one could read two numbers, divide the first by the second, and print the result:

\text{IN AX}
\text{COPY A,AX}
\text{IN AX}
\text{COPY B,AX}
\text{DIV AX,B}
\text{OUT AX}

Another type of instruction is the “jump,” which loads a new address into IP and causes the machine to jump to an instruction that is not next in sequence. As an example, consider the following program, which adds the number in memory location A to AX repeatedly:

\text{L1 ADD AX,A}
\text{JMP L1}

This program adds A to AX, and then the next instruction causes a jump to the instruction labeled L1. Here A is again added to AX. Then the jump instruction sends the machine back to L1 again and so forth. This program loops forever, adding A to AX an unlimited number of times.
Of course, it is usually preferred to write a loop that will halt after an appropriate number of repetitions. This is done with the combination of the “compare” and conditional jump instructions, 
\( JB \) and \( JNB \). An example of this type of program is the following code, which prints the numbers from 1 to 10 before exiting. This program assumes the numbers 0, 1, and 10 appear in memory locations \( M0, M1, \) and \( M10 \).

\[
\begin{align*}
\text{COPY} & \quad AX, M0 \\
\text{L1} & \quad \text{ADD} \quad AX, M1 \\
\text{OUT} & \quad AX \\
\text{CMP} & \quad AX, M10 \\
\text{JB} & \quad \text{L1}
\end{align*}
\]

The compare instruction loads \( B \) into register \( CF \) if \( AX \) is less than (below) \( M10 \). The “jump if below” instruction jumps to \( L1 \) if \( CF \) is \( B \), that is, if \( AX \) is below \( M10 \). A paraphrase of this program shows its method of operation.

- Put 0 into \( AX \).
- Add 1 to \( AX \).
- Print the contents of \( AX \).
- Check whether \( AX \) is less than 10.
- If it is, go to \( L1 \).

Another example of a program that uses the compare and jump commands is the following, which reads a number and changes its sign if it is negative. This program thus computes what is called the “absolute value” in mathematics:

\[
\begin{align*}
\text{IN} & \quad AX \\
\text{COPY} & \quad M1, AX \\
\text{SUB} & \quad AX, M1 \\
\text{CMP} & \quad AX, M1 \\
\text{JB} & \quad \text{NEXT} \\
\text{SUB} & \quad AX, M1 \\
\text{COPY} & \quad M1, AX \\
\text{NEXT} & \quad \text{COPY} \quad AX, M1 \\
\text{OUT} & \quad AX
\end{align*}
\]

The final example will be a program to add a series of nonnegative numbers. If a negative number is read at any time, the program prints the sum of all nonnegative numbers read and exits:
The language of machine instructions written in symbolic form as described here—COPY, ADD, CMP, etc.—is called assembly language. This type of language was heavily used during the 1940s and 1950s before higher-level programming languages like FORTRAN, PL/I, and Pascal were available. Each instruction in assembly language can be directly translated into a binary code, the machine language, denoted in this chapter by instructions surrounded by asterisks. The machine language instructions can be loaded into the IR register and executed as explained here and in chapter 6.

**Exercises**

1. Explain what function the following program computes:

```
IN AX
COPY M1,AX
SUB AX,M1
COPY ZERO,AX
COPY SUM,AX
COPY AX,M1
LOOP CMP AX,ZERO
   JB FIN
   ADD AX,SUM
   COPY SUM,AX
   IN AX
   JMP LOOP
FIN COPY AX,SUM
OUT AX
```
2. Write an assembly language program that reads two integers and prints the larger one.
3. Write an assembly language program that reads two integers—a small integer followed by a larger one. Then it prints all of the integers between but not including them.
4. A programmer noticed that a machine was running more slowly each day and wondered why. Furthermore, it acted erratically from time to time. She studied the code running in the machine, and after considerable effort, found the following code that she could identify as being of unknown origin:

```
V1    JMP    BEGIN
ZERO  0
ONE   1
TEN   0
FIFTY 50
LENGTH 33
COUNT  0
N1    0
BEGIN  COPY AX, RANDOM
       COPY N1, AX
       DIV AX, TEN
       MUL AX, TEN
       SUB AX, N1
       CMP AX, ZERO
       JB EXIT
       COPY AX, ZERO
       COPY COUNT, AX
       LOOP1 COPY CX, RANDOM
               COPY AX, COUNT
               CMP AX, FIFTY
               JNB EXIT
               COPY BX, ZERO
       LOOP2 CMP BX, LENGTH
               JNB NEXT
               COPY AX, V1+c (BX)
               COPY c (CX), AX
               ADD BX, ONE
               ADD CX, ONE
               JMP LOOP2
```
In fact, the programmer had found a computer virus, and your job is to analyze it to discover how it works. This is a program that might be inserted into a computer system by an unfriendly person. It sits quietly in the middle of any program that it is inserted into until the right moment. Then it springs into action and duplicates itself many times around the machine memory.

Can you answer the following questions about the virus: How does it make the decision to go into action? How does it hide when it is not doing anything? What mechanism does it use to duplicate itself? How many times does it duplicate itself? Why does the virus cause the machine to seem to slow down? Why does it cause the system to act erratically sometimes? What should the programmer do to get rid of this virus? What problem does this virus have in carrying out its destructive job, and how can it be improved?

The virus uses some features of the P88 machine not discussed before. It references a location called \textit{RANDOM}, which contains a random number. Every time that location is referenced, it gives a different random number. Two additional registers are assumed: they are called \textit{BX} and \textit{CX}. The notation \( c(CX) \) refers to the memory location with the address equal to the number in \( CX \). Thus, if register \( CX \) contains 100, then \( COPY c(CX), AX \) will put the contents of \( AX \) into location 100 in memory. The expression \( V1 + c(BX) \) refers to the memory location found by starting at location \( V1 \) at the beginning of the program and counting \( BX \) locations beyond it.

\textbf{Summary (B)}

This chapter describes the classic architecture of the great majority of digital computers. They have a CPU that runs the fetch-execute cycle on sequences of instructions, and these instructions are stored in memory with the data of the computation. This is called the \textit{von Neumann} architecture, and the only common deviations from it are the parallel machines described in chapter 12.

This chapter completes another link in the chain of concepts required to understand how computers work. At the lowest level are the transistors, the valves that control electric current flow. Above them are the electric circuits, which we can design to compute functions and store information. Such circuits can be organized to do useful tasks such as add numbers or manipulate characters. Binary coded machine language instructions are used to activate the appropriate computational circuits, and such instructions are sequenced through the Instruction Register \( IR \) in order to do a nontrivial calculation. The binary coded machine instructions are actually translations of symbolic assembly language instructions, which are much more convenient for hu-
mans to read. This assembly language, its binary coded form, and its execution on a machine have been the topic of this chapter.

The final and highest level set of concepts in the chain relates to the translation of a high-level language such as Pascal into assembly language. This study will complete the chain and will be the topic for the next chapter.

Looking at this chain in the other direction, you can trace the processing of a Pascal statement from its entry into the computer to the detailed switching of electrical currents inside of the machine. Consider, for example, the statement

\[ Z := (X + Y) \]

and all of the resulting processing. It is translated into assembly language and then into machine language. These instructions are sequenced through the Instruction Register which gets the job done. \( X \) is brought in from memory, \( Y \) is added to it, and the result is placed in \( Z \):
Readings

On computer architecture and assembly language programming: