# Microprocessors

Early computer science pioneers such as Alan Turing and John Von Neumann postulated that for a computing device to be really useful, it not only had to be able to generate specific outputs as dictated by programmed instructions, but it also had to be able to write data to memory, and be able to act on that data later. Both the program steps and the processed data were to reside in a common memory "pool," thus giving way to the label of the *stored-program computer*. Turing's theoretical machine utilized a sequential-access tape, which would store data for a control circuit to read, the control circuit re-writing data to the tape and/or moving the tape to a new position to read more data. Modern computers use random-access memory devices instead of sequential-access tapes to accomplish essentially the same thing, except with greater capability.

A helpful illustration is that of early automatic machine tool control technology. Called *open-loop*, or sometimes just *NC* (numerical control), these control systems would direct the motion of a machine tool such as a lathe or a mill by following instructions programmed as holes in paper tape. The tape would be run one direction through a "read" mechanism, and the machine would blindly follow the instructions on the tape without regard to any other conditions. While these devices eliminated the burden of having to have a human machinist direct every motion of the machine tool, it was limited in usefulness. Because the machine was blind to the real world, only following the instructions written on the tape, it could not compensate for changing conditions such as expansion of the metal or wear of the mechanisms. Also, the tape programmer had to be acutely aware of the sequence of previous instructions in the machine's program to avoid troublesome circumstances (such as telling the machine tool to move the drill bit laterally while it is still inserted into a hole in the work), since the device had no memory other than the tape itself, which was read-only. Upgrading from a simple tape reader to a Finite State control design gave the device a sort of memory that could be used to keep track of what it had already done (through feedback of some of the data bits to the address bits), so at least the programmer could decide to have the circuit remember "states" that the machine tool could be in (such as "coolant on," or tool position). However, there was still room for improvement.
The ultimate approach is to have the program give instructions which would include the writing of new data to a read/write (RAM) memory, which the program could easily recall and process. This way, the control system could record what it had

done, and any sensor-detectable process changes, much in the same way that a human machinist might jot down notes or measurements on a scratch-pad for future reference in his or her work. This is what is referred to as CNC, or *Closed-loop Numerical Control*.

Engineers and computer scientists looked forward to the possibility of building digital devices that could modify their own programming, much the same as the human brain adapts the strength of inter-neural connections depending on environmental experiences (that is why memory retention improves with repeated study, and behavior is modified through consequential feedback). Only if the computer's program were stored in the same writable memory "pool" as the data would this be practical. It is interesting to note that the notion of a self-modifying program is still considered to be on the cutting edge of computer science. Most computer programming relies on rather fixed sequences of instructions, with a separate field of data being the only information that gets altered.

To facilitate the stored-program approach, we require a device that is much more complex than the simple FSM, although many of the same principles apply. First, we need read/write memory that can be easily accessed: this is easy enough to do. Static or dynamic RAM chips do the job well, and are inexpensive. Secondly, we need some form of logic to process the data stored in memory. Because standard and Boolean arithmetic functions are so useful, we can use an Arithmetic Logic Unit (ALU) such as the look-up table ROM example explored earlier. Finally, we need a device that controls how and where data flows between the memory, the ALU, and the outside world. This so-called *Control Unit* is the most mysterious piece of the puzzle yet, being comprised of tri-state buffers (to direct data to and from buses) and decoding logic which interprets certain binary codes as instructions to carry out. Sample instructions might be something like: "add the number stored at memory address 0010 with the number stored at memory address 1101," or, "determine the parity of the data in memory address 0111." The choice of which binary codes represent which instructions for the Control Unit to decode is largely arbitrary, just as the choice of which binary codes to use in representing the letters of the alphabet in the ASCII standard was largely arbitrary. ASCII, however, is now an internationally recognized standard, whereas control unit instruction codes are almost always manufacturer-specific.

Putting these components together (read/write memory, ALU, and control unit) results in a digital device that is typically called a *processor*. If minimal memory is used, and all the necessary components are contained on a single integrated circuit, it is called a *microprocessor*. When combined with the necessary bus-control support circuitry, it is known as a *Central Processing Unit*, or CPU.
CPU operation is summed up in the so-called *fetch/execute cycle*. *Fetch* means to read an instruction from memory for the Control Unit to decode. A small binary counter in the CPU (known as the *program counter* or *instruction pointer*) holds the address value where the next instruction is stored in main memory. The Control Unit sends this binary address value to the main memory's address lines, and the memory's data output is read by the Control Unit to send to another holding register. If the fetched instruction requires reading more data from memory (for example, in adding two numbers together, we have to read both the numbers that are to be added from main memory or from some other source), the Control Unit appropriately addresses the location of the requested data and directs the data output to ALU registers. Next, the Control Unit would execute the instruction by signaling the ALU to do whatever was requested with the two numbers, and direct the result to another register called the*accumulator*. The instruction has now been "fetched" and "executed," so the Control Unit now increments the program counter to step the next instruction, and the cycle repeats itself.

```
        Microprocessor (CPU)

--------------------------------------
|       ** Program counter **        |
|  (increments address value sent to |
|  external memory chip(s) to fetch  |==========> Address bus
|  the next instruction)             |          (to RAM memory)
--------------------------------------
|        ** Control Unit **          |<=========> Control Bus
|  (decodes instructions read from   | (to all devices sharing
|  program in memory, enables flow   | address and/or data busses;
|  of data to and from ALU, internal | arbitrates all bus communi-
|  registers, and external devices)  | cations)
--------------------------------------
| ** Arithmetic Logic Unit (ALU) **  |
|    (performs all mathematical      |
|     calculations and Boolean       |
|     functions)                     |
--------------------------------------
|          ** Registers **           |
|    (small read/write memories for  |<=========> Data Bus
|     holding instruction codes,     | (from RAM memory and other
|     error codes, ALU data, etc;    |  external devices)
|     includes the "accumulator")    |
--------------------------------------
```

As one might guess, carrying out even simple instructions is a tedious process. Several steps are necessary for the Control Unit to complete the simplest of mathematical procedures. This is especially true for arithmetic procedures such as exponents, which involve repeated executions ("iterations") of simpler functions. Just imagine the sheer quantity of steps necessary within the CPU to update the bits of information for the graphic display on a flight simulator game! The only thing which makes such a tedious process practical is the fact that microprocessor circuits are able to repeat the fetch/execute cycle with great speed.

In some microprocessor designs, there are minimal programs stored within a special ROM memory internal to the device (called *microcode*) which handle all the sub-steps necessary to carry out more complex math operations. This way, only a single instruction has to be read from the program RAM to do the task, and the programmer doesn't have to deal with trying to tell the microprocessor how to do every minute step. In essence, its a processor inside of a processor; a program running inside of a program.

**Source: http://www.allaboutcircuits.com/vol_4/chpt_16/4.html**