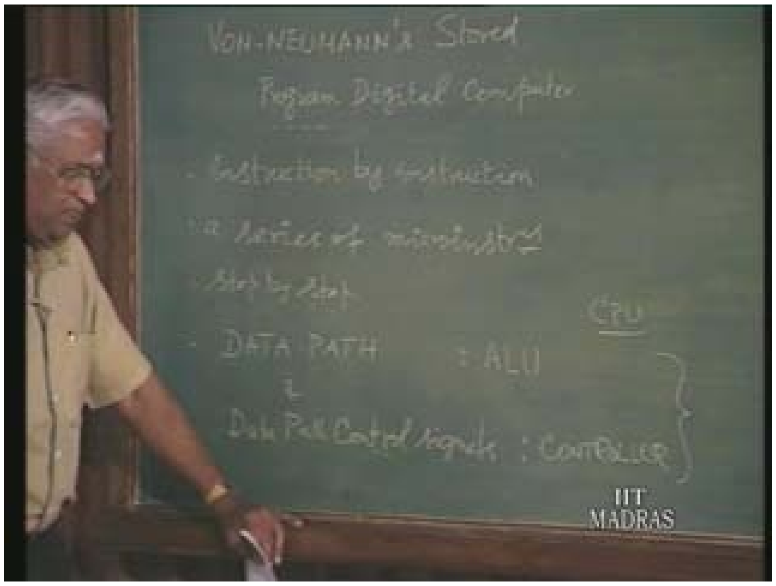
Computer Organization
Part – I

Prof. S. Raman
Department of Computer Science & Engineering
Indian Institute of Technology

Lecture – 14
Problem Exercise

In the previous lecture we worked out a few examples. So with those examples it must be very clear what exactly a processor is doing. Now let us just quickly review and then move on to the next part of the series, that is, about the memory. What did we start with? We started with saying that by and large we have what is known as Von-Neumann's model, that is, Von-Neumann's specifically called stored program digital computer. The architecture of this is what is there in most of the computers even in the present day. So the specific thing about this architecture is that the stored program digital computer is essentially that both the program and the data of it are stored in the memory.

(Refer Slide Time 06:08 min)



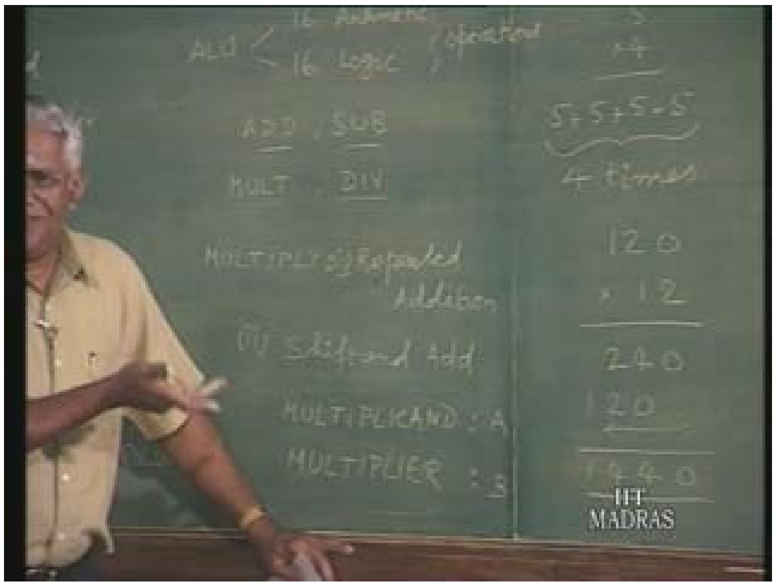
So the computer takes the program, executes the program instruction by instruction – it takes instruction after instruction and then executes that; then it moves on to the next instruction – that is what has been going on specifically. Now for executing one instruction, the computer will be invoking what are called a series of micro-instructions. A series of steps in one specific implementation; you may call them micro-instructions. We also do without micro-instruction; that is a different thing altogether. Then these micro-instructions: what do they specify? Essentially this thing gives a small step in the execution of an instruction. As just as program is executed instruction by instruction; an instruction gets executed step by step and this small step essentially corresponds to a micro-instruction in one implementation. So it is a small step. Now the controller will be issuing the necessary control signals, so that for the execution of the instruction, a path may be set up. That path depends on what exactly the instruction says.

Suppose you have an arithmetic instruction, then the path will be such as to carry out that arithmetic part, meaning, to execute that arithmetic instruction. So the architecture essentially consists of setting up a data path for executing an instruction. It is the only path over which data flows; and for setting this data path we need the appropriate control signals. So that is generally called data path control signals. Now there are two parts of the processor, that is, what is essentially concerned with a data path will include the necessary arithmetic and possibly logic circuits, and specifically one component will be called an arithmetic logic unit. This is only one part of the processor; the other part of the processor, which is consigned with issuing the necessary control signals, is the controller or just the control unit. Together, this ALU and controller form the processor or what we may call as a central processing unit; this is what we had seen. If this controller were implemented using micro-instructions, we call it micro-programmed controller. There are other ways of implementing this also.

Normally in a course on computer organization, a lot more may be said about this arithmetic processing also. I personally feel it is not necessary – I will just explain the basics behind this and then leave the rest to you to work it out. For instance, recall what we were talking about: we took ALU as an example, which can execute something like 16 arithmetic instructions. That is, an ALU is capable of carrying out some 16 arithmetic operations and, let us say, 16 logic operations. These are the operations and the way you arrange these operations could depend on the precise instruction that is to be executed. In a set of these 16 arithmetic instructions, you may include, for instance, the basic things, add and subtract. These are the two basic arithmetic instructions. There can be some more complex arithmetic also. The next step even in arithmetic could be, for instance, having a multiply instruction or having a divide instruction. Suppose we want to add this multiply instruction and divide instruction also, you need the necessary unit for this.

For instance, we may say add multiplied unit and add a dividing unit – you might say that. But then, is it always necessary? No, mainly because, for instance, multiply can always be carried out with some series of additions, which means multiply can be carried out with repeated additions. Now this is one that we may call an algorithm. Instead of repeated addition, for instance, the usual paper and pencil method would be the one with which you are familiar – that is shift and add algorithm or method. Now in the repeated addition let us see what it is. If it is straightforward when we multiply, we will be having a multiplier and a multiplicand. Now these are the two data.

(Refer Slide Time 13:13 min)



Let us call multiplicand as A and multiplier as B. Then the repeated addition would mean add A B times; that is, for instance, if you have something like 5 into 4, to get this particular result, that is the product, we can do a repeated addition – a repeated area addition could basically say add 5 four times; this particular thing can also be got like this. That is what we mean by saying repeated addition. Instead of that, you can also carry out this for any data. So let us say if you have a data like 120 and that has to be multiplied by 12 that same thing holds good – 120 must be added 12 times.

Now let us see the next algorithm, which says shift and add. We are quite familiar with this; there is no problem about that. What is that? Generally what we do is we know this is the multiplicand and this is the multiplier, so the algorithm could look at the multiplier and take only one of the n digits of the multiplier. So we take this least significant one, and then multiply that with a multiplicand; so in this case we will be getting 240 and then, when this digit is over, we move on to the next digit and then multiply that. In that case you have to shift multiplicand because for 1 into 120 or whatever, you have you actually shift and put it; you do not put it right below, but you shift it and then you add. So we will be getting 1440. These multiplication algorithms can be based on the repeated addition method or on shift and add. The software is related to this because an algorithm essentially tells us how a particular operation must be carried out step by step. Now when you say it is repeated addition, then it is enough if you have an adder. When you go for shift and add, we need an adder and then we need appropriate shifter also. How exactly do we carry out two times 120, which again is multiplication? It again is the repeated addition. Of course this problem will not be there if you are doing with binary number because the binary number is going to be 1 or 0.

Now let us just see what we mean by that. Suppose we have a multiplicand 1101and a multiplier 10, which is binary. Let us just do as per the shift and add; then it says the multiplier consists of 2 bits. Take the least significant bit – I am speaking about the algorithm – multiply if it is 0. If you multiply that with the multiplicand, it will result in 0. So we can say that if the multiplier bit is 0, the temporary product or intermediate product is 0; then we move on to the next significant bit. If it is 0, repeat it because it will again be 0; if it is 1, just write the multiplier and multiplicand, but shift it appropriately as I have done here. Then you add this; there you have the binary product.

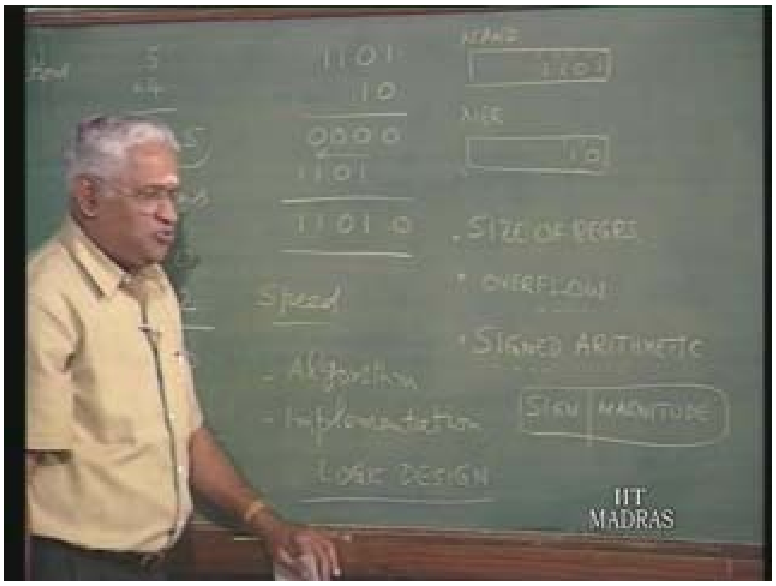
Now compared with this non-binary multiplication, it is easier because it says if the multiplier bit is 0, just the intermediate product is 0. If the multiplier bit is 1, check the multiplicand; but then, depending on the position of the multiplier bit, you shift as many. With this particular one, you just have to shift by 1 bit and then you add. We can work it out for other numbers also. Once you have this algorithm, there is no problem; so we need not really add the necessary arithmetic circuits or units because just like multiplier repeated addition, division can also be looked at as repeated; subtraction can be carried out but then there is an advantage if you have a special hardware circuit, let us say, for the multiplier or for the divider.

This way the implementation will be faster, because anything done through software means step by step and step by step could mean more time. Whereas here we can see that it can be done faster. For instance, even between these two repeated additions shift and add, repeated addition will take more time whereas shift and add will take relatively less time. So actually it is the speed of implementation, which will determine whether you should have a special unit, let us say, multiplier. Once you know what the algorithm is then it can be implemented and then implementation of that particular thing is essentially a matter of deciding on a few things like the register size.

Then, as the algorithm says, whatever you have to do will have to be done appropriately and generate the necessary control signals. So essentially it means for any arithmetic processing, first the algorithm must be decided, and then, after you have chosen the algorithm, it must be implemented. For the implementation of that, once you know what the algorithm is, then it is just based on the logic design principles, with which I am pretty sure you are all quite familiar. Basically, it will boil down to a simple logic design work.

Nevertheless, a few more points deserve to be noted here. Suppose we have an 8-bit CPU. It means the internal registers of the processor or CPU will be 8 bits; the ALU will be 8 bits and so on and so forth. Even in this particular example that we have taken, the multiplicand is a 4-bit number and the multiplier is a 2-bit number. It is alright for us; we are quite familiar with this, but then, when it comes to hardware implementation, which of the 4 bits in the 8 bits it has to be fitted in and which 2 bits of these 8 bits will have to be fit? For instance, we talk about a multiplicand: say a register is 8 bits.

(Refer Slide Time 23:45 min)



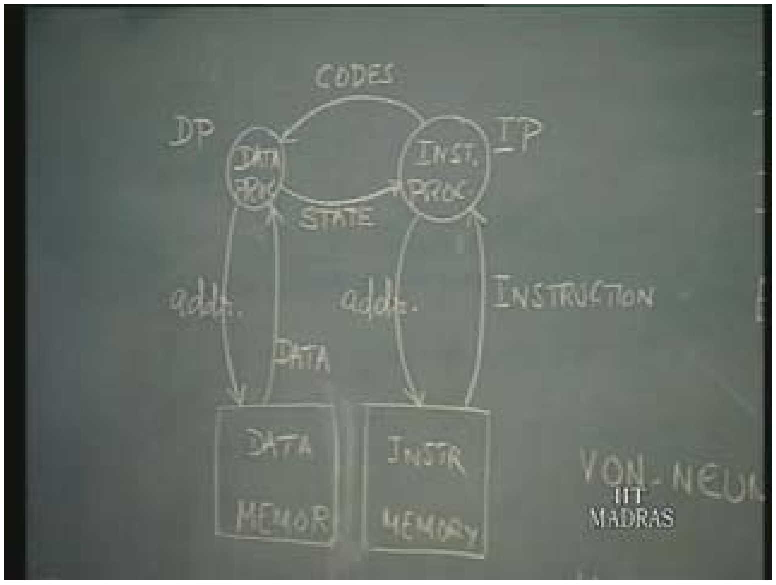
The least significant of these, say, 1101 will have to be entered. Now that is for the multiplicand; similarly for the multiplier this will have to be entered, not anywhere, but in the last. Then after looking at it, we may keep shifting here. For instance, multiplicand can be a bit number; multiplier can be another 8-bit number. When you multiply, you will be getting as much as a 16-bit result; this is a possibility. We really do not have 16-bit registers anywhere.

So these are certain things which will have to be taken into account in the design. So in other words, we may say that the design actually has the size of registers but we may not really know the size of the multiplicand and the multiplier. It depends on the problem but we have to necessarily handle only an 8-bit number, which means a maximum 8-bit multiplicand will be there, but the result will be 16, which will have to be accommodated in two registers, and the data path control signal must generate an 8-bit part of the 16-bit result. One part, the least significant part will be one register and the other one must be in other register. So the control signals must be generated from that point of view. In the case of 16-bit result, it will go out of this particular register, so we are moving it. That is called an overflow condition, a condition such as that, an overflow condition, is a possibility. How exactly are we going to handle this overflow situation?

So far we have not taken the sign of the data anywhere, but then, we know the numbers also have to be represented. So we have to take its design into account; so what is called the signed arithmetic is also something that we have to actually bother about. So far, we have not bothered about the sign part or signed arithmetic. The simple thing is, let us say if you add two numbers, something like plus 9 and minus 5, it really amounts to a subtract operation, is it not? If you are adding plus 9 and minus 5, it actually amounts to subtraction.

And in the case of multiplication, suppose you multiply minus 9 and minus 5, the result will actually be positive; the product will be a positive number, which means the algorithm that goes for the magnitude part of the number is different from the algorithm that is needed for the sign part, because any number will consist of the sign and then the magnitude. So there are two parts to the algorithm for the magnitude; in all these things we had not taken into account the counter sign. The algorithm for the sign also must be taken into account; one is the size of the result and registers, because these registers are present. We have to bother about the size, so that we will not lose any part of the result. Overflow is one situation and another thing is the algorithm, related to the sign.

(Refer Slide Time 35:36 min)

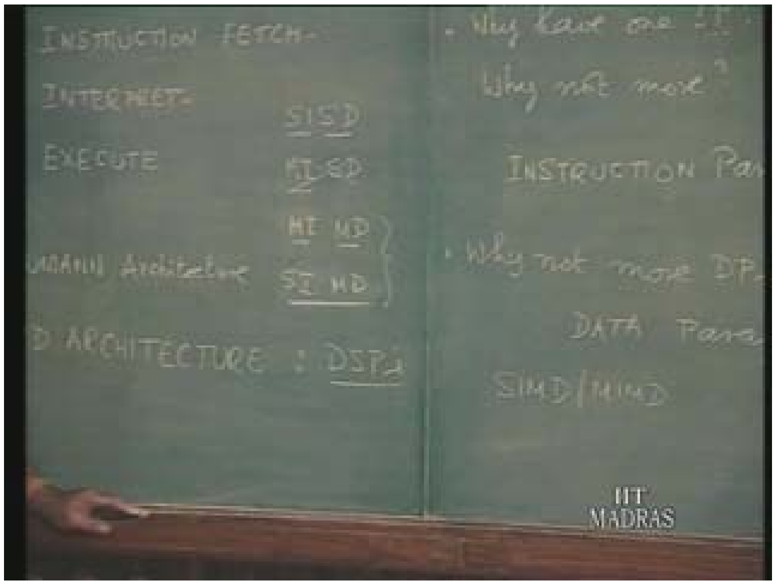


These are certain things, which will have to be taken into account, and another important thing is how exactly a number is represented. If you take the sign also into account, then as shown here, you can always do it like plus 9 and minus 5; that is, sign part and the magnetic part as I was telling. You also have other forms of representation, which is known as complement representation. I am pretty sure all of you know about that. There is another form; that is, a complement representation, and we all know that there are at least two things called 1’s complement and 2’s complement representation in binary numbers. In case the number is represented, say, in 2’s complement, the algorithm will change. That is, we have a different algorithm for 2’s complement representation, another algorithm for 1’s complement and a different algorithm for sign magnitude and so on. So these are the things, which must be considered. Once you consider all these, it just boils down to a simple logic design problem.

Now, let us see how exactly the CPU deals with it. If the special multiplier circuit is there, then the multiply command will activate the multiplier circuit, and inside the multiplier circuit, the necessary control signals will be such that the data path within the multiplier will be set up for the resultant product. If the multiplier is not there, it will make use of the general ALU chip and then two repeated additions or some such thing will be carried out.

That means the movements have an algorithm; we just tell you the software part and there may or may not be a direct hardware implementation of the algorithm. So that is the particular thing, which you have to take into account. In other words, the data path design can be enhanced by considering all these additional units. That is, in a data path, whatever you have considered earlier is the main thing, and data path design can be enhanced by having these extra hardware units. Essentially, it is a question of enhancement, as I said, with special purpose hardware. In other words, it is hardware implementation of software. Now why do we do all these? As I already mentioned, it is essentially to see that the speed with which we want the arithmetic or any other operation can be carried out with the respective unit.

(Refer Slide Time 41:53 min)



INSTRUCTION FETCH-

INTERPRET- SISD

EXECUTE MISD

MIMD

ANN Architecture SIMD

D ARCHITECTURE : DSPs

• Why have one ??

Why not more?

INSTRUCTION Par

• Why not more DP

DATA Para

SIMD/MIMD

To round up our discussion on the processor, what did we see? We were essentially talking about stored program computer, which has the memory. Now let us just represent memory as a block: the memory contains both instructions and data, and the processor takes instruction from the memory because the memory includes the program. The program consists of a series of instructions; the processor takes the instruction from the memory. We said that it does so by addressing – send a code, that is, the address of the instruction, which it gets from the memory, and later on, after it gets the particular instruction code, it decodes it, and internally generates a certain code which may result possibly in fetching some data. The processor, in that case, will have to again do a similar thing, that is, it takes the instruction and generates a set of codes; for instance, these codes can be anything, even the micro-instructions.

Let us not bother about it; we are assuming the processor continues to access the memory this time for getting the operand or the data from the memory, because that also is available in the memory as, essentially, that is what we said is the stored program concept. Now the processor is accessing the memory; it gets the instruction; that is what our basic cycle is. We said earlier the instruction cycle consists essentially of fetching the instruction: that is the first phase, and then interpreting or decoding that particular instruction, and in that process it will internally generate the necessary codes, and then it goes into the execution phase or the execute phase.

If necessary, after interpret the instruction, it may fetch the data also, that is, operand fetch, and that is what we are seeing here. It fetches an instruction; internally it would tell what else it wants, say, fetching the data and so on. After the processor has everything from the instruction – it knows the operation that needs to be done and it has the necessary data – it executes the instruction. At the end of that instruction cycle, that is, after it executes, this will be generating certain information; generally it is called the state of the processor. Do not confuse this with the state of the state machine; again it is some code, which indicates the state of the process.

Let us say the instruction is an add instruction and after it has got the two data, A and B, the addition may result in a carry or an overflow. So that is the state it has got to register, because based on that, the next instruction will have to be fetched. That is the state we are talking about here. This part of the processor may be appropriately called an instruction processor, and this part may be appropriately called the data processor. So IP or the instruction processor does something and DP or the data processor does something; it is possible that one part of the CPU is concerned essentially with fetching and decoding instruction; and another part of the CPU is essentially concerning itself with the data part and the execution of it. This particular architecture is what have seen as Von-Neumann's architecture – Von-Neumann's, as we said a little earlier, is a stored program digital computer architecture. Even to this day, 90% of all the computers still follow this architecture.

The first major change in this can be that we can start from anywhere; let us say, you split this memory into two. Suppose I split this particular memory into two and I keep the instruction part here and the data here; this also is memory. Now you can see that the two parts of the processor can carry on some activities independently because if it has a single monolithic block, then, when the instruction processor accesses the memory, the data processor cannot. But now, if I keep them as two separate units, when the instruction processor is fetching the instruction, the data processor can also fetch the data – that parallel thing is possible. This first step, that is, having two different means of accessing the respective part of the program, that is, the instruction and data, which really gives rise to a data bus separately and an instruction bus separately – not having a single common bus – has, in fact, given rise to what is known as the Harvard architecture, in which you have these two buses separately. You have them separate, so that it can carry on both these simultaneously.

This improves the speed of execution. In fact, Harvard architecture is essentially followed in DSPs, that is, digital signal processing chips because there we have huge amounts of computation to be carried out. In the case of digital signal processing, sometimes the data may be less, but the amount of processing is enormous. In those situations, it may be better to improve by bringing in, as I said, some kind of parallelism. That is what has been achieved here by splitting and having two separate buses – an instruction bus and a data bus separately. Now this is one aspect of it; we have already seen the other aspect even without mentioning. The circuitry, which deals with the instruction and the circuitry which deals with the data within the CPU have already been segregated here. This is also some kind of parallelism. The second thing, which it automatically leads to, is given here in this diagram. The first question is, you have one IP – why have only one IP? Why not more? We have only one IP; we can have more IPs, in which case, we can simultaneously fetch a lot of instructions and carry them out. Now we have the same question – why have only one DP? We will ask that later.

If it is possible to fetch more than one instruction and carry out, then we talk about a system in which we have what is known as instruction parallelism. That is, we started with one instruction and one data; this thing is generally called a single instruction and single data. Now we are talking about multiple instructions and we continue with single data, then when we say that the system is carrying out some instruction parallelism, it may not always be meaningful. It may be meaningful to carry out, in parallel, more than one instruction; but it may not always be that – we will not bother about that now.

The same thing in that is to have more than one IP. If we have more than one IP, can you still continue with one memory block? Maybe; here also we have to have more than one memory block – we will not bother about that yet. We can now say the same about this DP or data processor – why have only one DP? Why not more DPs? In this case, obviously we could be talking about data parallelism, when we will be having a single processor. It may not be very meaningful; we will continue with the same thing: multiple instructions, that is, you are continuing with more than one IP and now data also is multiple – we have multiple instructions and multiple data. So we can have more than one DP; correspondingly we can have more than one data memory. There are other issues related to it, but just look at it. The reason I am giving this particular figure is that you can logically proceed and then go on adding more and more and then try to understand what other issues the particular thing brings in – that is the main thing here. Instruction parallelism and data parallelism come from this simple thing.

All these things are made possible because of the technology – the processors have become cheaper, so what was not possible earlier, such as, having 100 processor has also become possible; it has become cost-effective; but the application must support this. This is important because we may not be able to carry on multiple instructions; you may not be able to carry on multiple data processing all the time; it depends. Now let us work out the other combinations possible here, that is, we have the instruction stream and data stream and what we call single and multiple. So now what we are left with is single instruction and multiple data; that is the only thing.

Earlier, we discussed these multiple things: single instruction and single data is quite conventional. Multiple instructions multiple data is also fine because you can have more than one instruction and more than one data; and then we discussed single instruction and multiple data. Any time multiple data can be processed with one instruction and another instruction and so on and so forth. But take a look at this multiple instruction and single data – what is this particular thing? This is what we started with; nevertheless let us take a look at it. You have only one data and then you have multiple instructions. What happens at the end of the multiple instructions? Execution of one set of multiple instructions on single data will obviously lead to multiple data; we will come back to this. So really speaking, this is not practical – this will not lead to a practical system. So generally, we when we talk about parallel architecture, we talk about this multiple instruction multiple data or MIMD and single instruction multiple data or SIMD.

Generally we talk about MIMD or SIMD architecture as far as parallelism is concerned; the other one is a straightforward, single thing. We would take a look at more of these later on.
This is just to give you a glimpse of parallelism, which naturally comes from this. We may say this is a primitive block diagram, and there are other issues because when you have multiple data and even if it is single or multiple instructions, it does not matter; more than one result will be available here. How exactly can these things be communicated to the other data processors among those n processors? These are all the other issues; so apart from this computation, another issue is of communication among those processors. We will possibly take a look at all these towards the end of this series of lectures.

As the next thing, we will concentrate on how exactly the CPU in general deals with the memory, that is, a CPU–memory interaction. Before that, we know the CPU addresses the memory and then memory responds and so on and so forth. We will take a look at the memory in detail and then the different memory hierarchies. We had talked about it, may be towards the beginning of the series. We will take a look at them and then work out the details and then, we will also take a look at the interaction between CPU and memory. What about IO? That is the third one that is left. Already said IO is an extended memory. We have talked about it; nevertheless, we will also discuss IO later on. In the next part of this series of lectures, we will take a look at the memory. We have learnt a lot about the CPU or the processor; essentially, this is all that you need to remember about the processor for just getting an idea about what goes on further between CPU and memory or CPU and IO or even expanding it further to include all the latest things, such as parallel architectures and so on.