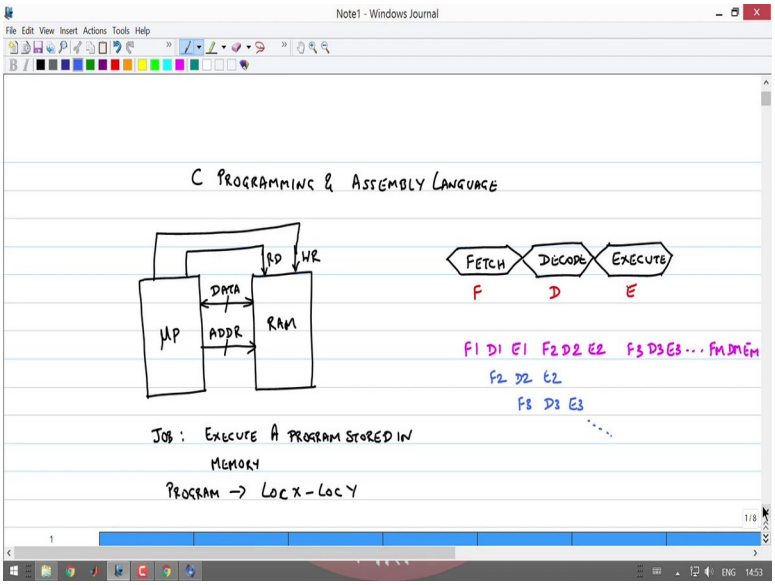
C Programming and Assembly language
Prof. Janakiraman Viraraghavan
Electrical Engineering Department
Indian Institute of Technology, Madras

Lecture - 02

So, welcome back to this course on C Programming and Assembly language. We are in module 1, and last lecture, we abstracted out the model of the memory as follows.

(Refer Slide Time: 00:23)



We said that a memory is a simple black box. It is a random access memory that has a bi-directional data bus and a unidirectional address bus. So, this is my data, this is my address.

So, here I have the microprocessor which is controlling the memory by issuing various commands. So, this is a mu P and the memory also has commands called read and write, and these are not being controlled from elsewhere. The microprocessor again issues these commands at appropriate times in order to read or write to the memory. So, in this lecture let us look at the necessary abstraction of the microprocessor that we need for this course.

So, let us start with the basic utility of a microprocessor. A microprocessor is meant to execute a program that is stored in a memory. So, the job is to execute a stored in memory. So, let us assume that this program is stored from location X to location Y. So,

let us assume that there is a location X in the memory where the program starts and a location Y where the program ends. The job of the microprocessor is to fetch instruction by instruction from this particular starting location and execute one by one until it reaches memory location Y.

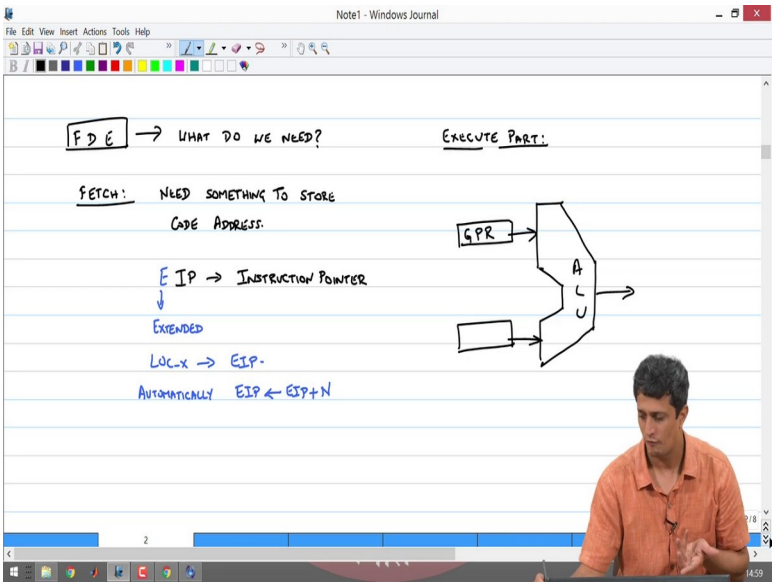
So, traditionally a microprocessor does the following tasks. It does what is known as fetch, decode and execute. The name fetch suggests what it should do. The idea is to fetch a particular instruction from the memory, then decode it and execute the instruction accordingly. So, if I denote this fetch by F, decode by D and execute by E and let us assume that there are M instructions between location X and location Y.

Then what the microprocessor does is, it actually fetches the first instruction, decodes the first instruction and executes the first instruction. Then it fetches the next instruction, decodes the next instruction and executes the next one and so on; F3 D3 E3 and all the way to FM DM EM. At this point I have to mention that microprocessors have come a long way where they do not just execute this fetch, decode execute cycle serially it does it in what is known as a pipeline manner.

So, while the instruction is being fetched and decoded for instruction 1, and then the first instruction is being executed in the microprocessor the data bus happens to be idle during the execution phase. And therefore, you can pre fetch the next instruction in that period. So, F2 D2 and E2 will happen like this F3 D3 E3 happens and so on. So, you can clearly see here that in modern microprocessors because of this concept of pipelining instructions are pre-fetched and they end up getting a much better throughput from the particular microprocessor.

So, in order to perform this fetch, decode execute process in a loop we need to see what all a microprocessor has to have and that is the exact abstraction that we are going to deal with in this particular lecture.

(Refer Slide Time: 05:39)



So, F D and E in order to perform this what do we need? So, let us start with fetch. You have to go and fetch a particular instruction from the memory which means that I need some place in my microprocessor to store an address of where this code exists in memory. So, the first requirement is need something to store, code address and this is precisely satisfied by what is known as the instruction pointer register.

So, I have an IP which is INSTRUCTION POINTER and if in 8086 it is 16 bits in length and the instead of the register was called IP as you move forward into the other x86 architectures, 286, 386 and down to the Pentium 4s this would essentially just be called EIP, where this E stands for extended and this is true for any other register in the microprocessor as well as you will see a little later.

So, the way we start is we load this location X into the instruction pointer and tell the microprocessor to start executing the instruction from that particular location. So, here you load location_Xinto EIP and you start the execution. So, as the microprocessor fetches a particular instruction at location X and decodes that instruction the microprocessor will know exactly where the next code or the next instruction is in memory. So, therefore, it can automatically calculate what the next address should be and where the next instruction is located.

So, automatically EIP is now incremented plus N; N could just be the next byte or it could be the next word or the next d word, it depends on the instruction size ; it depends

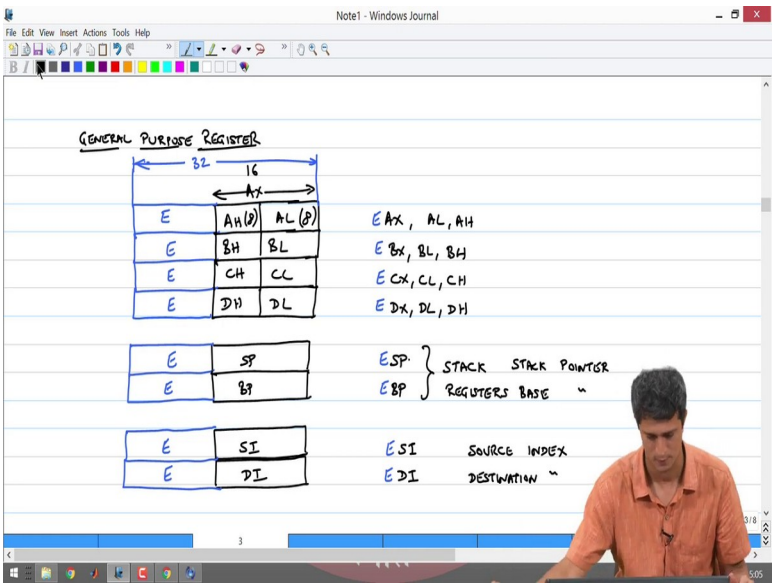
on what is known as the Opcode size. So, the microprocessor during the decode phase knows exactly what this number N should be and automatically increments the instruction pointer to the new address and continues this fetch decode execute process. So, that is with regard to the code.

Now, let us come to the execute part of the instruction. The decode by the way is pretty straightforward. It is just a combinational logic where you decode what the instruction should be and invoke you know those registers and so on. So, there is nothing specifically needed for decode as far as our abstraction goes. So, let us now look at the execute part.

The execution typically is going to refer to some sort of an arithmetic or logical instruction or it could even be some sort of a data movement from one location to another. So, what do we need in order to implement this execute part in a microprocessor? So, typically if I take my ALU which you know is something like this, my ALU. You have two operands right; operand 1 and operand 2 and typically these at least one of this will end up being a general purpose register.

So, you have a general purpose register here and the other operand could be something more generic. We will come to what those operands could be . So, in order to perform any ALU operation we need to see what are the kind of general purpose registers that are available in a microprocessor.

(Refer Slide Time: 11:15)



So, let us look at. Traditionally, every the 8085 microprocessor implicitly needed to have one of the registers hard coded and fixed as one of the operands to the ALU and this happened to be the AX register or the accumulator register. So, a stands for accumulator here and if you look at the size of this register in the 8086 microprocessor it is 16 bits in length. So, you have AL which is the lower 8 bits, AH which is the higher 8 bits and this is AX the combined register which is 16 bits in length.

So, this is 16, each of this happens to be 8. And, just like the instruction pointer where we had an extended instruction pointer for 32 bits the higher 16 bits is referred to as the E AX. So, the 32 bits is E AX . So, here I have E and AX apart from this you can also access 8 bits of AL or. So, note that you cannot access the higher order 16 bits of E AX directly as the register.

Similarly, I have the other registers which are basically my registers BH, BL and the extended value here again . So, this is E BX, BL and BH; and similarly for CH, CL and my E. And, of course, I have one more register called the DX register DL and my extended value E. So, I have CX, CL this is also H CH, DX, DL and DH and my extended 32 bit registers as well.

So, you will find that typically for any of the addition subtraction you can use a combination of any of these two registers, you can also use these registers as pointers to something in the memory which we will come to a little later when we do the move instructions and the ALU instructions in some detail .

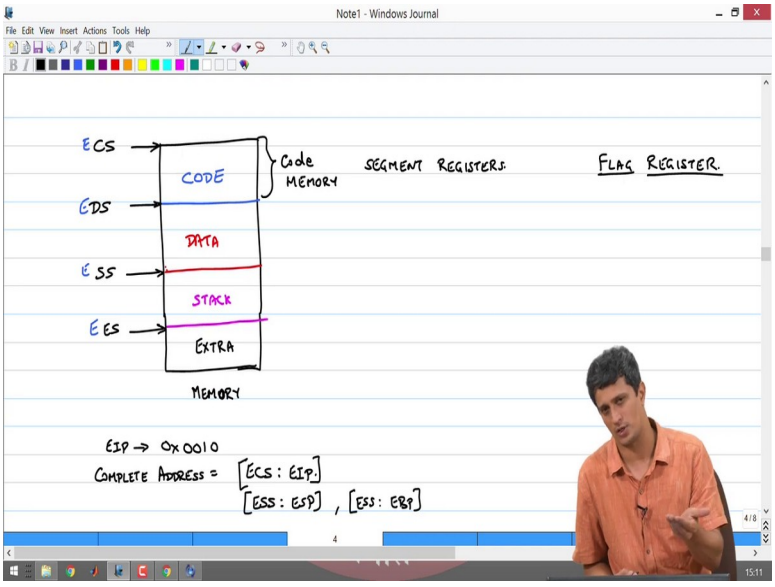
So, apart from this we also need something known as a stack. A stack is a last in first out kind of memory and in order to keep track of the top of stack we need a specific register and that is known as the stack pointer SP and the extended value 32-bit version is E SP. Unfortunately, it is not sufficient if you are able to do just the last in first out kind of operation with the stack. We also need to be able to do some amount of random access and therefore, we have another register which is known as the base pointer BP and E BP . So, these are my stack registers.

It also turns out that we are able to do some sort of complicated array manipulation or string manipulation at the hardware level and for that we have two other registers which are known as the source index and destination index SI and DI. So, SI and DI this by the way is the stack pointer and this is the base pointer. Similarly, this is the source index

and destination index. Like every other register I have a 32 bit version of this as well in the modern microprocessors E DI.

So, we will look at some detailed examples of how all of these registers can be used for specific instructions and how they can be exploited to speed up certain functions in C as well later in the course.

(Refer Slide Time: 18:11)



So, till now we have been referring to certain address pointers in memory. For example, the instruction pointer accessing the code, we have the stack pointer accessing the stack and so on. So, the question is how do we partition the memory into different segments. For example, I have a large memory. This is my memory what I want to ensure is the segment where the code is stored is different from where my data is stored where my stack data is stored and so on.

And, this is enabled by another set of registers which are known as the segment registers. So, we will talk about segment registers. So, I have my segment registers which are simply going to demarcate what the different kinds of memories are in a microprocessor. This for example, is my code; this could be my data; this could be my stack and this could be my extra segment.

So, now the question is how do I decide that location X to location Y or location A to location B is meant for code; location B plus 1 to C is meant for data and so on. So, code

memory that is achieved by having one pointer to the starting of my code segment as my CS register. Similarly, pointer to my data segment is my DS register, pointer to my stack segment is SS register, pointer to my extra segment is ES register and of course, the 32-bit extensions are just E DS, ESS and EES so on.

So, if I now have a particular instruction pointer EIP available now ; let us say that there is some address it has 0x0010. What is the complete address of this particular location or where is this location stored in the external memory is the question. So, the way that complete address is constructed is using a combination of the segment register and my instruction pointer register.

So, the complete address is my ECS code segment register and EIP. This is my complete address. Similarly, if I am dealing with my stack segment then the complete address would be stack segment registerESS; ESP or you know ESS: EBP. So, by default, the stack the stack pointer and the base pointer are associated with the stack segment. The instruction pointer EIP is associated with the code segment register the data registers ABCD are associated with the data segment, the EDI and ESI are associated with the data segment and extra segment respectively.

So, with that which is a combination of general purpose registers and some segment registers, we are able to address the entire memory to access code and data. The last register that is needed in order to complete this execution process and is almost mandatory in order to do branching and looping is known as the flag register. So, the flag register is just an indication of a result because of an ALU operation, arithmetic or logical operation in the microprocessor. For example, if I subtract two registers and it happens to go down to 0 then the 0 flag will be set. If I subtract two numbers and the result is negative, then you have a particular sign bit that is set.

So, you can check the value of these bits in the flag register and make certain decisions in order to branch and loop. So, in summary we have now discussed a bunch of registers that make up a microprocessor in order for the microprocessor to execute this fetch, decode, execute in a loop again and again.

(Refer Slide Time: 24:05)