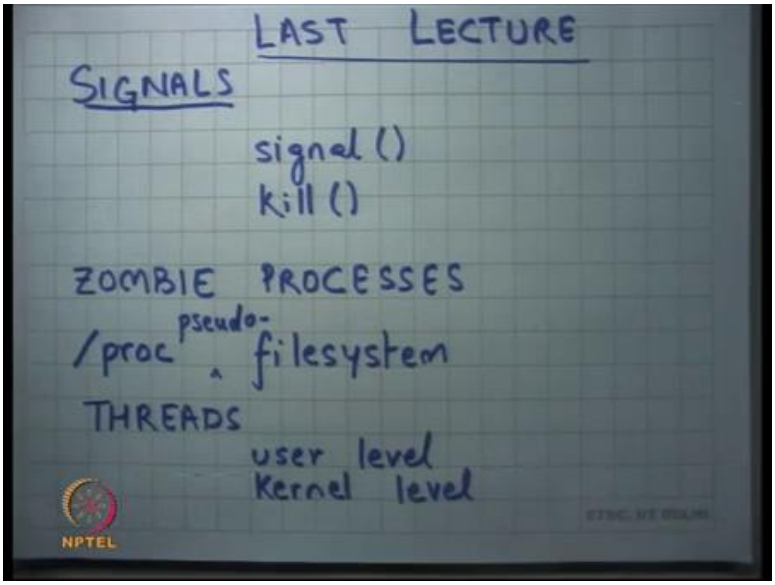
Operating Systems
Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 04
PC Architecture

We welcome to Operating Systems lecture 4.

(Refer Slide Time: 00:33)



In the last lecture we looked at Unix signals. Signals is another way of doing inter process communication or communication from the OS to the process as we saw. And it is useful if the communication indicates a rare event, a relatively rare event for which the process may not be ready. And so, it is delivered to the process in a very asynchronous fashion.

You can think of it like there is a process that is running, and another process wants to bring to send something to another process. So, process p1 wants to send an some message to process p2 and process p2 may not be expecting that message or process p2 just does not think that it is worthwhile to keep trying to poll for that message every few you know at pure time intervals.

So, one way to design your processes would be that you basically let install a signal handler in the receiving process. And, in the sending process you send something; let us

say you send something to the pipe and then you send a signal into the receiving process, the signal handler of the receiving process is going to read from the pipe. And so, in that sense you have an event driven message transfer between the sender and the receiver. And the same concept can apply from for communication between the OS to the process and various signals we saw last time; like interrupt segmentation fault etcetera are ways of doing it.

So, it is another way of inter-process communication which is useful in settings where the communication is relatively rare and it needs to be handled in an event driven fashion, ok. So, we saw two system call; signal which install the signal handler and kill which allows you to send a signal to another process. So in fact, when you for example, type control C on the shell what happens is, that the shell passes the control C command and sends a makes a kill system call to send SIGNIT to the process that is currently running in the foreground of that shell, alright.

We also looked at you know we also talked about this concept of zombie processes. So, we have seen that the two system calls as exit and there is wait; a process can call exit, but whatever is the exit code that is returned by the process should be communicated to the parent and the parent can read that through the wait system call, right. Now there is this small dilemma that if the process has exited and the parent has not called wait, what should the OS do, right.

The OS has no idea when the parent may call wait; the parent may call wait one second from now, the parent may call wait 10 years from now or the parent may actually never call wait, right. So, all these are possibilities and the OS should maintain its semantics and all these possibilities. So, what the OS needs to do is it needs to reserve, it needs to store this return value somewhere in it is data structure, so that when the parent calls wait it can serve it to it, right. And so, this extra state of storing this information of the process of the exited process, a process that is no longer living is extra overhead on the OS.

And so, these kinds of processes which are not really living, but they are some state of that some of the state is still living in the OS are called zombie processes, right. And we also said that zombie processes are a very common programming bug, because programmers often forget to call you know wait in the parent, they just spawn a process something exits. And what happens is eventually although the state is very small,

eventually the state is going to keep leaking and so, some at some point in future your OS data structures may actually find very little space for themselves, alright.

Then you also talked about the slash proc pseudo file system, here is an example of a file system which is actually not a file system, it is not it does not have real files, it actually just shares name space with the file system; which means it has you know it has a name which nests inside the file system namespace. A process can call open read write close on this on these names, just like it can call open read write close on names of files. And the OS can use this file system to expose it is own data structures to the process, right.

So, for example, the OS can expose the list of processes or you know that process the concerned resource consumption of the processes or even allow the application to actually set values in the kernel in the OS's data structure, right. I have been using the term kernel and operating system interchangeably, they are mean roughly the same thing in this discussion. It is the kernel is the kernel is the real operating system layer that is implementing all these system calls ok.

So, we looked at the slash proc file system and then we also looked at and then you also said that, you know we talked about processes which are relatively isolated. And, we understand the utility of having processes; because one program does one program writer does not need to be worried about another program writer and you also have trust boundaries, one program does not need to trust another program and so on.

But then you know inter process communication becomes relatively expensive, because it involves system calls and going through the kernel. Often you do not need this level of protection and so, what you could instead do was, instead of use processes use shared address space and allow multiple threads of control within the same address space. And so, these are called threads, right.

And so, threads can communicate between each other in a very fast way, because they share memory; one thread can write another thread can read and so, it absolutely involves, does not involve the kernel at, alright. Some examples of applications where you use threads; let us say you have a browser and you open multiple tabs inside the browser, you know all the code of the tabs, each of the code is actually doing separate things simultaneously.

But this code has sort of mutually trusted, so you do not really need to have separate silos for each tab, they can actually you know run in the same address space that way they consume less memory number 1. Number 2 if they need to communicate with each other for whatever reason let say statistics, then they can do it in a very fast way.

And, but what this imposes a burden on the programmer is that, the programmer should be aware that when you know when one thread is running, another thread could become concurrently running on the same memory. And so, there should be safety in that sense and we are going to talk about you know what kind of safety and how it is ensured later and that is you know that is a very interesting topic in itself.

We also said that there are two types of threads as user level threads, where the operating system is completely blind to what is happening; it is actually just looks at a process. But, inside the process the user can actually orchestrate his logic in such a way that it appears that there are multiple threads and each thread can now appear to run concurrently, right.

There may not be physical concurrency, but there is logical concurrency when you are doing user level threads, alright. And we also said that actually the, you know the system calls that we have discussed so far are enough for the user to be able to implement this layer of abstracting one process into multiple threads, right. And this will be part of your homework problem next week. So, you will see how this is done for example, alright.

Then the other type of threads are kernel level threads, where you tell the OS that yes I need threads and you know the OS understands threads; basically gives you one address space and allows you multiple threads of control. And, that allows you full physical concurrency, if there are multiple CPUs and these threads can be scheduled simultaneously physically. And also, you know one thread could be waiting on the disk and another thread could be waiting on the network and yet another thread could be working on the CPU, alright. So, we saw all this, yes question.

Student: When you switch off the machine does the zombie processes get killed?

When you switch off the machine do the zombie processes get killed? Yes, right. So, the semantics of an operating system typically are that when you switch off the machine all memory state is wiped out, all process state is wiped out, it is only the discontents that

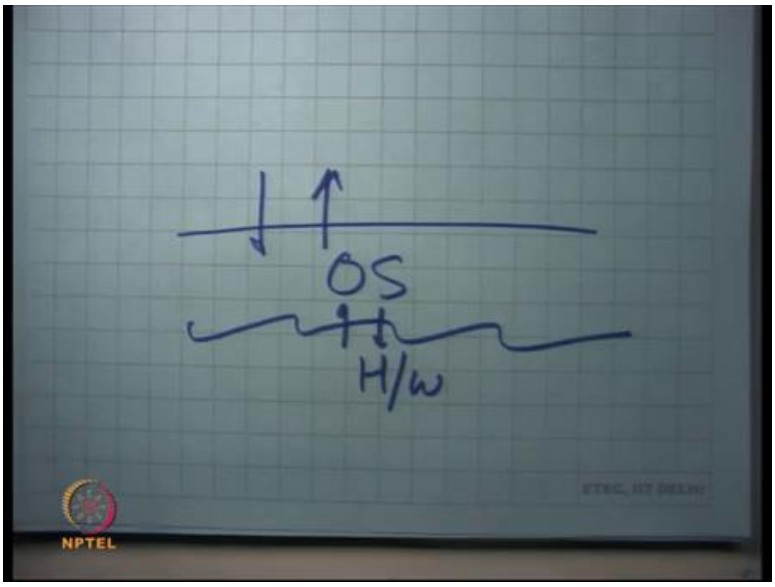
remain. And, we are going to look at you know even the discontents the that we guarantee on what remains and what does not remain and we going to discuss that when you are talking about file systems question.

Student: In the event when parent does not call wait for child and parent exits before child.

Interesting question; so, in the event that the parent does not call wait on child and the parent exits before the child has exited called exit, then what should happen; should the child remain as a zombie process or should it be freed completely, right. So, I mean, so you know an OS typically what it does is if you know, if a child loses its parent then it is an orphan process and that orphan process is actually attached to one of the default processes it is called the INIT process and so, you know if an orphan's process parent is the INIT process.

And so, the INIT process is you know, the code of the INIT process typically would just call keep calling wait over and over against to clean up all the zombies, right. So, that is one way of handing this, alright.

(Refer Slide Time: 09:09)

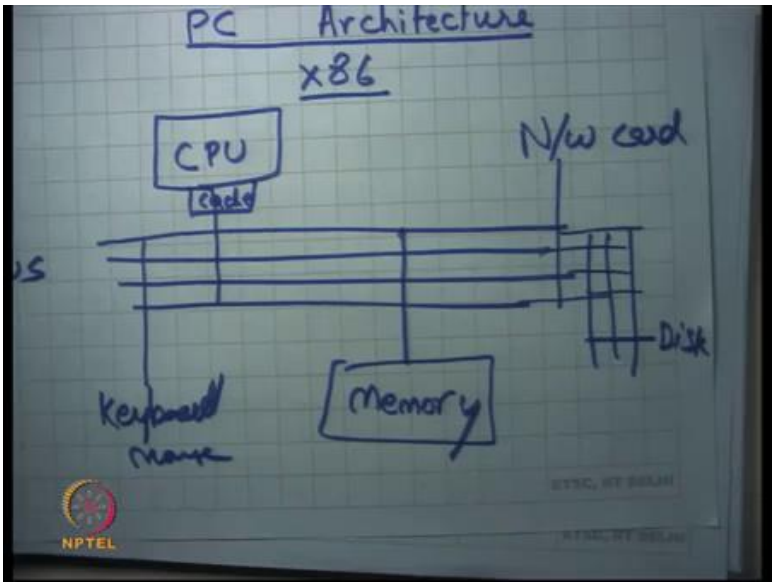


So far, we have looked at the OS and we have looked at what kind of abstractions does an OS provide? Now I am going to change gears and I am going to look at how does the

OS provide these abstractions and we are I mean it is going to be a relatively longer discussion.

But we are going to start with looking at what kind of interfaces does the hardware provide to the OS, right. So, let us look at the other side of how the OS works, you know how. So, let us start by looking bottom up and let us look at what the hardware provides, right.

(Refer Slide Time: 09:45)



And I am going to start, so we are going to have a discussion on the PC architecture or PC hardware. And in particular I am going to discuss the x86 architecture. Now there are many architectures, the many machine architectures that have evolved over the years and Intel's x86 is one of the very popular architectures. And, the reason I discuss it in this class is, because your programming assignments are based on this architecture and moreover most of our machines are actually that we use based on Linux or windows are usually based on this architecture. So, it is also a good practical experience of what the x86 architecture looks like.

As a word of warning the x86 architecture is very complex, much more complex than it needs to be, so we are going to slowly dig through that complexity. But I would like you to pay attention, because so that you understand this stuff and that will literally help you through your programming assignments for example, alright ok. So, let us first understand what a machine looks like, what a computer system looks like. So, let us say

here is the CPU, here is the memory, a CPU has a cache let us say, right and let us say this is the BUS, alright the system bus basically. And, then there are you know other buses attached to this BUS, which are let us say attached to devices which are like disk or you know network card and so on, right.

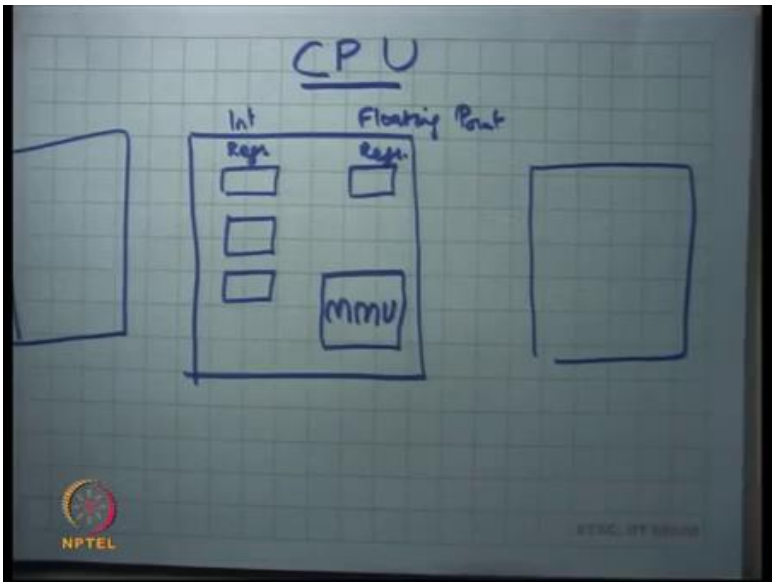
So, a CPU is basically a processor, it is like you know it is a silicon chip that has circuitry built into it. The nice thing about having a small silicon chip is that, you know operations within this chip are very fast, right. So, you can read and write within the chip in a very fast way, you can execute logic inside the chip in a very fast way, right. So, you have all these gates, all these logic gates inside the chip which are executing things that you want to execute. And, then you know there is the bus, this is this is a high-speed bus; you know that is a 10 to 100 gigabytes per second, it is connecting to memory.

And so, the chip can only have that much storage space, because you know that is the physical that is a limitation of the physical technology. And so, but if you allow slightly slower storage, which is you know typically the kind of technology used for physical memory, technologies let us say called DRAM; then you could actually, so you could have larger sizes of this memory. And these sizes could range anywhere from you know a few 100 of MB's to now even few 100 of GB's for example, right. So, memories are actually constantly increasing size in a huge way.

And so, the bandwidth to the memory is actually dictated by this bus. And so, the CPU can actually say that, can send a message to those memory on the bus saying give me the contents of the address x and the memory will reply on this bus saying here the contents of address x, alright. Or so, this is the read operation, write operation similar, the CPU you can say chain the contents of address x with the data D. And so, you know and then there are the memory has certain semantics that if you write to address x data D and you read it later you are going to get the same value that you wrote.

So, those are semantics of memory and they are implemented in a fast way using different technologies, one of the popular technologies doing data alright. And similarly, there is a device, so the CPU has ways of sending commands or device and the device has ways to send results back to the CPU, alright ok.

(Refer Slide Time: 13:51)



So, now, let us look what is inside the CPU. So, let us look at the CPU, alright. So, what does the CPU look like? The CPU has some integer registers, right. So, these are let us call them registers; what are registers? Register is just storage inside the chip, right; it is very fast storage, much faster than DRAM, ok. You can access it as at less than say nanosecond latencies, alright.

Then you have you know let us say these are integer registers and then you have floating point registers and you have you know a memory management unit, MMU. What is an MMU? An MMU is basically an instruct, so the CPU wants to access memory; the memory management unit sits between the instruction that wanted to access memory and the actual physical memory, and does some sort of bookkeeping or translation or access control checks, right.

So, you know an instruction wants to access certain name physical memory, the MMU is going to do some checking or some translation before it actually sends an address to the bus, right. So, some translation between what the instruction says I want to access and what the actual address goes on out to the bus for the memory to read, right that is MMU, alright.

And then you know modern CPUs actually have multiple of these on single chip, right. So, you know when you hear about multi core architectures or you do dual core processors etcetera, it is one chip which has multiple of these inside one. And, now you

also have some logic to basically be able to communicate between these processors. So, I am going to skip that for now and let us just talk about a single processor I make things simple, alright.

Of course, in this picture you know what else remains is basically you know, there is let us say keyboard and mouse etcetera. The way typically you would organize the system, is you look at the bandwidth requirements of that particular device. So, the bandwidth requirement memory is typically very high, because you may want to read you know megabytes of data advance and you want that to happen as fast as possible. Bandwidth requirements for keyboard are relatively very low, because you know how off, how frequently can I use a really press a key; it can at most be you know at millisecond you know, one key per millisecond at most.

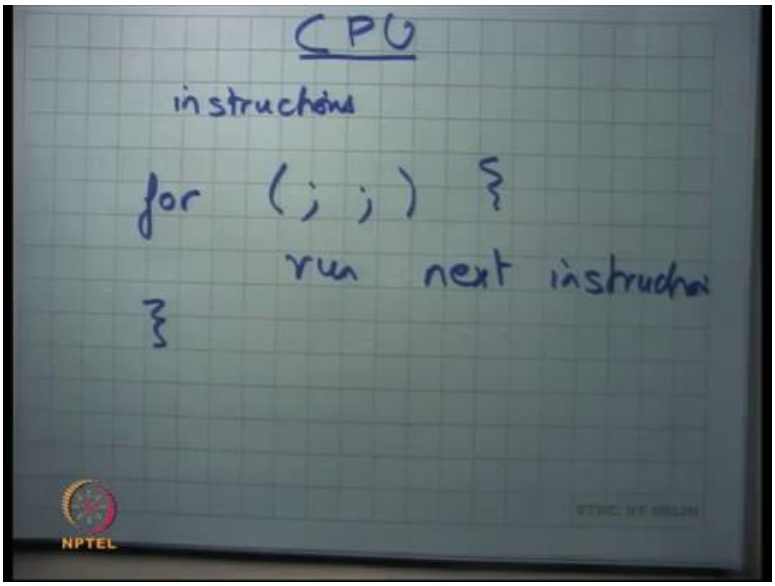
So, you do not need that much bandwidth to the keyboard. So, you would probably have a very thin bus to connect to the keyboard or mouse for that matter. For something like a network card, you know it really depends on what kind of network technology you are using; you know earlier network technology used to be let us say 100 megabytes, so the bus inside the CPU did not need to be that fat to connect to the network card.

Now, you have 10 gigabits or even 40 gigabits Ethernet, so you know the buses inside the CPU also need to be broadened appropriately. The disk, the disk typically you know is mechanical device you know when you actually see the disk moving, it actually moving a spindle; I am going to talk about how a disk works later.

And some mechanical devices do not have that much throughput anyways, so you know typically they you do not need that fatter bus to a disk let us say. So, I mean a hardware designer is going to do all this; engineering is going to say that ok, this is how I am going to maximize my throughput for the devices that need throughput and not maximize the throughput device that do not need through ok, alright.

So, we looked at the state inside the CPU. Now what is the logic of a CPU, right? So, the way CPUs have evolved is basically that, CPUs have you know divided the logic of a computation into hardware and software, right. And so, hardware implements what are called instructions. And software implements the execute these instructions or sequences these instructions to implement it is logic, right.

(Refer Slide Time: 18:09)



So, the hardware designers are basically implementing instructions and the software designers writing these instructions. And the logic of a CPU, is basically an infinite loop; if I were to write it and see let us say is just you know an infinite loop forever run the next instruction.

So, you just say is what is the next instruction run it, what is the next instruction run it that is the logic of a CPU, right. And the software is basically a sequence of instructions that needs to be run and when it in puts it in the CPU with the CPU starts running those instruction.

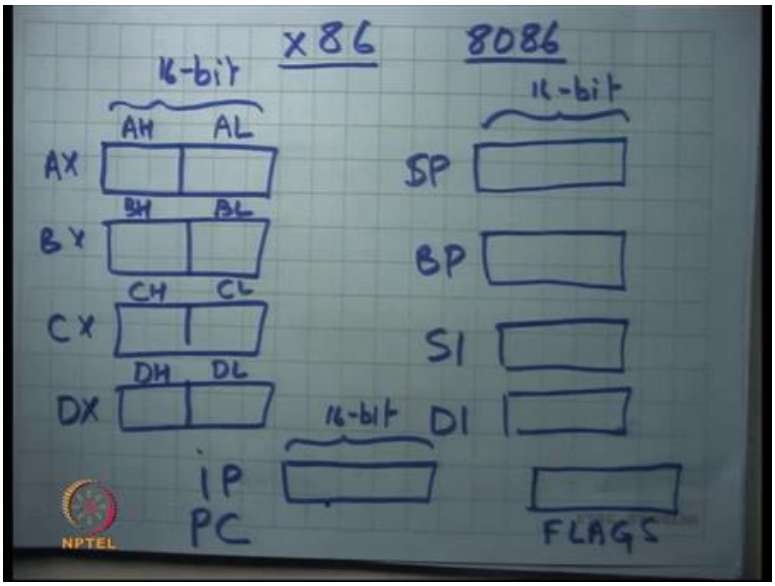
Student: Sir.

Yes.

Student: So, but this logic implemented through the logic gates and finite state machines on as in all like actual code is there for this.

So how are these instructions actually implemented in hardware; are they implemented using logic gates and finite state machines or is some other sort of software sitting inside the hardware that is actually emulating these instructions? Both are possibilities, there are time and space tradeoffs and in doing one or the other and both approaches have been tried. But let us for now assume that these instructions I implemented using logic gates and finite state machines, right alright ok.

(Refer Slide Time: 19:35)



So, now let us concentrate on this on the x86 architecture. So, x86 architecture has a four data registers, they are called AX, BX, CX and DX; these are 16-bit in length in width, right. So, I am actually talking about the 8086 architecture, you know that was perhaps the first generation of these chips that we are using today, ok. So, you know let us say roughly in early 1980's or something.

And so, this architecture had you know, at that time computers were actually operating on 16-bit values and it was felt that operations on 16-bit values is enough for most purposes and so, these this chip was basically an a 16-bit chip. What does it mean for a chip to be a 16-bit chip? It basically means that, the registers width is 16-bits and the instructions that execute are also going to execute on 16-bit values, alright ok.

Also you know they say that most, the many instruction that only need to execute on 8-bit value; so they actually you know divided each resistor into two halves, two 8-bit halves and they also allowed instructions to name them separately. So, each of these 8-bit halves was called be AL, AH, BL, BH, CL, CH and DL, DH etcetera, right. So, an instruction could choose to run on the full 16-bit value or an instruction could actually choose to run on a 8-bit value; and the 8-bit value could be named using one of these identifiers DL, DH and so, you could do that, alright ok.

So, this was you know this was imagined to be primarily for integer computation, like you know whatever you want to do add subtract multiply etcetera. And then you know

you also want a wait; you also want some registers to be able to access memory, right. So, to generate addresses for memory and so, there were four more registers that were present in the x86 in the 8086 architecture. And you know they were called SP, BP, SI and DI, right these were also 16-bit.

So, the idea was that these registers could perhaps primarily be used for providing addresses to memory, right. And, you know the naming was also suggestive SP stands for stack pointer, BP stands for base pointer; I am going to discuss what a base pointer means really and SI and DI stands for I think source index and destination index, in any case these are four resistors. And in fact, you know the way they design their instructions they could operate on any of these instructor registers equally.

Apart from this there was also another register which is called the instruction pointer, alright or let us call it you know it is also commonly known as the program counter. Instruction pointer or program counter same thing, this is also 16-bit, alright. This is also 16-bit, it is a register in the CPU and the semantics are that, when I showed you that loop that it is going to run the next instruction in an infinite loop.

How does that know what is the next instruction? It is just going to dereference, it is just going to send the value of IP; perhaps the value of IP is going to go through the memory management unit, the address is going to go to in the bus, the memory is going to serve the contents of that address and that is the instruction, that contents of that address is basically the instruction that gets executed.

So, in other words the program itself will live in memory. So, this was done for simplicity, they can be other schemes where the program lives somewhere else. So, you know there have been architectures that have separate memory for instruction then separate memory for data. And so, you know they optimize path for instruction, execution and they optimize the path for data execution separately, but the word, but here the same memory is holding both instructions and data, right.

Student: Sir base pointer is same as the frame pointer.

What is the base pointer? Let us just wait, right; now we are going to talk about a base pointer, alright. The other semantics of IP is that, it gets automatically incremented on every instruction execution. So, you execute an instruction, IP gets incremented

appropriately to point to the next instruction. So, that is also part of the logic of the CPU, so that you do not have to have a separate instruction software instruction; I mean that would be that would not be possible so, right. So, I mean to be able to do this for loop, you basically want that the semantics of the instruction pointer that it gets incremented on every fetch of the instruction, alright.

Student: Sir.

Yes question.

Student: Would a program want to access a data than memo address greater than 2 the power 16 that how many your program counter will address them

So, what if have you wanted to access an instruction that lives inside memory, which is greater than 2 to the power 16, right. In the scheme so far, we have discussed so far it does not seem possible right; because the address is only 16-bits and so, you cannot have more than 2 to the power 16 bytes in memory, assuming a byte addressable memory.

So, let us for now just assume that the maximum amount of memory that you can have is 2 to power 16 that is it 2 to, so for gave it. But actually, it is not. So, you know 8086 actually allowed bigger memories and how it allowed, so you know discuss later. But basically, it is the MMU, the memory management unit that allows you to do greater physical memories, alright good.

And also the IP or the IP gets incremented on every instruction executed, on every in execution of an instruction; plus there are special instructions that actually can manipulate the behavior of IP, like the jump instruction right or the jump conditional instruction or the call instruction or an indirect jump instruction or a return instruction.

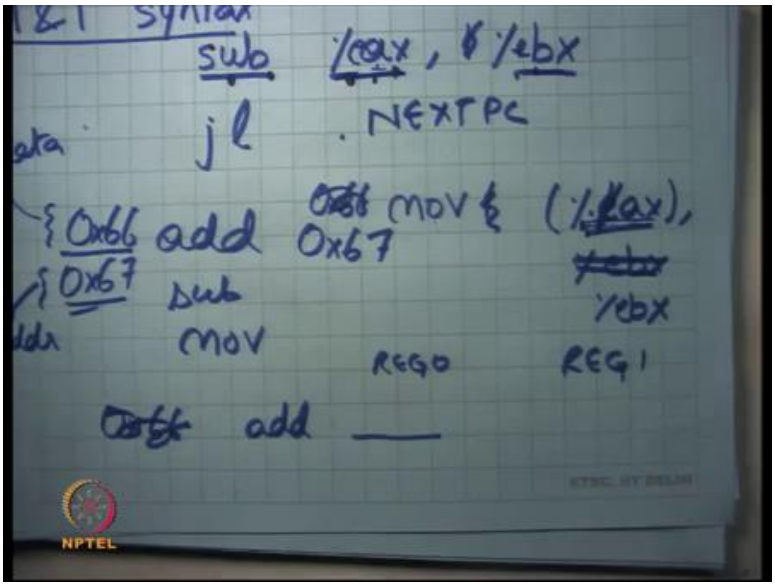
So, these are all you know different flavors of instructions that allow you to be able to manipulate the instruction pointer. There is no instruction on x86 that can actually just allow you to write to IP directly, right. If you want to write to IP you should, if you want to say write value x to IP; the way to do that is to have the instruction jump x ok. So, so IP is different from all the other registers in that sense, alright.

One more thing there is another register inside x86 which is called the flag register, alright. This is also a special register, you cannot just read or write to the flag register

like you can read or write to the other integer registers; but it stores information like whether the last information last arithmetic operation you performed overflowed or not, right.

So, there is some semantics for instructions. So, for example, if you said add AX and BX; if the result overflowed, then you know a flag in the and the flag register, one bit in the flag register will get set, alright. Similarly it can it stores information like whether the last computed result was positive or negative, whether it was 0 or not or whether there was a carry or a borrow in an add or subtract; it also has a system level information like whether interrupts are enabled or not, right. So, we are going to talk about that very soon.

(Refer Slide Time: 27:55)



And so, what happens is, the way you would actually do control flow typically is that; you would let us say in execute some kind of an arithmetic instruction, let us say you executed subtract and you say it subtract AX BX, alright. So, I am using some syntax here, this is the op code; you must have seen this in your computer architecture class. The percentage sign here means that I am talking about a register and whatever follows the percentage sign is the name of the register.

So, I am saying you know, the AX register and the BX register and we are using a syntax called the AT&T syntax for x86, right. There are the two types of its syntax of for x86 what we are going to use in this class and programming assignment is the AT&T syntax. There is another syntax called the Intel syntax, if you read the Intel reference manuals

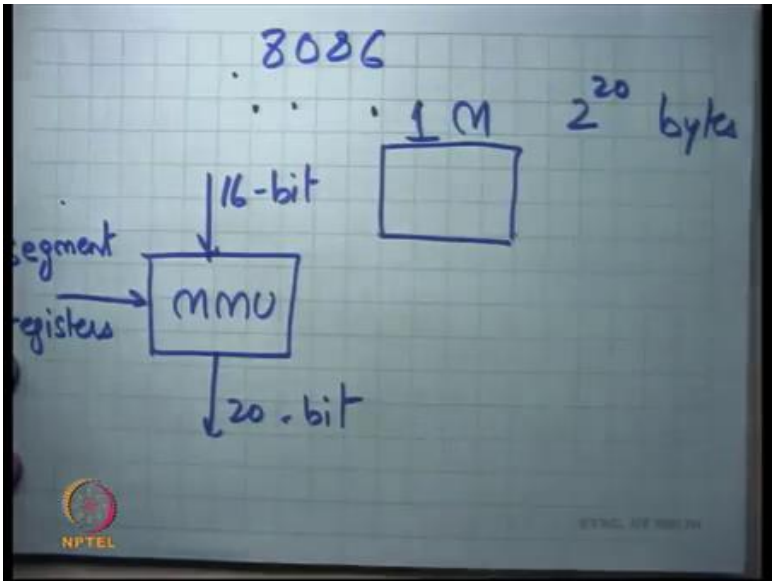
you going to find the Intel syntax; but this is this is you know this is more common and this is what we are going to use in this course, alright.

So, this the syntax says that every register needs to be prefixed with the percentage instruction. And, also says that the first argument is the destination, the first operand is the destination operand and the second operand is the source operand. And so, this instruction means, and they are only two operands you know for an instruction for an op code typically. And, in this basically means that replace the value in AX with AX minus BX right ok.

And, now if I want to say you know subtract AX from subtract BX from AX and then if the result is greater than 0 then jump somewhere, right. So, the way it will happen is that this instruction is going to set a bit in the flags register. And then there is an instruction which says let us say jump less than; which basically says if the neck if the negative bit in the flag resistor is set, so if the last operand was if the last computed value was negative, then you jump to some you know some other next PC, right.

So, these conditional jump instructions are a way to read the flags register and these arithmetic instructions are away to write these flags registers. A CPU a software cannot just directly read or write the flags register, these are the only two ways to read and write in those flag register ok, alright. So, you know there are other instructions like add, subtract, move etcetera that do the usual thing of their name suggests, alright ok

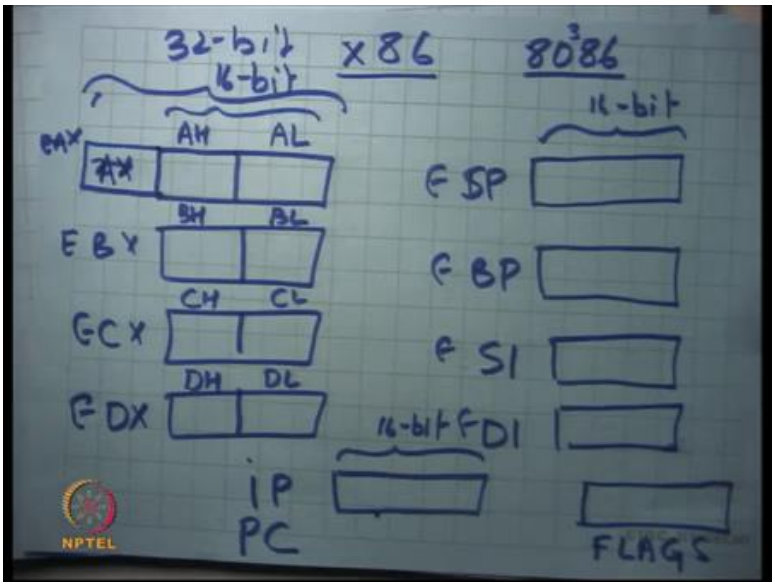
(Refer Slide Time: 30:33)



So, now let us come to the question that you know, do 8086 process, did 8086 processes support only 64 KB memory? So, answer is no, actually they supported up to 1-megabyte memory, right, so 1 M, right; that is the 2 to the power 20 bytes, right. And so, the CPU architects needed a way to be able to address this physical memory using 2 to the power 20 bytes and you know the architecture itself is 16-bit.

So, what they did was, you know the MMU would take in a 16-bit value and generate a 20-bit value with the help of certain register which are called the segment registers. Once again, I am talking about a particular architecture that is 8086, right ok. So, how does this work?

(Refer Slide Time: 31:29)



Apart from all the registers that we have talked about there are also, so let us say you know there is A B C D these are the integer registers, there is IP and there are flags and you know there are also those ESP etcetera, you know want to write them. And then there are these segment registers that, there are six segment registers in on x86 CS code segment, DS data segment, SS stack segment and then there are and then there was another segment which is called the ES, this is you know for anything else you want to do it is typically used for strings.

So, it is called you know, it is usually they are just used for the string operations and we are going to talk about that implement. These registers themselves were 16-bits, right and what MMU would do is; so, any at if you want to dereference memory, you cannot just

say that here is a register and I want to dereference the address at SP. You have to actually address the memory using a pair of the segment register and an address; and the address could be a direct address or a register address or whatever else, right.

So, I would either say you know %CS: I %IP, this is the full address that I want to access. And the semantics of this is that, the value of CS, so when I put something in brackets it basically means; you dereference that, right. So, actually do not dereference it here. So, you the value of CS that is %CS into 16 plus the value of IP; so, that will be your total address, that will be your complete address that will actually go on the bus to the memory, right.

So, in other words your MMU on 8086 was this very simple MMU the memory management unit, which would take an address named by a pair of segment registers and some integer which could be used; which could be the resistor, one of the resistors integer resistors and it will do this computation and send the address on the physical bus right.

Student: Sir, 16 or 2 to the power 16.

16.

Student: Only 16.

Yeah. So, basically it only allowed up to 1 megabyte of memory that is 2 to the power 20 bits. So, 16 is 2 to the power 4. So, you can imagine you can see that, this totally this can address any byte in a space of 2 to the power 20, it 2 to the power 16 into 2 to the power 4 is 2 to the power 20, so you can address any of the 2 to power 20 bytes ok, alright. And plus, you know there was some default things.

So, for example, any dereference through the IP will always go through the CS, right. So, you know for example, when I say run next instruction, the run next instruction is only not only dereferenced in the IP, it is dereferencing CS: IP, right. So, whatever you have set up in CS that is going to you be used in your MMU. So, another the intention must have been that, you know the code will live in a separate segment and you are going to or separate region of memory. It is all uniform memory; but you are basically dividing this memory artificially, using these segment resistors.

And you will basically say that CS holds the base of the area where you going to store your program, DS holds the base of the area which holds your data, SS holds the base of the program which holds your stack, right. And then also they were default things like, whenever you dereference an instruction it is going to go through CS; whenever you dereference the stack using the SP or BP pointers, then you are going to go through the SS segment. And everything else by default is going to go through DS right and then there was some special instructions that will go through ES.

Student: Sir certain, but you think CS has 16-bit resistors is not it page full of memory, because we can just you used as the 4-bit register and still perform the same work.

Could we have used CS as a 4-bit registers and perform the same operation; well a 16-bit register allows you to you know start anywhere, right. So, the code segment could start on the top of the one megabyte region; but if you have a 4-bit register, then you know you have you lose the you lose some freedom in where the base will start, ok. So, and you know and from a hardware designer's point of view all my logic is anyway 16-bit, you know having such special 4-bit where all logic is actually more costly than less costly. So, I am just guessing, yeah.

Student: Is there any specific reason of keeping memory 1 M.

Is there any specific reason, no; I mean at that time they just thought you know 1 M is more than enough. So, let us just. So, every memory will have a limit, even today's memory you cannot, I mean the CPU will only support of memory up to this size, right. And so, whenever you know in the real world you will always have limits, you cannot just have infinite memory and at that time 1 M was thought to be enough and of course, we know now that is not enough.

And so, the architecture has evolved, and I am really going to talk about how it is evolved, it is basically you know it is evolved in a backward compatible way, alright. So, I mean this was a little you know I mean not very clean actually, because there is no protection and I can you know, I can reference the same byte in physical memory using many different addresses, right. I could set up CS in some way and IP in some way or CS something CS lower IP higher, because the same byte could have multiple addresses in this scheme.

Because as you pointed out there is redundancy and in the number of bits are being used to address a certain address space right. 32-bits have been used to address subspace of 2 to the power 20, so there is clearly redundancy and the way things are done, alright. So, it is a little complex, it is it was also a little difficult for to program this kind of a thing; because you know we are used to writing programs where we talk about variables and now it is a job of a compiler to actually translate those variables into memory addresses.

And now the compiler has to worry about where this memory address is going to be allocated, whether it is DS or SS or CS. And also, it has to worry about oh there should not be any overlap, so you know CS plus some address should never be able to reach this variable you know. So, all these things make life very difficult for the software writer or in particular the compiler writer if you are using a compiler, alright.

So, clearly 16-bits was painfully small, quickly this was realized you know in a few years time that you know this is not going to work we need to extend this; but you know what they also needed to do was provide backward compatibility. So, there is some software that is been written for the 16-bit architecture, you know there has been OS, there are some OS is that have been written for the 16-bit architecture.

And they have you know they have invested in the company, they have bought the processor, they written software for that come for that processor and now if the processor gets changed then you know they will cry. So, what they do is they provide backward compatibility. And the backward compatibility means that; when the process starts, the processors boots when you are power to on a processor, it actually always boots starts in the 16-bit mode.

So, if you know OS for written for 16-bit, it will just you know happily run on the 16-bit mode. And then you provide certain special instructions, which were actually not present in the 16-bit processor to switch to 32-bit mode, right. So, if an OS is actually a 32-bit O S, it will start in the 16-bit mode and very soon it will call an instruction to switch to 32-bit mode and now it can run in 32-bit mode. However, if there was an OS which was which does not understand 32-bit only understand some 16-bit, it can still happily run on the old processor, right.

And actually this has continued and to this day you know when we are when we have our chips which are 64 bit, we still I mean boots in the 16-bit and it is a job of the OS writer

to you know, the OS writer is aware that it will always boot in the 16-bit. And now it will execute some code in the 16-bit mode, call us a special instruction to switch to 32-bit mode; then it will execute some code in the 32-bit mode, then we will call a special instruction to switch to the 64 bit mode and that is you know finally, I can run my applications, right.

So, the architecture developed from 16-bit to 32-bit to 64 bit and to provide backward compatibility it will it still boots in 16-bit and then third and the OS has to switch it from 16 to 32 to 64 in that way, right. So, that older OS is can still run. So, for example, the 32-bit OS can still run on a 64-bit processor and the reason is basically because it is still you know boots in the 16-bit mode and then 32-bit, alright ok. So, how did the 32-bit architecture change?

So, the 32-bit series was let us say you know 80 something 386, 186, 286 etcetera and they basically made all these registers 32-bit. So, you know this was extended by another 16-bits, this becomes 32-bit and the new name of this register is EAX, extended AX, right.

Similarly, this becomes EBX, ECX, EDX, ESP, EBP, ESI, EDI, alright. So, all these registers get extended by an extra 16-bits and they are renamed to ESP, EBP etcetera. They still allow you to access the lower 16-bits of a register by using the old names. So, you can still say AX and that will mean the lower 16-bits of EAX, right or you can say SP and that will mean the lower 16-bits of ESP, right ok.

And now they changed all these instructions to you know with the same op codes to mean the same thing except that they will now operate on 32-bit values, alright. So, subtract basically means subtract EAX, EBX alright; add, move whatever right, they all basically become 32-bit, they become 32-bit versions on themselves, right.

So, the same op code, the same encoding of an instruction becomes a 32-bit version of itself. If for some reason you want to still access a 16-bit region, a 16-bit value inside the register; you can prefix an instruction with our special prefix. Let us say the prefix is you know some bytes let us say 0x66, it basically says that treat this instruction as a 16-bit instruction as opposed to a 32-bit instruction.

So, these are just some things that the designers did. So, to be able to maintain both worlds, you can you know by default everything becomes 32-bit; but if you want to use 16-bit you can prefix an instruction with this special prefix and that instruction now behaves as though it is a 16-bit instruction, alright. And similar things are actually true about 32-bits and 64 bits ok.

Student: Sir.

Yes

Student: Do not we prefix with this 16-bit, the prefix to make a 16-bit then the opponents have to be the older ones that is the AX, BX not EAX, EBX.

Right. So, for example, if I say 0x66 add and let us say you know I specified no. So, there is some number encoding for each register right; for example, AX the encoding for AX is 0 and the encoding for BX is let us say or say let us say 1 hypothetically, right. So, it will say you know add register 0 to resistor 1, right. And if I do not use the prefix, this encoding means add EAX to EBX; if you use the prefix it basically means add AX to BX, right that is that only difference.

And they are actually two prefixes there is 66 and 067 which stand for you know, this basically says I want to change the size of the data. So, data is basically the value that you are operating on for example, AX or EAX and 67 basically means I want to change the value of the address, right.

So, for example, if I wanted to dereference memory, I could actually do something like move percentage EAX to percentage EBX. Oh! did I say that the first operand to the destination sorry, so the second operand is the destination, alright. So, in this in that in the syntax of AT&T the second operand or the last operand is the destination correction, alright.

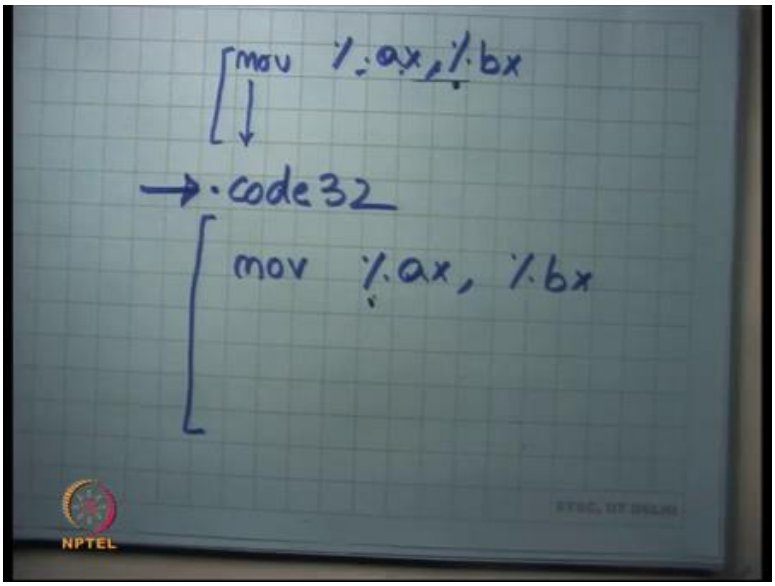
So, when I say move EAX to EBX, it is basically going to read the value in EX, dereference it; which means it is going to you know send that address to the MMU, generate a physical address, ask them memory what to fetch it is contents and the contents are going to be stored in this register called EBX. So, that is what is going to happen, right.

If I prefix it with 66; what is going to happen is, the address remains 32-bit, but the data becomes 16-bit, right. So, I still use 32-bits to dereference memory to address memory through my MMU; but the number of bytes I fetch from that address is only 2 or 16-bits, right. So, that is what the prefix 66 means, it is basically referring to the size of the data it is.

So, if you see you if you prefix it with 0x66 let us say, then you are going to you basically changing toggling the size of the data on which this instruction is operating. On the other hand, if I use 67, I am toggling the size of the address, right. In this case the address is EAX. So, now, this instruction is going to become move EAX and if I do not have 66, then it is going to say move AX to EBX. So, it is basically saying dereference using AX, but the data still remains 32-bit right and if I use both then both becomes 16-bit, right.

So, they are two things does in every instruction will have an address component to it and a data component to it if it accesses memory. And so, these two prefixes can toggle either or both, alright yeah. So, lot of details, but good to understand good to know once and then you know be comfortable order it forever, alright.

(Refer Slide Time: 46:39)



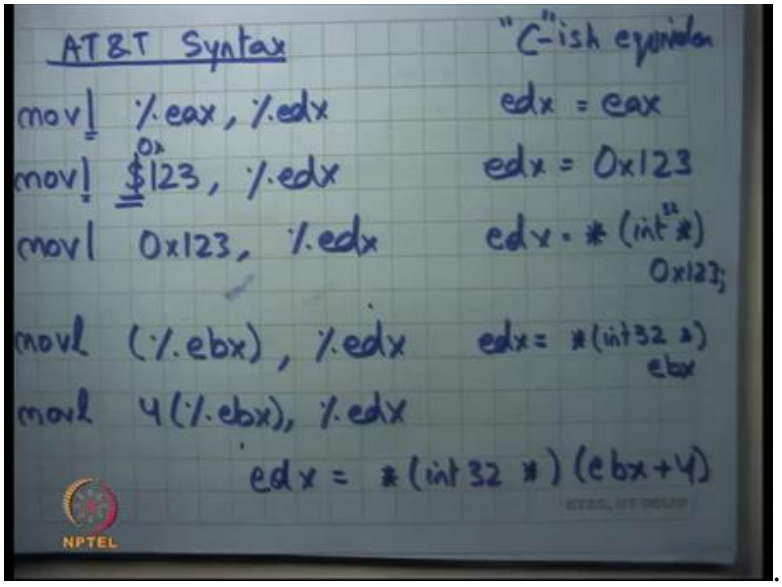
Let us also understand you know how an assembler works. So, for example, when you write code, when an OS is going to write code is going to write some assembly instruction. So, it is going to say move something, something and so on. So, you can tell

the assembler that look this part of the code is, compile this code as 16-bit code and then you know you can tell the assembler.

So, there is a directive called let us say code 32 and then you can tell that all future code should be compiled as 32-bit code, right. So, for example, if there is an instruction called move AX BX here and there is another instruction called move AX BX here; here I could just use the encoding of this instruction without having to prefix it with anything, right. But here if I want to use the same instruction, I need to prefix it with 0x66 let us say right, because it is a 32-bit mode.

So, this directive is a way to tell the assembler that in what mode should you compile this code because, the same string can mean different encodings in different mode ok, alright. So, will we look at 32-bit architectures for our programming assignments and most of our discussion; but you know seems already seems very complex I do not want to discuss 32-bit or 64 and I do not think you want it discuss it either, right. But you can imagine it is a similar kind of complexity that is going to occur and in doing that also, alright ok.

(Refer Slide Time: 48:07)



Let us look at some instructions, some real instructions. So, here is the AT&T syntax. Let us say move l percentage EAX to percentage EDX. Let me write the C-it equivalent or you know semantics of this instruction in a language we understand C, right. So, let us say.

So, what this basically means is, EDX is equal to EAX right just to understand what this means; if you do not understand the syntax let me just explain it to you and see. I could say move $123 to EDX; $123 means it is a constant; it is an immediate value, right. It is an immediate operand; you must have seen immediate operands in your computer architecture class, and this basically means EDX is equal to 0x123 let us just say it was you know it was a hexadecimal number 123.

Notice the use of the dollar sign to specifically say that this is a constant that I want to move into EDX. On the other hand, if I say move, so, I am also prefixing it with a character which says what is the size of the data I am using. So, by default on a 32-bit mode it is 32-bit, but you can also specify it using l. So, l stands for 32-bit. So, you want you will basically be moving 32-bit values, alright.

You can also say move l's 0x123 to percentage EDX. Notice the difference between this and the previous one is that, I am not using the dollar sign, alright. And what does this mean? Yeah so, treat 123 as an address for the memory, right. So, it is basically saying EDX is equal to dereference; first typecast 123 as a pointer, that points to a 32-bit value because the 32-bit instruction and then dereference that pointer and whatever result you get put it in EDX right; just dereferencing EDX and dereferencing 123 ok.

If I say mov l, so this is immediate addressing, I could also say move l EBX to EDX; this is indirect addressing basically means EDX is equal to star you know it is int32 just to specify that it is a 32-bit number (int32 *)EBX, alright. So, basically saying look at the value in EBX, use it an address to memory, dereference it in memory and fetch the contents and store it in EBX, EDX ok. And finally, I could you also say something like movl 4(%EBX) to %EDX.

So, x86 allows you these kinds of instructions where you can specify an offset with a resistor. So, basically means that you add 4 to EDX and then you dereference it and then you get the contents and put it in EDX. So, basically means EDX=*(int 32 *) (EBX + 4). What is the difference between the prefix 0x66, 0x67 and this suffix l in my assembly code, this is assembly code, alright?

So, I can use you know, I am specifying assembly code using strings where I specify an op code using a string and I use a as a suffix l to indicate whether the 32-bit over 64-bit. The 0x66, 0x67 is for the instruction encoding in binary, right. So, the assembler is going

to convert this string into prefixed or un-prefixed versions of instruction encodings, right ok.

So, I will stop here.