Processing

Introduction

To do anything, the computer hardware follows a computer program. A program is a step-by-step list of actions the hardware can do as well as any numbers, letters, or other information needed to do these actions: a list of instructions and data.

Instructions

The CPU is the part of the computer hardware that does processing. It has a number of built-in actions it can perform, each of which has an associated binary code, the *instruction*.

The Instruction Register of a CPU is wired to the rest of the components of the CPU and essential components on the motherboard, such as the bus, so the pattern of on's and off's in an instruction can turn on and off different components to control what the computer does.

Most instructions involve the ALU performing mathematical operations (such as adding, or comparing for equality) to numbers stored in the data registers, and storing the result in a data register. The ALU can only operate on data that is in the data registers, and puts its results back into data registers, rather than dealing directly with Main Memory, so data must be copied back and forth across the bus.

This means we need instructions to tell the ALU what arithmetic operations to carry out, and instructions that tell the CPU to copy

data between Main Memory and a Data Register[1]. We also need instructions for other basic actions, such as changing the address stored in the Program Counter or dealing with ending a program, as well as more complex tasks like communicating with the Hard Drive.

Machine Language and Assembly Language

The list of all the binary patterns for instructions the CPU can follow is its *machine language*. Each type of CPU has its own machine language.

Since remembering binary patterns is difficult for humans, we often write instructions in Assembly Language, which uses mnemonic codes for instructions[2], e.g. "ADD R1 R2 R3" might mean "Have the ALU add the data in data register 1 to the data in data register 2 and put the result in data register 3" and is easier for humans to remember than 0000000000100010 0011000000100000.

Assembly language programs are stored in the computer using the binary for the letters and numbers in them, which are *not* the same as the binary for the instructions they represent[3]; so Assembly Language programs are useless to the CPU. They must be translated to Machine Language before they can actually be run by the CPU.

---

[1] A more complex computer might have a single instruction that to copies data to a register, does arithmetic, and then copies the result back to MM. We will assume a less-complex computer that has individual instructions for these, to keep our examples short and simple.

[2] I will use Assembly in examples and problems, but you do not need to memorize what the Assembly instructions mean; I will always give you the definitions.

[3] The binary to store "ADD R1 R2 R3" would consist of the binary pattern for A then the binary pattern for D (twice) and so on. The actual instruction for this in machine language would be a single binary pattern with on's in the positions to activate the right parts of the ALU to add, etc.

Running Programs

To run a program, the computer must do all the instructions in the program. The process of doing a single instruction in a program is called an *instruction cycle*. The Program Counter's job is to keep track of the Main Memory address of the next instruction to be done. During an instruction cycle the CPU gets the next instruction into the IR, determines what it says to do, does it, and then updates the Program Counter for the next instruction.

In fact, the instruction cycle is the only thing the CPU ever does, from the time the computer is turned on until it is turned off. The CPU doesn't even know about programs. It simply does the Instruction Cycle process over and over again.

For a program to run, its instructions and data must be stored in Main Memory, and the Program Counter must be set to the Main Memory address of the first instruction. Then the CPU does the instruction cycle over and over for all the instructions in the program.

Although we often think of the CPU as the "brain" of the computer, keep in mind that it does not actually know how to do anything at all except the Instruction Cycle that is built into the hardware. If we give it an instruction to do, it does it, whether it makes any sense or not.

The Instruction Cycle

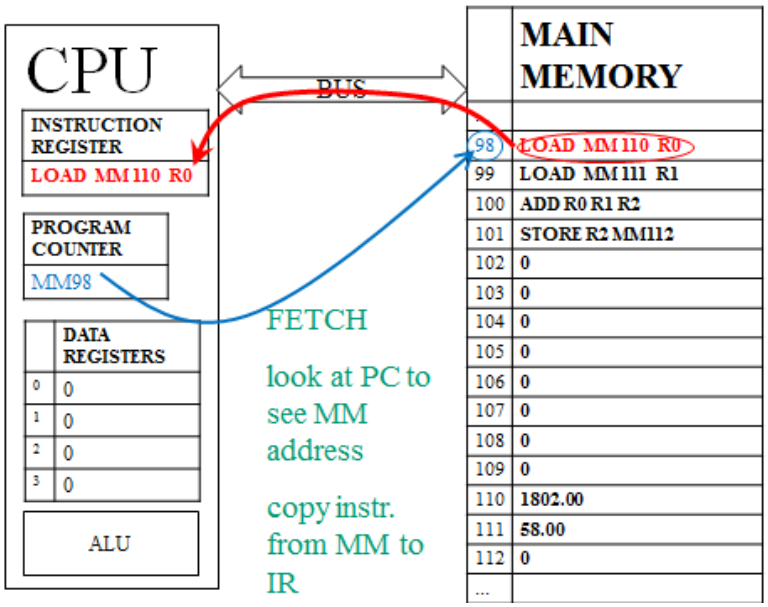
The Instruction Cycle is a four-stage process that the CPU repeats over and over. The stages are: Fetch, Decode, Execute, and

Update. Since this is a cycle, after each Update the CPU continues on to the next Fetch.

The purpose of the cycle is to complete an instruction. The Execute stage is where the instruction is actually carried out. During Fetch and Decode, the CPU prepares for Execute, and during Update, it prepares for the next cycle.

Fetch

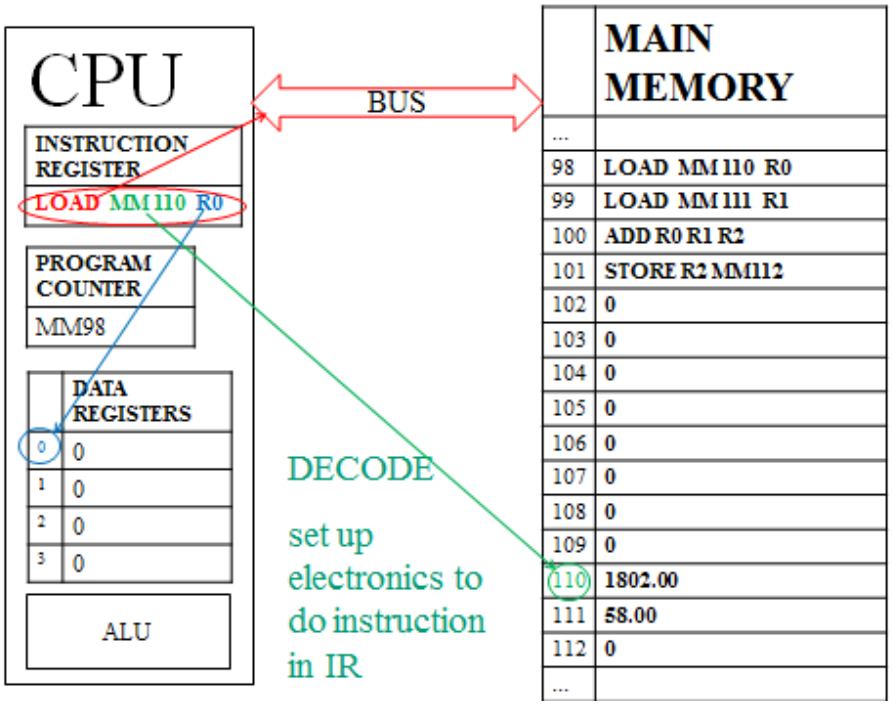
Just as the ALU can only operate on data that is in a data register, the CPU can only execute an instruction that is in the instruction register. So the first stage, Fetch, is to copy the instruction to the IR. But with potentially millions of instructions in MM, how does the CPU know which one to fetch? This is the purpose of the Program Counter. The PC holds the MM address of the next instruction to fetch.



So, during Fetch, the address from the PC is sent to MM, which sends back whatever instruction is at that address. This instruction is copied into the IR.

Decode

Once the binary pattern is in the IR, the CPU can set up to actually do the instruction. The ons and offs in the binary pattern activate some parts of the CPU and deactivate others in preparation to carry out the right task.



For example, if the instruction says to have the ALU add numbers, then the data registers containing those numbers must be activated, as must the adding component of the ALU, and it must be set to put its result into the correct data register. If the instruction says to copy data from MM to a Data Register, the address of the data must be sent over the bus, and the bus set to copy the data into the correct register when the MM sends it back.

Depending on the CPU, it actually may require some very complex actions to set up for execution based on the instruction. If the binary pattern copied to the IR turns out not to have been

a valid instruction, the CPU will set itself up to alert the system of an error[4].

Execute

Finally the CPU can do what the instruction says. What happens during Execute depends on the instruction. For some instructions, the ALU will be performing mathematical operations, for others, data will be copied over the bus, for others, the PC will be changed, or a request will be sent to the HD or... etc. etc.

Note that sometimes during Execute we are copying data from MM to a data register. This is entirely separate from the fact that we *always* copy an instruction from MM to the instruction register during Fetch.

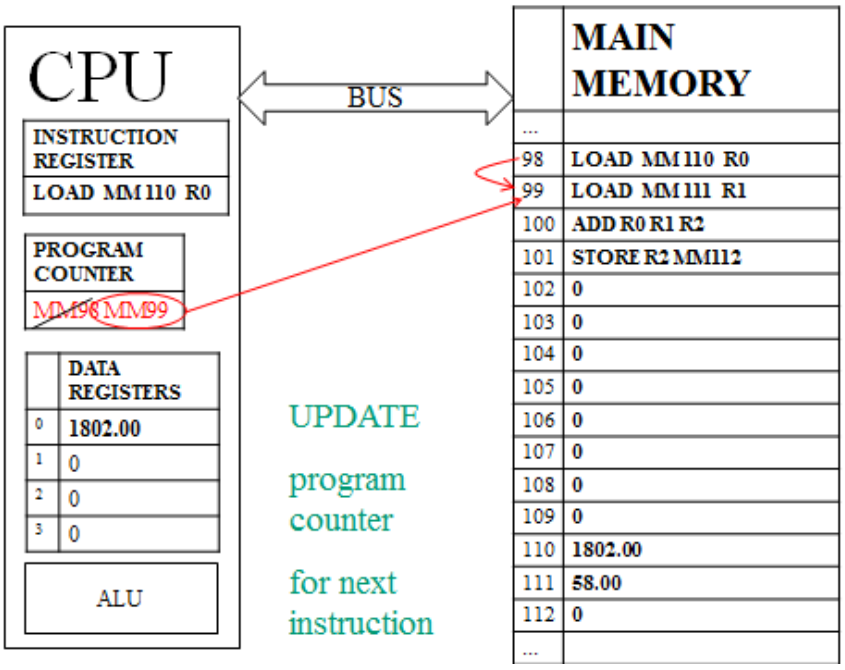
The CPU does not check whether the instruction makes any sense from a human point of view. As long as the instruction is one of the ones built into the CPU, it simply does what it is told.

Update

Once the instruction has been executed, we are done with it. But we want to continue on to the next instruction in the program when we do the next cycle. If the CPU returned to Fetch immediately after Execute, it would be fetching the same instruction it just did!

---

[4] Generally, it would change the program counter so that the next instruction to be run will be the first step in the program that handles such an error.



During the Update stage, the CPU adds one to whatever value is in the Program Counter, so that on the next Fetch it can do the next instruction waiting in Main Memory.

This happens no matter what. The CPU can't see what's in Main Memory, so even if there isn't a valid instruction at the next MM address, the CPU will still update the address by 1. This also means that if we want our program to change the PC to jump to some other point in a program, or some other program entirely, we'd have to have our instruction set the address to one less than where we want to end up, because the following Update will add 1 to whatever address we choose.

Processing Speed

Since a program is run by doing the Instruction Cycle over and over, we can influence the speed of our programs by influencing the speed at which the cycle happens.

The basic approach, in which we do one Instruction Cycle at a time, finishing each Update before starting the next Fetch, is called *serial processing*.

We could speed things up by, instead, doing *pipelined processing*. Pipelining speeds up processing by partially overlapping cycles. Since Fetch always requires copying an instruction from MM to the IR, and getting things from MM is slow from the CPU's point of view, we might as well get that started as soon as possible. So in pipelined processing, we start the next Fetch while still working on the previous Execute. This requires a more complex CPU, to do more than one stage at a time and deal with problems like fetching the wrong next instruction[5].

We could also overlap whole cycles, doing two Fetches, two Decodes, two Executes, and two Updates at the same time. Or four, or eight, or... This is called *parallel processing*[6]. This requires that we have more than one CPU — one to do each cycle. In most cases, instead of separate CPUs, this is done by having a multi-core processor; this looks like a single processor to the rest of the computer, but actually has two or more CPUs inside it.

In a modern system, the parallel CPUs are generally pipelined as well, for even more speed.

Note that having two CPUs will not necessarily make a computer twice as fast at running a program. Many parts of programs need to be done in order and can't be done at the same time; it makes

---

[5] For instance, suppose we just did the instruction at MM99, but what happened during execute set the PC back to MM89, so the next instruction should be the one at MM90, not MM100, which is the one we already started fetching. We need to detect this and fetch the correct instruction instead.

[6] Note that we can also do *networked* parallel processing, by having multiple computers working together over a network, but here we are talking about a single computer running more than one instruction at a time.

no sense to do a calculation at the same time that the data it needs is still being fetched, or to store a result at the same time as it is still being calculated. Increasingly, programs are being written to take advantage of parallel processing, but it is most beneficial for speeding up running more than one separate program at the same time.

A different approach to speeding up the system is to simply do the instruction cycle faster. The system clock is like a metronome that ticks regularly to keep all the parts of the system running at the same rate. Typically one instruction cycle happens for each tick of the clock. If we increase the speed of this clock, which is called *overclocking*, the cycle will happen faster.

Obviously we can only increase this speed a little and still have all the parts of the hardware keep up, and when the system runs faster, it also runs hotter, which can do damage and uses more power.

Some systems can be overclocked through software, others require setting something in the hardware, and others have already been set to the highest speed their hardware can handle when they are manufactured.