

**Design and Engineering of Computer Systems**  
**Professor Mythili Vutukuru**  
**Computer Science and Engineering**  
**Indian Institute of Technology, Bombay**  
**Lecture -3**

**Overview of CPU Hardware**

Hello everyone, welcome to the third lecture in this course Design and Engineering of Computer Systems. So, in this lecture, we are going to begin with understanding the underlying hardware over which computer systems run. So, in this lecture we will begin with the CPU hardware. So, what is a CPU? The main function of a CPU is to run user programs.

(Refer Slide Time: 00:42)

**CPU hardware**

- User program = code (instructions for CPU) + data
- Stored program concept
  - User programs stored in main memory (RAM)
  - CPU fetches code/data from RAM and executes instructions
- CPU runs processes = running programs
- Modern CPUs have multiple CPU cores for parallel execution
  - Each CPU core runs one process at a time each
  - Modern CPUs have hyperthreading, where each core can run more than one process also

So, what is a program a program is nothing but the code that you have written along with the data on which the code operates. So, modern computers use this stored program concept. What is the stored program concept? You will store your program in RAM also called main memory. So, your RAM will contain the code plus the data the variables data structures everything in your program all of this will be stored in ram. And then your CPU which is the piece of hardware i.e. the Central Processing Unit will fetch each instruction one by one and will fetch the pieces of data and operate on them and will run your code on your data.

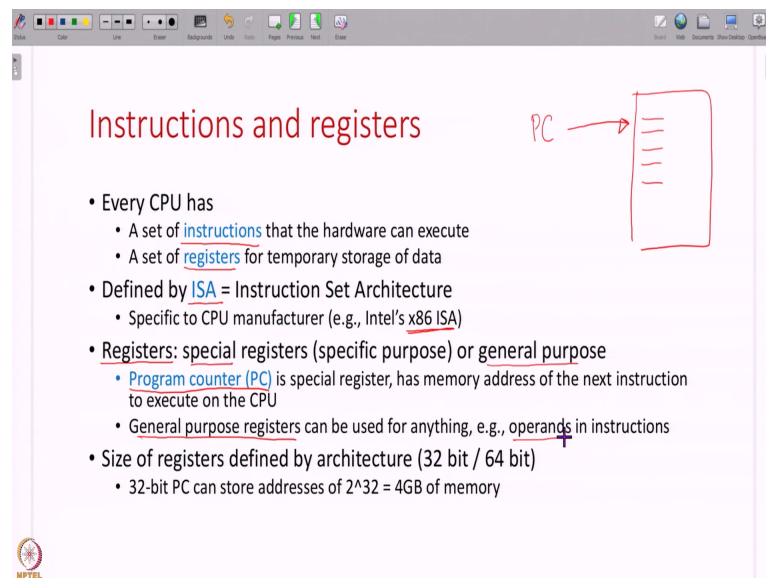
So, the execution unit is the CPU and the storage unit is the RAM, so this is how modern computer systems are built. At the very basic level you have CPU executing code on data both of

which are residing in the RAM. So, we will introduce this term called a process, what is a process, a process is a running program. When you run a program, a living program is called a process. So, the main job of the CPU is to run processes. And how many processes does a CPU for example your laptop or desktop how many processes does it run.

So, it turns out that modern CPUs have multiple CPU cores. If you look at the CPU in your computer it has multiple execution units or CPU cores, even things like your phones have multiple CPU cores these days. And each CPU core is capable of running one process at a time. Of course, you can run different processes at different points of time but at any point of time one CPU core is running one process.

So, this is the assumption we will make for simplicity in this course. Of course, modern CPUs have a property called hyper threading, where each CPU core can simultaneously run more than one process which is another optimization but we will ignore that for the purpose of this course. So, for the purpose of this course, the CPU model we will use is a CPU has multiple CPU cores each of which is capable of running one process at a time. So, now in this lecture let us go into more details about the CPU hardware and understand how it actually runs these processes.

(Refer Slide Time: 03:22)



### Instructions and registers

- Every CPU has
  - A set of instructions that the hardware can execute
  - A set of registers for temporary storage of data
- Defined by ISA = Instruction Set Architecture
  - Specific to CPU manufacturer (e.g., Intel's x86 ISA)
- Registers: special registers (specific purpose) or general purpose
  - Program counter (PC) is special register, has memory address of the next instruction to execute on the CPU
  - General purpose registers can be used for anything, e.g., operands in instructions
- Size of registers defined by architecture (32 bit / 64 bit)
  - 32-bit PC can store addresses of  $2^{32} = 4\text{GB}$  of memory

So, every CPU is defined by two things what are these? These are the set of instructions that the CPU hardware can execute as well as the set of registers present within the CPU. So, the

registers are nothing but just temporary memory for temporary storage of data while the instruction is running that is the concept of a register. So, both these things what instructions can a CPU hardware run and what registers does it use for temporary storage while running these instructions these two things are called the Instruction Set Architecture or the ISA of a CPU.

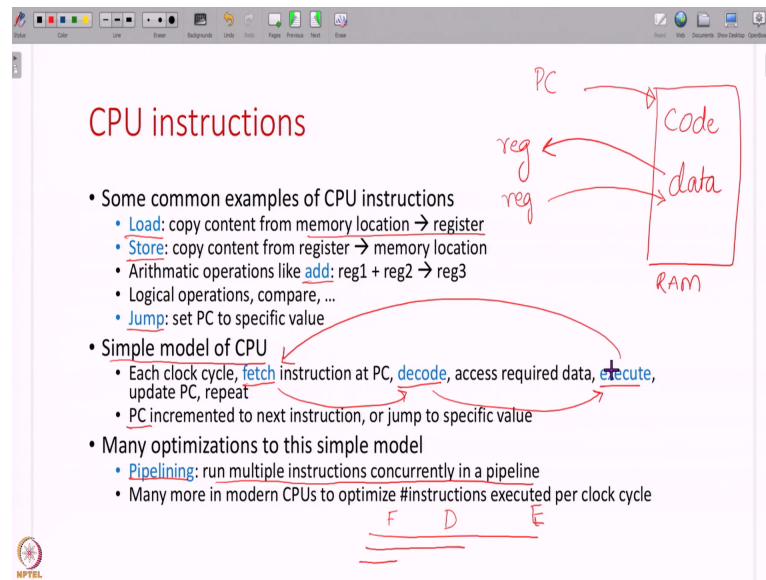
So, every CPU manufacturer has its own ISA defined, for example, intel has the x86 instruction set architecture which defines the set of instructions that intel CPUs run and the registers that they use. So, in this course we are not going to go into details about any one particular ISA. But in this lecture, I will just give you a very high-level overview of what are instructions and what are registers. So, coming back to what are registers as I said just before. So, registers are memory in the CPU present for temporary storage of data.

So, there are two types of registers. Some registers are called special registers they are there for a specific purpose in the CPU they store specific types of data and the others are called general purpose registers. So, I will give you an example. A very popular commonly discussed special register is the Program Counter or the PC. So, what does a Program Counter store? It stores the memory address of the next instruction that the CPU has to execute. So in your RAM you have code, you have many instructions and the program counter stores the address of the next instruction the location of the next instruction that the CPU has to execute.

So, this is a special purpose register which stores a specific piece of information. There are many other special registers like this in CPUs and a lot of other registers are just general-purpose registers that can be used to store anything, for example, if you are adding two numbers then the numbers being added might be stored in the general-purpose registers like operands in instructions. So, these are the various types of registers that CPUs have.

So, what is the size of these registers? It is defined by the architecture, for example, you must have heard some CPUs are 32 bit some are 64 bit. So if you have a 32 bit CPU then the register size is typically 32 bits so your program counter will be 32 bits in size and in this 32 bits it will store a 32 bit address of the next instruction to run. So that is about registers.

(Refer Slide Time: 06:18)



Next, let us just briefly understand what are CPU instructions. So, again every CPU has a large set of instructions defined and, in this course, we would not go into a lot of details of the specific instructions, but I will just give you a overview of the common flavors of instructions that are available.

So, what are the kinds of instructions that modern CPUs have? One common instruction is what is called the Load Instruction. You have RAM and you have the code and data of your program stored in RAM and the CPU has to periodically copy some content from a memory location into a CPU register. So, it will load some variables and some data from memory into the CPU register. So an instruction is there for that that is called the load instruction.

Similarly, a Store instruction what does it do from a CPU register you will store the data back into RAM or main memory that is the Store instruction. And you also have various instructions for arithmetic operations, for example add once you have loaded two variables into two registers you have an instruction to add those two registers and store it into a third register that is the add instruction you might have subtract various other logical operations comparing two registers.

So, there are many arithmetic and logical operations that a CPU can perform and you also have some other kinds of instructions like jump. So, your program counter is pointing to the next instruction to run so normally you will sequentially run the instructions one by one but



sometimes you might want to jump to some other part of your code, for say if there is any false condition or something you want to jump. So, there are instructions to jump to set the pc at certain specific values to make your program counter move to different parts of the code as well.

So, these are just some common examples of CPU instructions that are found in modern CPUs. So, in this course we will just describe a simple model of how a CPU works. every clock cycle at the hardware work gets done at the granularity of clock cycles. So, in every cycle what a CPU will do is it will fetch the next instruction that the PC is pointing to it will decode the instruction and it will execute the instruction, so to execute the instruction you might have to fetch more data from memory you might have to load some data into registers you do all of that you execute the instruction then you update the program counter and you repeat this process again fetch, decode, execute and repeat.

So, every clock cycle this is what the CPU does. And this program counter itself you either simply keep going to the next instruction, next instruction or you jump to specific values using say a jump instruction, so that is how the program counter itself is updated. So, this is a very simple model of CPU but of course modern hardware has many more bells and whistles you know enhancements added to the simple model. A very common enhancement is what is called pipelining.

That is suppose you know there is the fetching process there is the decode there is the execute and when one instruction is in the execute phase while waiting for it to complete you might get the next instruction fetch it start decoding it while this instruction is decoding you might want to start fetching the even further instruction.

So, modern CPUs run multiple instructions like this concurrently in a pipeline. This is to increase the efficiency of your hardware. And there are also many more such optimizations that modern CPUs do they do not do the simple fetch decode execute cycle that I have described. But the simple model is enough for us for the purpose of this course. And if you want to understand CPUs in more detail then you are encouraged to take a course on computer organization or architecture. So, this is about CPU instructions. Moving on now that we know what are registers what are instructions let us understand what happens when you actually run a program.

(Refer Slide Time: 10:40)

The slide is titled "Running a program" in red. To the right of the title is a hand-drawn diagram. It shows a box labeled "CPU" containing "PC" and "reg". An arrow points from "reg" to a box labeled "process" which contains "code" and "data". Below the "process" box is the label "Ram". To the right of the "process" box is a cylinder labeled "a.out". An arrow points from "a.out" to the "process" box. Another arrow points from the "process" box back to the "CPU" box. Below the diagram is a list of bullet points.

### Running a program

- What happens when you run a C program?
  - C code translated into **executable** = instructions that the CPU can understand
  - Translation done by program called **compiler**
  - Executable file stored on hard disk (say, "a.out")
  - When executable is run, a new **process** is created in RAM
  - **Memory image** of process in RAM contains code+data (and other things)
  - CPU starts executing the instructions of the program
- When CPU core is running a process, CPU registers contain the execution **context** of the process
  - PC points to instruction in the program, general purpose registers store data in the program, and so on

So, all of you must have written a program at some point of time. Suppose you have written say a C program. How does this program run on the CPU? First thing that happens is this C program you will compile it. There is a piece of software called compiler that will translate your C program into an executable. So, this executable is nothing but a sequence of instructions that the CPU can understand. So CPU cannot understand c language so the compiler will translate your C code into a language that the CPU can understand.

So, you first create an executable and say this executable is stored in you know for example hard disk as a file you have this a.out file that is stored in the hard disk. Now, when you want to run this program what happens all the code and data in your program has to be copied into RAM into main memory. You will copy the code and data in the executable and a new process is created. Every time you run a program, a running program is called a process, a new process is created in RAM.

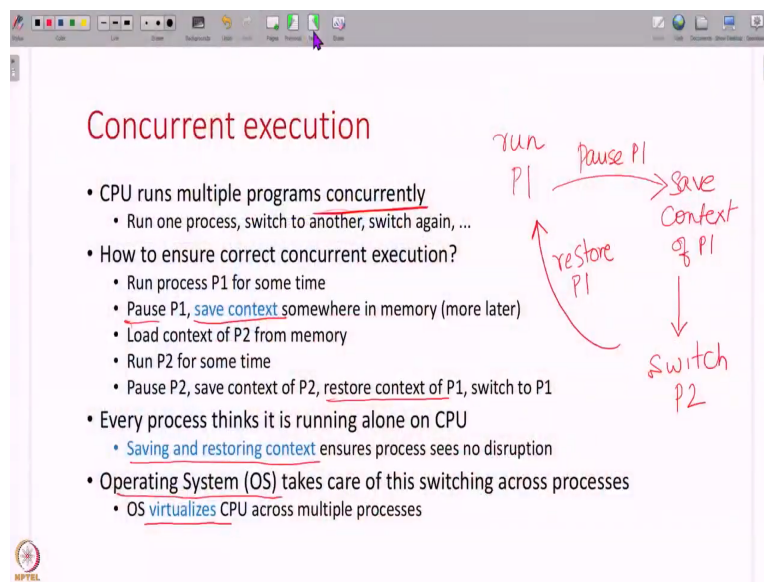
So, memory for this process is allocated in main memory, the code and data everything is stored in ram. So this code and data of the process that is stored in RAM is called the memory image of a process. You create a process you create its memory image. And then now that the code and data everything is available in the memory image of the process in RAM, the CPU can start executing the process. The program counter say points to some instruction to run next the various

registers, load data from the RAM, store data into RAM and the CPU starts executing the process.

So, when a CPU core is running a process the various CPU registers they have information about the process. The registers general purpose registers may have data fetched for arithmetic operations the program counter has the address of the code. So, the various CPU registers contain information about the process and that is called the context of a process the execution context of a process.

So, the execution context of a process is nothing but the value the various CPU registers that are stored pertaining to the process information about the process that is stored in the various CPU registers is called the execution context of a process. So, in this way the process has a memory image in RAM and its execution context is there in the various CPU registers.

(Refer Slide Time: 13:31)



So, now the next question comes up you might be wondering does a CPU core run just one process and wait for the process to finish and then start another process or does it simultaneously run multiple processes at the same time. So, it turns out that modern CPUs are capable of running multiple processes concurrently that is at the same time there could be multiple active processes and the way CPUs work is they will run one process for some time stop it switch to

another process, stop it switch to another process in this way concurrently multiple processes will keep running.

So, that you as the user you do not really realize that a process is being stopped and restarted and all of that this is managed efficiently at the hardware level. So, how do we do concurrent execution? So, if you run a process for some time and stop it, it was in the middle of some calculation it loaded some values into registers and you suddenly stop it then what happens you cannot just leave a process midway like that. So, for correct concurrent execution what we need is whenever we pause a process we should be able to save its context that is save all the values in the CPU registers that the process has somewhere in memory so where in memory will we store this will come to this a little bit later.

But for now, I want you to understand that when you pause a process you have to save its context and when you restart a process once again you know you pause process P1 you jump to some other process you run it for some time and then when you want to come back to process P1 again you will have to restore the context of P1.

So, you run a process for some time so this is a process P1 you are running it for some time then after some time you will say you want to pause P1 then you will save the context of P1 somewhere all the values of the CPU registers which instruction did it stop at what are the various values in the general purpose registers all of this you will store somewhere then you will switch to P2.

And then after running P2 for some time you will restore the context of P1 you will copy back all those values into CPU registers once again and then you will go back to running P1 again. In this way processes are run concurrently on the CPU, so the saving and restoring context ensures that a process by itself sees no disruption or sees no weird behavior when the program is run. And who does all of this saving context restoring context? All of this is taken care of by the operating system. So this operating system takes care of switching across multiple processes seamlessly so that you as the user do not realize this is happening.

So, we will study how the operating system does this in a lot of detail later on. So the common term used for this is the OS virtualizes the CPU across multiple processes. It gives multiple

processes the illusion that they are exclusively running on the CPU even though in the background the operating system is pausing a process saving its context restoring its context later on. All of this is being done in the background by the operating system.

(Refer Slide Time: 16:54)

**Interrupt handling**

- In addition to running user programs, CPU also has to handle external events (e.g., mouse click, keyboard input)
- Interrupt = external signal from I/O device asking for CPU's attention
- How are interrupts handled?
  - CPU is running process P1 and interrupt arrives
  - CPU saves context of P1, runs code to handle interrupt (e.g., read keyboard character)
  - Restore context of P1, resume P1
- Interrupt handling code is part of OS
  - CPU runs interrupt handler of OS and returns back to user code

So, moving on what are some of the other things that the CPU does. The other thing in addition to running user programs the CPU also has to handle external events. It is not just user program that is running, there are other pieces of hardware connected to the computer like a mouse or a keyboard or a hard disk or a network card and all of these devices might sometime need some attention of the CPU.

For example, if you click something on the mouse the CPU has to handle that event. There are all of these external events that keep happening and these are called interrupts and the CPU also has to give some attention give some time to handling these external events or interrupts and these are typically from I/O devices.

So, how are interrupts handled? Suppose the CPU is running a process P1 and an interrupt occurs and the CPU has to do some work to handle the interrupt. Again we use the same principle we have to save the context of P1. And then handle the interrupt whatever work is needed to handle

the interrupt for example if it is a keyboard interrupt you might have to read what the character is from the keyboard into the computer.

So, you will save the context of P1 you will pause this process save its context handle the interrupt and then restore the context again and go back to process P1. So, the same logic of saving and restoring context is used even when handling interrupts. So again all of this interrupt handling is also part of the operating system code. So, for example, you as a user program you may not know how to handle a keyboard interrupt or how to read characters from a keyboard into your program all of this you may not know.

So, this is done by the operating system what the CPU does is whenever there is an interrupt it will save the context of a process and hand over control to the operating system which has code for handling various interrupts. And then once this interrupt is handled the CPU will go back to running the user program. So, this is how interrupts are handled we will of course study this in more detail once we study operating systems in this course.

(Refer Slide Time: 19:13)

**Isolation**

Handwritten notes:  $P_1$ ,  $P_2$ , ...,  $P_n$  (boxed), HW

- How to protect processes from one another?
  - Can one process mess up the memory or files of another process?
- Modern CPUs have mechanisms for isolation
- Privileged and unprivileged instructions
  - Privileged instruction = access to sensitive information (e.g., hardware)
  - Regular instructions (e.g., add) are unprivileged
- CPU has multiple modes of operation (Intel x86 CPUs run in 4 rings)
  - Low privilege level (e.g., ring 3) only allows unprivileged instructions
  - High privilege level (e.g., ring 0) allows privileged instructions also
- User code has unprivileged instructions, runs at low privilege level
  - CPU does not execute privileged instructions when in unprivileged user mode
- OS code has privileged instructions, runs at high privilege level
- When user program wants to do privileged operations, it must ask OS
  - CPU shifts to high privilege level, runs OS code, returns to low privilege, back to user code

So, the next important concept that modern CPUs have is that of Isolation. So, you might be wondering if there are so many programs running at the same time on a computer is it safe, can it so happen that one program or one process can mess up with some other process, can it say overwrite the memory of another process, can it access the files delete the files of another

process do something bad can all of this happen. So, that is a very valid question if you have that in your mind and it turns out that modern CPUs do not allow such things. Modern CPUs provide mechanisms for Isolation.

That is even if you have multiple processes P1, P2, Pn all of them running on the same underlying hardware these are all isolated from each other. one process cannot in any way disturb the execution of another process. So, how is this isolation done? It is done by the following mechanism. So CPU instructions are of two types. They are some privileged instructions and unprivileged instructions.

So, what is a privileged instruction is? It is an instruction that accesses any sensitive information for example if you want to access the hardware access, memory access, files stored on the hard disk all of these instructions that do these operations are all sensitive instructions. And regular instructions that add values into registers for example they are somewhat harmless they are all unprivileged instructions. So, modern CPUs what they do is they have multiple modes of operation you know different states in which they run these are also called rings in x86 CPUs.

There will be a low privileged ring which is the ring 3 in intel CPUs which only allows unprivileged instructions to run. Similarly, there will be a high privilege ring like ring 0 in which you can run both privileged as well as unprivileged instructions. So, you have a low privilege level where you have very little access and you have a high privilege level where you have lots of access with privileged instructions access to hardware and so on.

So, what CPUs do is they will allow user code to only run at a low privilege level and only use unprivileged instructions the programs that you write you cannot execute privileged instructions and if you somehow insert any privileged instruction into your user programs the CPU will not allow it to run. So, when you start your user program the CPU goes into a low privilege level and only unprivileged instructions are allowed. On the other hand, special trusted software like the operating system code will have privileged instructions and will be allowed to run at a high privilege level.

So, therefore when the operating system code is running the CPU will be in a different mode and it will allow the execution of privileged instructions. So, if you as a user program you want to

access some hardware you want to do some privileged operation then you have to go through the operating system. You cannot do it on your own in your user program. You cannot write code that will go access the memory of another program delete the files of another program all of that you cannot do directly you have to ask the operating system.

And when you ask the operating system then the CPU will shift to a high privilege level it will execute the corresponding operating system code and come back to your code in a low privilege level. And when you ask the operating system the operating system will of course check are you only modifying your files and your memory then it will allow it are you trying to do something bad by accessing somebody else's memory then the operating system will not satisfy your request.

So, in this way the isolation property is maintained mainly because modern CPUs have these two different modes of operation a high privilege level and a low privilege level and you will somehow sandbox isolate user processes in the slow privilege level so that they cannot do any privileged operations.

(Refer Slide Time: 23:28)

**CPU caches**

CPU

Cache

code + data

RAM

- CPU must access memory to fetch instructions, load data into registers
  - But main memory (DRAM) is very slow (100s of CPU cycles)
  - CPU cannot do useful work while waiting for memory
- To avoid many memory accesses, CPU stores recently accessed instructions and data in CPU caches
  - Multi-level cache hierarchy, some private to cores, some common
  - Example: private L1, L2, common last level cache (LLC or L3)
  - Can be separate for instructions and data, or common (e.g., L1 is separate)
- Caches have low access latency (tens of CPU cycles), faster than DRAM but smaller in size, more expensive
  - Can only store most recently used instructions and data

C0

C1

L1

L2

L3

So, moving on the next concept that we are going to study about CPU hardware is the notion of CPU caches. So, we have seen that code and data are stored in RAM and the CPU has to fetch these instructions from all the way from wrap in order to execute them it has to fetch the data to



load them into registers and so on. But main memory or D-RAM is actually very slow compared to CPU. So CPU can execute one or more instructions in every CPU cycle but memory access takes like hundreds of CPU cycles.

So, therefore, the CPU cannot do any useful work while it is waiting for the memory access the CPU wants to add two numbers but if it takes a long time to get those numbers from ram then the CPU time is basically wasted. So, to avoid this what modern CPUs do is? The most recently accessed code plus data are stored in some special memory closer to the CPU called the CPU caches. So, this is a different kind of memory from main memory or D-RAM and this memory is faster it can be accessed much more quickly by the CPU and it is located close to the CPU so that recently used data are stored here. So, that CPU can quickly access them later on in the future.

So, modern CPUs have multiple levels of caches. They just do not have one cache they have multiple levels of caches, some which are you know private to each core and some which are shared. For example, if you have say core 0 and core 1 then these cores might have level 1 level 2 which are called L1 L2 caches and then together in common they might have a level 3 cache which is also called the last level cache.

So, this is the common design that is used in modern CPUs private L1 cache L2 cache and a common last level cache. Here in this common cache the code and data accessed by both cores can be stored whereas in this L1 and L2 caches only the code and data accessed by this particular core will be stored. And of course, these caches can also be separate for instructions and data or common for example, commonly the L1 cache is separated into a code section and a data section.

And because caches have very low access latency of you know tens of CPU cycles as compared to dram which is hundreds of CPU cycles then the CPU execution becomes much faster by using caches. But of course, you cannot have a lot of cache because the caches are the cache technology the memory is more expensive and therefore the caches will be smaller in size and you can only store the most recently used instructions and data.

(Refer Slide Time: 26:25)

**Reading into cache**

64 bytes

- Memory content fetched into cache in size of cache line (64 bytes)
  - When CPU requests contents at address X, 64 bytes around X are fetched
  - Why? Memory around recently accessed memory is most likely to be accessed in near future = spatial locality of reference (e.g., accessing array)
- Which level of cache hierarchy is data read into?
  - Inclusive cache: fetched into all levels of cache
  - Exclusive cache: fetched into lower level of cache
- What if cache is full?
  - Least recently used (LRU) cache lines are evicted into next level of cache
  - Why? Most recently used memory is most likely to be accessed again in near future = temporal locality of reference (e.g., for loop)

Handwritten diagram: C0 points to L1 and L2. L1 and L2 each have an 'X' and an arrow pointing to the other, indicating data flow or eviction.

So, now, let us understand how you will read into cache. So, when a memory location a data at say address x is accessed this is your D-RAM if some memory at address x is accessed then all the 64 bytes around it so the 64 bytes of data around it are fetched into CPU caches. So, these 64 bytes is called a cache line, so why we doing this when CPU was requested for one byte why are we not just getting that 1 byte or 4 bytes into cache that is because usually memory accesses have a property known as locality of reference or spatial locality of reference in this case. That is if you access one memory with high probability you will access the memory around it also in the near future that is called Spatial locality of reference.

Therefore, when you are going all the way to dram you might as well just get a larger chunk of it instead of just the data that was requested. Now, when you read something into cache where you store it when you have multiple levels of cache again there are two possible designs sometimes you have inclusive caches which is if you have a com CPU core and you have L1 L2 multiple levels of caches in an inclusive cache what you will do is when you get something from D-RAM you will store it in all the levels of the cache this is an inclusive cache.

On the other hand, in an exclusive cache what you will do is you will only store it in the level closest to the CPU core and you will not store anything in the other levels of cache. If the since the CPU core wants it you will store it in L1 cache and if your L1 cache is full then what do you do then you will evict the item and then push it into the next level of cache. So, any level of

cache once it is full what we will do is we will evict some data and push it into the next level of cache to make space in the lower level of cache.

And what items will you evict you will evict the Least Recently Used or LRU cache lines, why the Least Recently Used data because the most recently used data with a high probability will access it again but on the other hand data that you use long back you may not access it again. So, there is what is called temporal locality of reference when you access caches that is in time whatever you recently used you are likely to use it again.

For example, like a for loop. Therefore you will keep the most recently used data in cache and the least recently used data you will evict if there is no space in a certain level of cache you will evict it and move it to the next level in the cache.

(Refer Slide Time: 29:12)

The slide is titled "Writing into cache" in red. It features a diagram on the right showing two cores, C0 and C1, each with a local cache (LI). Core C0's cache contains a "dirty" entry (marked with an 'X' and a '+'). An arrow points from C0's cache to a box labeled "RAM". Another arrow points from "RAM" to C1's cache, which also contains an 'X'. Below the diagram, the word "RAM" is written in red. To the left of the diagram is a list of bullet points.

- CPU writes into cached memory, makes it dirty
  - Dirty cache line = different from original copy in memory
- When is dirty cache line written back to memory?
  - Write through cache = written immediately
  - Write back cache = written later (more efficient)
- What if dirty cache line in private cache of one core needs to be accessed by another core?
  - CPU cores exchange modified data with each other
- Cache coherence protocols keep CPU caches in sync with each other and with main memory

Now, the next thing comes up what about writing into cache. So, suppose there is a core C0 and in L1 cache there is some data at memory location x then when the CPU core writes into it the change is made first in the cache. And this cache entry that is written into is known as a dirty cache line, dirty because it is different from the original copy in the memory you got it all the way from RAM and you have modified it in the cache so this is a dirty cache line. Now, when now you cannot just keep the dirty cache line in cache you have to send it back into RAM eventually.

So, when do you do that? When is a dirty cache line written back into memory there are two possible options one is a write through cache which is you will write it immediately as soon as a cache line is modified in cache you will write it into RAM the other is a write back cache which is you will write it later when there is when you have more time or when the CPU is free the memory is free at some point later in time you will write it back, these are the two possible options.

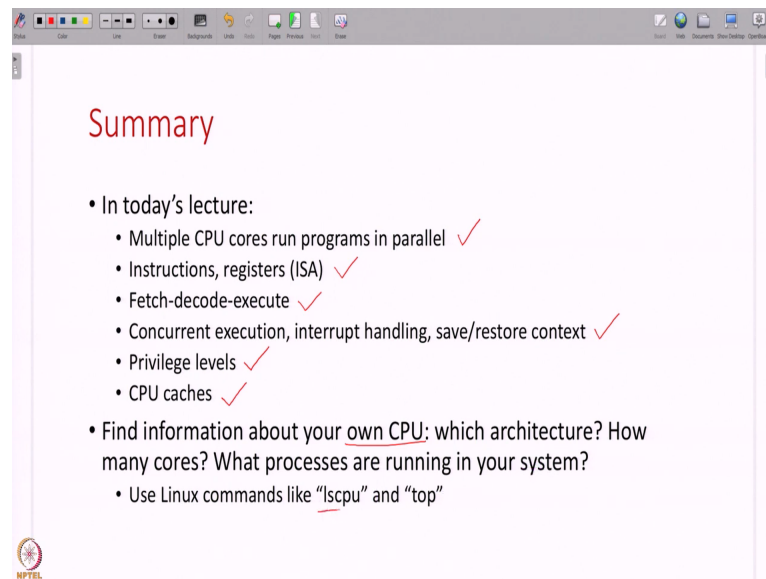
And most caches today use a write back policy because it is more efficient. Now, if you use a pack cache and you have accessed some memory location x and modified it and another CPU core also wants to access the same memory location x, then what happens it cannot go get it from

RAM. Because the value in RAM is not up to date this memory location X has been modified here and the latest value is in the private cache of core C0.

So, C1 wants to write it has to somehow coordinate with core C0 and get that latest copy of the data. So, if you use write back caches these CPU caches this private caches and multiple course have to coordinate with each other and run what is called a cache coherence protocol to keep these caches in sync with each other and in sync with main memory.

This care has to be taken otherwise if you end up reading the scale data from RAM while a more modified recent version exists in the private cache of another core then your program would not execute correctly, so these cache coherence protocols are needed to keep CPU caches in sync with each other.

(Refer Slide Time: 31:32)



So, that ends our lecture today on CPU hardware. In today's lecture what have we studied? We have studied how modern CPUs have multiple CPU cores to run multiple processes in parallel how a CPU is defined by its instruction set architecture the instructions and registers it has how the CPU runs a fetch decode execute cycle to execute programs. And how the CPU concurrently executes multiple programs at the same time by saving and restoring context and a similar mechanism is used to handle interrupts as well.

Then we have seen how CPUs guarantee isolation via multiple privilege levels and finally we have seen what are CPU caches which are like fast storage closer to the CPU to store the most recently used code and data. So, I will end this lecture by giving you a small exercise so find out information about your own CPU which architecture, which ISA, how many cores does it have how many processes is it running at a given time? So, if you have a Linux machine try to use commands like `lscpu` or `top` to understand more about your CPU and this is a way for you to get a hands-on feel for what we have studied in this lecture.

So, with this I would like to end this third lecture on CPU hardware. In the next lecture we will continue our discussion on the various hardware present in a computer system we will study main memory as well as I/O devices thank you everyone.