GPU Architectures and Programing

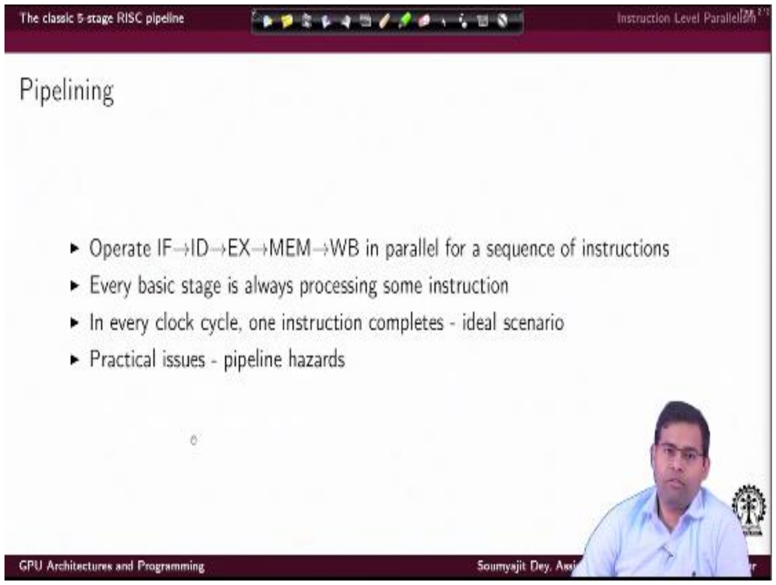
Prof R. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No. 2
Review of Basic COA w.r.t. Performance (contd.)

Hi, so we have been discussing about the different stages, presenting a basic RISC pipeline. Why pipelining really helps? And when I have a pipeline implementation. Although each of the instructions are taking multiple cycles to complete. Based on the cycle being defined with respect to each of the pipeline stages, but by looking at the end of the pipeline, it will seem that in every cycle have one instruction completing.

That means for each, each instruction the execution latency is one cycle right?

(Refer Slide Time: 01:01)

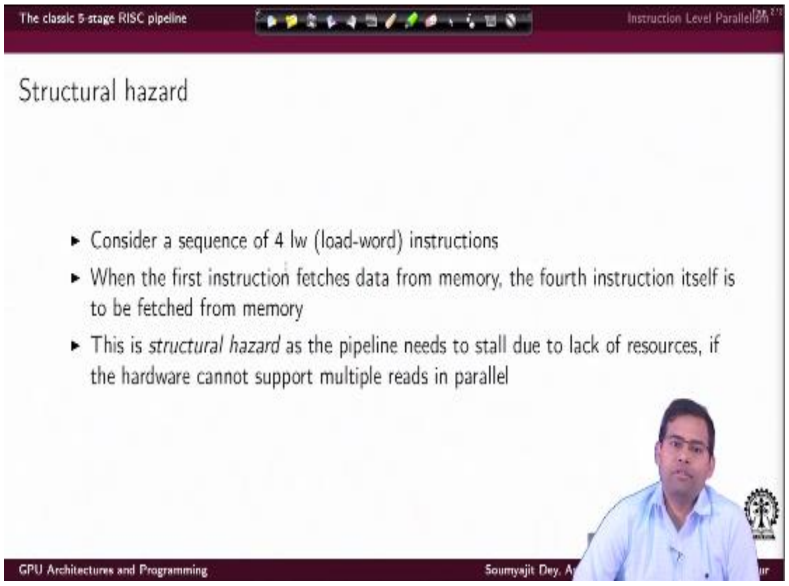

The classic 5-stage RISC pipeline
Instruction Level Parallelism
GPU Architectures and Programming
Soumyajit Dey, Ass

Pipelining

- Operate IF→ID→EX→MEM→WB in parallel for a sequence of instructions
- Every basic stage is always processing some instruction
- In every clock cycle, one instruction completes - ideal scenario
- Practical issues - pipeline hazards

Now, as we also discussed that, this is an ideal scenario where and it is also being getting assume that the instruction, when it's moving from one stage of the pipeline to the succeeding stage is the movement is seamless there is no issue of delay or anything, but that really doesn't happen. And these are the issues known as pipeline hazards.

(Refer Slide Time: 01:32)


The classic 5-stage RISC pipeline

Instruction Level Parallelism

Structural hazard

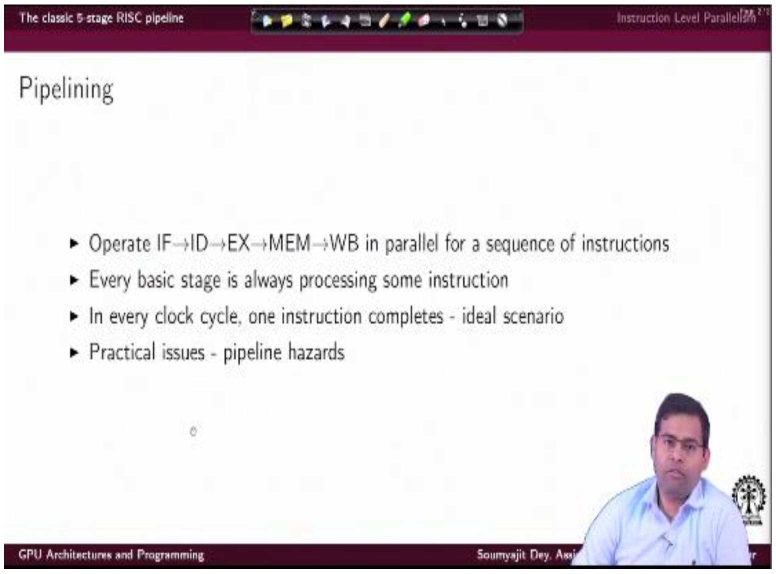
- Consider a sequence of 4 lw (load-word) instructions
- When the first instruction fetches data from memory, the fourth instruction itself is to be fetched from memory
- This is *structural hazard* as the pipeline needs to stall due to lack of resources, if the hardware cannot support multiple reads in parallel

GPU Architectures and Programming

Soumyajit Dey, A

So let us look at a few possible, such hazards. And in a practical pipeline and why such hazards actually happened. Such hazards can be classified into several types. One of them is structural hazard. So let us consider a sequence of four load instructions in MIPS we have them as Lw load- word instructions. So that means let us say I am loading data from memory for consecutive loads. So when the first instruction features data from memory. That means, it is in which stage of the execution.

(Refer Slide Time: 02:06)


The classic 5-stage RISC pipeline

Instruction Level Parallelism

Pipelining

- Operate IF→ID→EX→MEM→WB in parallel for a sequence of instructions
- Every basic stage is always processing some instruction
- In every clock cycle, one instruction completes - ideal scenario
- Practical issues - pipeline hazards

GPU Architectures and Programming

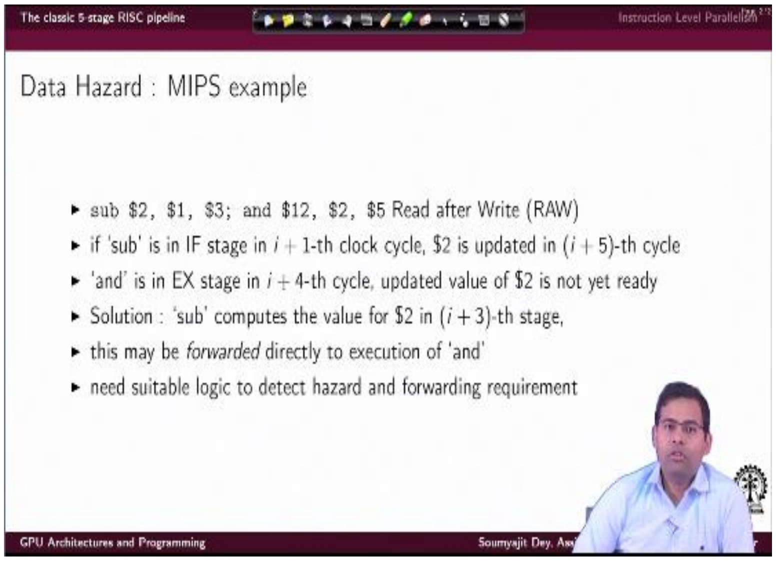
Soumyajit Dey, Assi

Going back, the first instruction is fetching data from memory. In the fourth stage of its execution. That means the second instruction is in EX stage, third instruction is an ID stage forth instruction is in instruction fetch stage. So that would mean, when the first instruction is fetching data from memory. The fourth instruction itself is to be fetch from memory.

Considering memory element, where I do not have the facility of parallel reads, even from different locations. This is a situation where the hardware does not have support in terms of resources, because both the instruction, and the data may be loaded in the same memory, element. And they cannot be read together. So this is what we call a structural hazard. It's a hazard, that means the pipeline needs to stall.

Before fetching the for the fourth instruction, and it needs to complete that data read for the first instruction in its memory stage. And the reason for the stall is lack of resources, because I have only one memory element as a resource, and I am assuming that although the data is located at different locations in the memory there's no memory is not a multi port supported.

(Refer Slide Time: 03:30)


The classic 5-stage RISC pipeline
Instruction Level Parallelism
GPU Architectures and Programming
Soumyajit Dey, As

Data Hazard : MIPS example

- sub $2, $1, $3; and $12, $2, $5 Read after Write (RAW)
- if 'sub' is in IF stage in $i+1$-th clock cycle, $2 is updated in $(i+5)$-th cycle
- 'and' is in EX stage in $i+4$-th cycle, updated value of $2 is not yet ready
- Solution : 'sub' computes the value for $2 in $(i+3)$-th stage,
- this may be *forwarded* directly to execution of 'and'
- need suitable logic to detect hazard and forwarding requirement

Consider another example of another kind of hazard, there's a data hazard.So what happens in a data hazard. Let us have this example of two instructions, instruction for subtraction. So, I have having the different registers enumerated as 123, and the instruction encoding is such that here we are representing the registers by $1 $2 $3 like a standard for MIPS instructions. So, this is a substraction instruction.

The idea is that you want to subtract the content of 3rd register from the content the 1st register and store the result in register 2 this instruction is followed by and operation, where the content of register, 2 and 5 will be Anded. And the result is to be stored in this register 12. Now, this is our data, called a data hazard, because we have what we call popularly as a read after write dependency, let us look into it in, with more attention now.

So let us consider that this sub instruction has entered the pipeline, that means it is in the IF stage of the pipeline in some i+1-th clock cycle. So, if it's in the IF stage in the i+1th is closed cycle, then it moves to the decode stage i+12-th clock cycle, it moves to the execute stage in the i+3-th clock cycle, and that would mean. After the execution of the instruction. The content of this registered 2 is getting updated in the i +5-th clock cycle, is not it.

Because i+1 instruction fetch, i+2 instruction decode i+3 instruction execute, substraction instruction executed. Then i+4 is a memory stage, and then I have right back happening in the i+5-th cycle. And that is when $2 is getting updated. But what is the situation with the And instruction. So when the sub instruction is in its i+5-th cycle. The end instruction in an ideal scenario will just be following it right?

That means, in the i +4-th cycle sub was in the memory stage. And in the same i+4-th cycle, And is in the execute stage. That means, And demands a value, a proper value in the registered 2 in i+4-th cycle. But when is registered to getting updated by sub is getting updated in the i +5-th cycle, which means if sub and And are executing in the pipeline. One after another, and following sub exactly with the deeper with the, with no intermediate lag.

Among them in terms of execution stages. Then, And requires an updated value in i+4-th cycle, and the value is not ready, it is supposed to get ready in the i+5-th cycle. So, that is why you start hazard, right, because in that case, I should not be able to execute, and without any delay, getting inserted in the pipeline. Unless I adding some extra hardware support. What can be the solution in this case, the solution is.

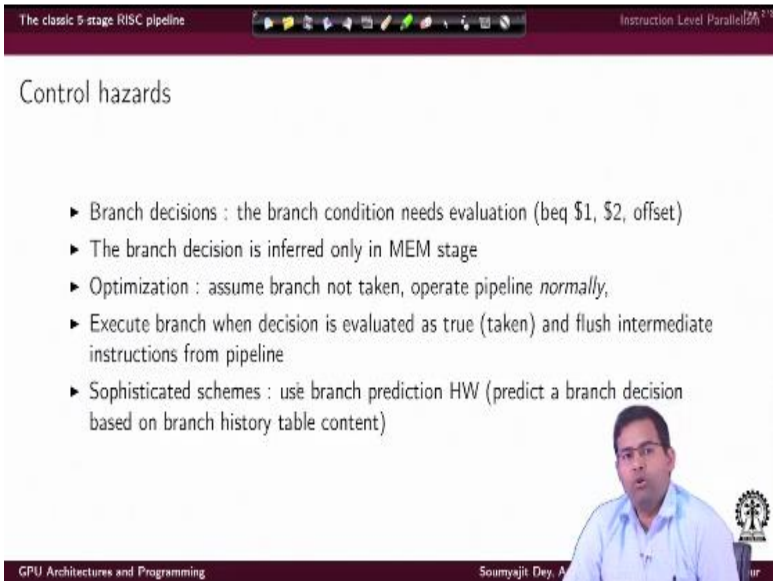
Observe the scenario that although the instruction sub updates the content of register 2 in the i+5-th cycle. When does it get computed. It gets computed in the execute stage for instruction self which is the i+3-rd cycle right? So, the value is actually ready, is just not outdated. So what if we have some extra hardware support such that, whenever the value is ready. It can immediately be forwarded wherever it is required for execution.

Without the value getting transmitted to the register file, and then getting used from the register fight. So these are called forwarding units, which may be present inside a pipeline for resolving these kinds of dependencies. But then again there is the issue that hardware. There's the CPU data path should have support in terms of detecting these kind of hazards that means while executing these instructions sequence.

If I am using a CPU data path the simplistic one I showed earlier, it will not be able to do this right? The hardware should be able to look into this instruction sequence and identify that this instruction sequence has a read after write dependency. So this can be modeled by a formal condition. And that condition needs to be checked by the hardware. And when the chip details his dependency.

Accordingly, the forwarding units should be activated. As we got to understand. These are the ways in which data hazards can be detected and suitably handled inside the pipeline.

(Refer Slide Time: 09:01)


The classic 5-stage RISC pipeline

Instruction Level Parallelism

Control hazards

- Branch decisions : the branch condition needs evaluation (beq $1, $2, offset)
- The branch decision is inferred only in MEM stage
- Optimization : assume branch not taken, operate pipeline *normally*,
- Execute branch when decision is evaluated as true (taken) and flush intermediate instructions from pipeline
- Sophisticated schemes : use branch prediction HW (predict a branch decision based on branch history table content)

GPU Architectures and Programming

Soumyajit Dey, A

Some example of other kinds of hazards, for example control hazards. So what's that? Take the execution example of a branch decision. We have branch instructions, and they need to be supported by any processor Why? Because, think of the situation that you have written a problem in the C language, definitely there are changes in execution flows, which you model by writing code using IF ELSE blocks.

Now, those will be translated to branch conditions or branch instructions. When it gets translated to machine code, right? So, definitely, any CPU has to support branches structures. So, if we take the example of this branch instruction branch if equal $1, $2 of set right? So, in case the content of the registers is found to be equal, then the branch has to be taken branch is equal, right? So, this condition of branching has to be evaluated.

So when this branch instruction is getting executed where will this condition be evaluated, it will be evaluated in the execute unit right? So the result of this branch conditions, evaluation is inferred only when the instruction is in the MEM stage right? So, up to this point, what shall be going on for the instructions, after this branch instruction because as we remember, our standard pipelining approach has been you execute one instruction.

In every cycle you provide the pipeline with one instruction in every cycle. Following that philosophy. When the branch instruction goes from fetch to decode that means when the meaning of branch is getting decoded the instruction fetch has to happen. IF stage should be fetching the next instruction. When the branch condition is getting evaluated. IF stage should be fetching another instruction.

And when the branch conditions meaning has been inferred and it's available in the MEM stage. The IF stage should be fetching MLF the third instruction right? But then only after the branch conditions result the branch decision has been inferred it is known whether to follow the current sequence of instructions. Or whether to abort it and move the PC content to some updated address and accordingly fetch instructions from the new branch of execution.

Now, that is why we call it a hazard why? Because just after fetching the instruction is really not alone right? Now what can be an obvious solution in this case that you execute the branch. When the decision is evaluated as true and flush, intermediate instructions from the pipeline. That means you, you simply keep on feeding the pipeline with instructions right. However, when the decision is found to be true that means the branch has to be taken.

That means whatever further instructions you have sent into the pipeline, they, their execution doesn't have any role. Then you flush the pipeline, and then you move to the new address, and then start executing instructions from that address the branch address. But is that a good way to go about handling branch, maybe not why? Because then I am always thinking that the branch decision.

I mean, will I mean there is a few there's in a statistical specificity speaking there's a 50-50 chance, and I am not considering facts like for what kind of problem we are talking about. And all that, I am not modeling the branch decision, and just thinking is fair enough to keep on executing instructions one after another. And only when the decision is known, then I flush the pipeline and go to the branch address.

That is why there are sophisticated statistical schemes implemented which is in hardware in modern CPUs. So they have this specific kind of hardware called branch predictors. So branch predictor is a lightweight hardware, which could contain something like a branch history table. That means, what were the branch decisions in a few of the earlier branches. And based on not, it will try to predict whether to take or not a current branch.

So then, it's not as nice as the previous approach. Its banking on statistics of previous branch history and trying to take an informed decision based on that. However, again if it's found that when the branch condition is evaluated is something different from whatever was predicted, then the pipeline flushing operation is definitely required.

(Refer Slide Time: 14:09)

The classic 5-stage RISC pipeline

Instruction Level Parallelism

Section 2

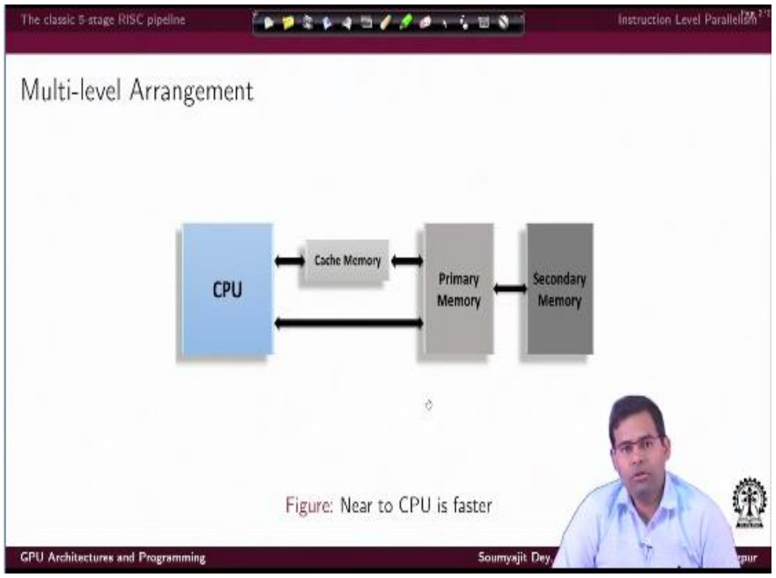
The memory hierarchy

GPU Architectures and Programming

Soumyajit Dey, As

So that is all about very brief overview of pipelining, and how the pipelining deviates in several of the known ideas. Some of those scenarios can be handled. We feel that this is small but necessary overview, which will be helpful in going further into the GPU part but before that. There is something we need to discuss about how memory is organized, in a standard CPU.

(Refer Slide Time: 14:42)


The classic 5-stage RISC pipeline

Instruction Level Parallelism

Multi-level Arrangement

Figure: Near to CPU is faster

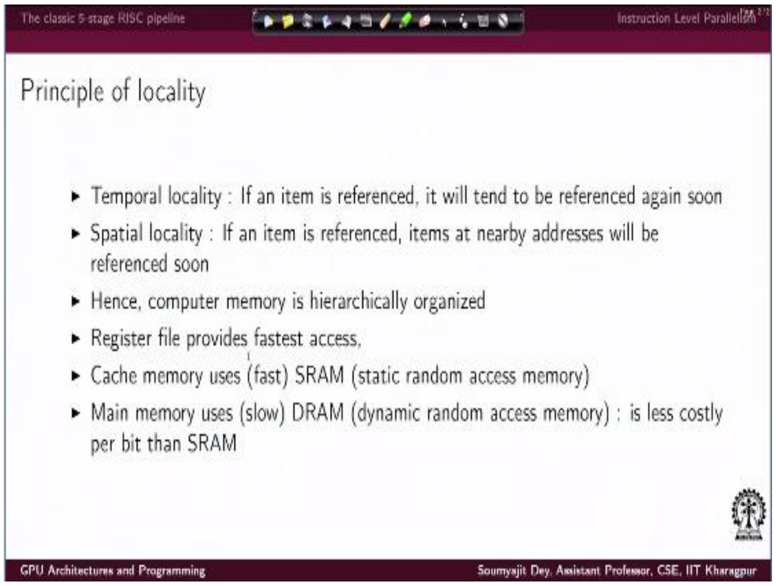
GPU Architectures and Programming

Soumyajit Dey

So, the instructions that the CPU will be crunching. They are fetch from a primary memory or main memory that is known to us, more like a RAM. And then this primary memory will be connected to a secondary memory or disk drive in between the CPU and the primary memory. We have something called a cache memory. So these are the three primary levels of memory, which defines the storage.

From which instructions and data are fetched and executed by the CPU as and when required. Now, the memory segment which is near to the CPU is the faster cache memory is not a discrete thing. It's something which is on chip with the CPU.

(Refer Slide Time: 15:33)


The classic 5-stage RISC pipeline

Instruction Level Parallelism

Principle of locality

- Temporal locality : If an item is referenced, it will tend to be referenced again soon
- Spatial locality : If an item is referenced, items at nearby addresses will be referenced soon
- Hence, computer memory is hierarchically organized
- Register file provides fastest access,
- Cache memory uses (fast) SRAM (static random access memory)
- Main memory uses (slow) DRAM (dynamic random access memory) : is less costly per bit than SRAM

GPU Architectures and Programming

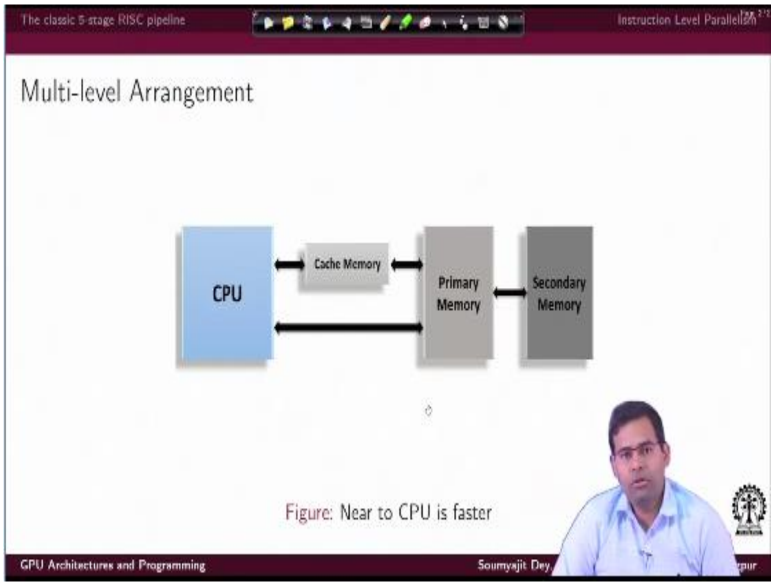
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, why really is cache memory used, the specific reason between them. Manners from the principle of locality, but is this principle of locality there can be kinds of localities temporal locality as well as special locality. So what is temporal locality. If a program references an item, let us say a variable or another location. Then, there is a high chance that that item will be referenced again soon by the program.

This is temporal locality locality with respect to time. Similarly, we have the concept of special locality. Which would mean that if an item is reference, let us say an array content, a1, items which are located at nearby addresses in the memory are likely to be referred as soon. So if a program fee refers. That means loads the content of an array a, let us say a1 is highly likely the content of a2, a3.

Those will also be referenced very near in very near future. So this notion of spatial locality and they hold true for most of our programs. Based on that computer memory gets hierarchically organized as we told this go back to the picture.

(Refer Slide Time: 17:01)


The classic 5-stage RISC pipeline

Instruction Level Parallelism

Multi-level Arrangement

Figure: Near to CPU is faster

GPU Architectures and Programming

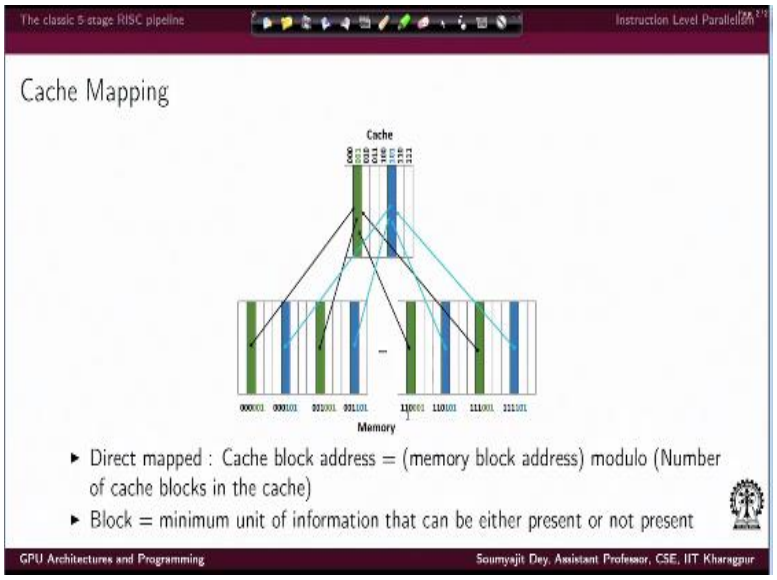
Soumyajit Dey

The memory element line nearest to the CPU is going to be the fastest. The one that is going to be further down on the right hand side will be slower. So cache memory will be some, some kind of technology which is faster with respect to primary or main memory, and the secondary memory would be slower with respect to this primary memory. But also, as we go right hand side, the speed decreases was a size increases the fastest memory element.

To be more specific would be the CPU registers. Because they are sitting really near to the arithmetic logic units remaining execution units. So in that way the register file provides the fastest access to the data, you just load from the register file to the ALU, and do some computation. The cache memory uses the SRAM or static random access memory technology. So SRAM technology is fast, but costly also.

The main memory uses comparatively slower DRAM technology, which is less costly per bit than as well. That is why I can have a big main memory, but I cannot have a big cache memory is fast, but costly. The main memory is low, but I can have a much bigger memory.

(Refer Slide Time: 18:22)



The classic 5-stage RISC pipeline

Instruction Level Parallelism

Cache Mapping

- Direct mapped : Cache block address = (memory block address) modulo (Number of cache blocks in the cache)
- Block = minimum unit of information that can be either present or not present

GPU Architectures and Programming

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now the question is, how really do we decide that okay, I am bringing in data from main memory location, I am going to load it in the cache memory, and then I will load it in the corresponding register, do some computation. And all that, but where really should I load the data in the cache memory? That gives rise to the problem of what we call us cache mapping. So what's cache mapping is basically a set of rules that will tell you that okay.

If you bring data from some memory location, mi, what should be the location in the cache, where that data has to be stored. Now of course, there can be different policies or different functions which dictate the mapping. We will discuss the popular ones, and their relative advantages and disadvantages. So, let us start with one of the strategies. So, there can be a direct map cache mapping.

So as we discussed the the cache is a much smaller memory segment with respect to the main memory. So here in this picture. We have the main memory, these kind of again representative from the well known book on Harrison Patterson. So this is the main memory, we are trying to show the locations of the main memory. And we have a cache which is a much smaller memory, the addresses in the cache are kind of marked here as consecutive addresses.

Starting from 000 to 111 every location in this cache is we are calling it as a cache of block. So what's the block? A block is defined as a minimum amount of information that can be either present or not present. Now, what does this really mean? Earlier we have talked about the concept of memory work right. A word means, the amount of data that may be stored together in a register or the amount of data that can be passed on the CPUs bus right?

For example, if I am talking about a 32 bit CPU that means the memory word length is 32 the register file is containing registers of 32 bit size. Memory becomes byte addressable that is why when we increment the program counter I do a PC to PC+4. I increment by 4 bytes, I go to the next memory world, which is the next consecutive 32 bits in the memory. Now when we talk about cache is not really the case that when I read data from memory.

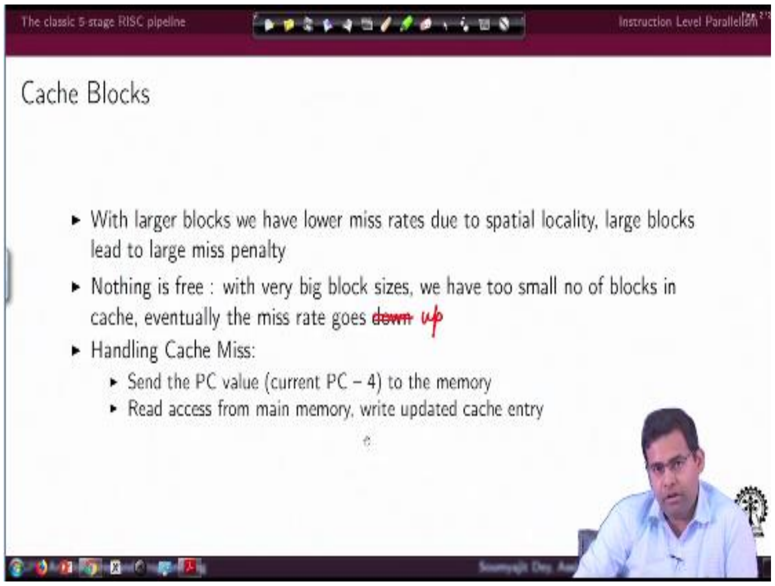
I will read 1 memory word and put 1 memory word in the cache, I may read more than that, I may read, let us say 4 consecutive memory words. And put them in the cache. So this is what is defined as the cache block size. So then I would start saying that okay, this is the minimum amount of information that I would read from the memory put in the cache.

Or I will, I will update that cache location with some other locational data from the memory right? So then what is the addressing scheme of the cache. So the address of a cache block is given by, suppose I am trying to load data from the memory. I define a memory blocks address. So that is from where I am starting to load data. Let us say I am loading four consecutive cache memory words, this the block size is four.

So the address of the memory block modulo the number of cache blocks. So just as an example here in this picture, how many cache blocks are there, so I start from 000 to 111 right, so I have 12345678, 8 cache blocks. That means, from every memory location, you do modulo 8 operation, you take the address of every memory any local memory location, you do a modulo 8 operation.

Whatever you get is the location in the cache, where the data from this memory block will get loaded, so that is the idea of a direct mapping for every block in memory. You have a corresponding unique mapping to some location of the cache, some unique block in the cache.

(Refer Slide Time: 22:57)



The classic 5-stage RISC pipeline — Instruction Level Parallelism

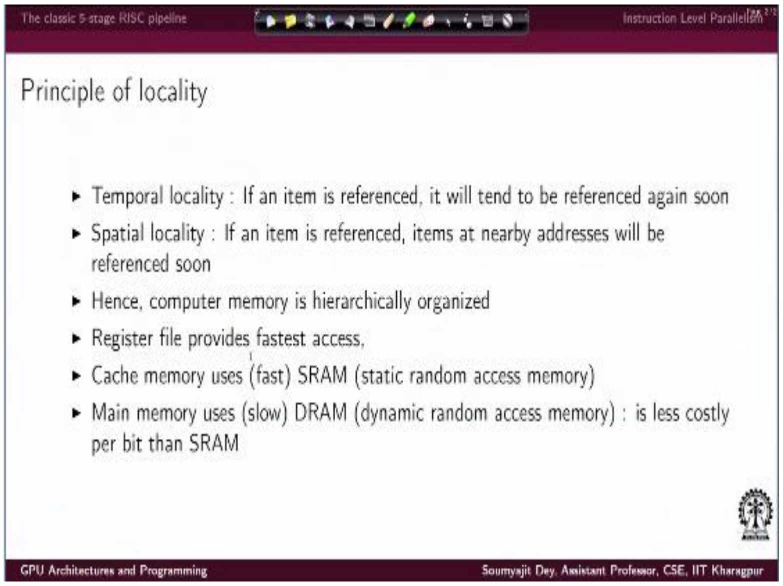
Cache Blocks

- With larger blocks we have lower miss rates due to spatial locality, large blocks lead to large miss penalty
- Nothing is free : with very big block sizes, we have too small no of blocks in cache, eventually the miss rate goes ~~down~~ up
- Handling Cache Miss:
  - Send the PC value (current PC − 4) to the memory
  - Read access from main memory, write updated cache entry

Now, coming to this idea of cache blocks, how to decide what should be the size of a cache block. As we say that it doesn't mean, it may not make sense that you load only one word from the memory, because you may have a wider access to the memory. So, you may load multiple constituting words and that divides the cache block size of the memory block size. Now then the question is, how large should be the block.

And what is the impact of the block size? If I have large blocks in the cache. That means whenever I read data from the memory to the cache. I read more amount of consecutive data more amount of consecutive memory words and store it in a cache block. That is what I mean by a large block size. Then we have lower misread due to special locality. Why is that? Because, as we said the principle of special locality sets. Let us go back.

(Refer Slide Time: 24:02)



Principle of locality

- Temporal locality : If an item is referenced, it will tend to be referenced again soon
- Spatial locality : If an item is referenced, items at nearby addresses will be referenced soon
- Hence, computer memory is hierarchically organized
- Register file provides fastest access,
- Cache memory uses (fast) SRAM (static random access memory)
- Main memory uses (slow) DRAM (dynamic random access memory) : is less costly per bit than SRAM

If an item is referenced items that nearby addresses will be referenced soon. Right. So then, in case I have loaded a specific data element from memory. The block size is large. There is a good chance that many of the consecutive memory locations, which are getting loaded. Since the block size is large, I have loaded many search consecutive locations. And some of them would get reference soon. And they are already in the cache.

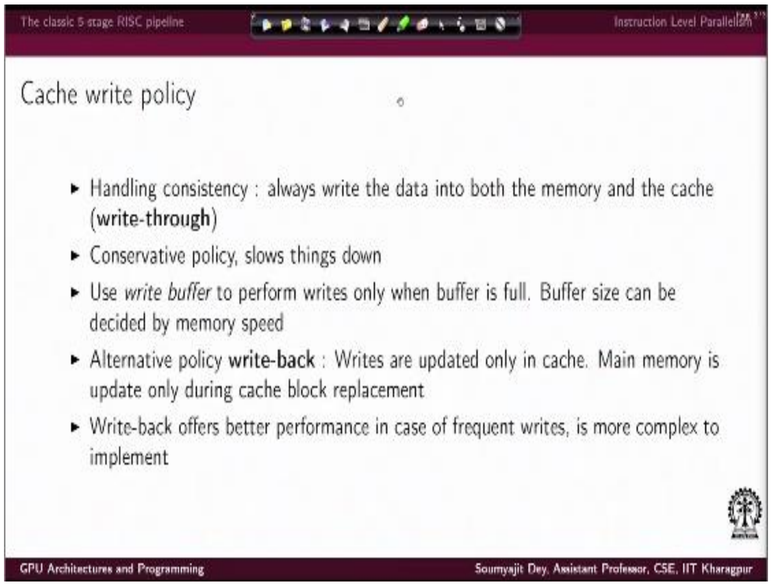
So, I will have a lower miss rate due to this special locality. But then what is the word. The word is since I am loading data in large blocks am reading a lot of data together. So, if there is a miss the miss penalty is large, in case of a mis. I have to do a memory read and memory read, it will take a lot of time. However, I cannot keep on increasing the block size, because then at some point of time, the mis read will not keep on going.

I mean, I mean, I will not it I mean, it will not be the case that the mis read is low, but I will have the situation that the mis read goes up. So with very large block size. We have two small number of blocks, and eventually the miss rate goes up, sorry, here. maybe we should do a correction. Yeah. So that is it. This should be.Now, how are things taken care of, when there is a cache miss. So of course, in case there is a cache miss that means of value has been referenced by the CPU.

The first place to search for that memory address should be in the cache is found that okay content of that address is not loaded. So then the location of the PC has to be updated by current PC -4. Why because they instruct by the time this is in getting done, the instruction count has already gone up.So that is why you do a current PC -4, and you send that value to the memory right?

So that is how you handle a cache miss the sequel you send the PC value, the program counter value to the memory, and you do a read access from the main memory, and after doing the read access using the corresponding mapping scheme. You find out what is the cache block location, and, accordingly you update the cache.

(Refer Slide Time: 26:57)


The classic 5 stage RISC pipeline

Instruction Level Parallelism

Cache write policy

- Handling consistency : always write the data into both the memory and the cache (**write-through**)
- Conservative policy, slows things down
- Use *write buffer* to perform writes only when buffer is full. Buffer size can be decided by memory speed
- Alternative policy **write-back** : Writes are updated only in cache. Main memory is update only during cache block replacement
- Write-back offers better performance in case of frequent writes, is more complex to implement

GPU Architectures and Programming

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So an important issue, apart from deciding the mapping of data to the cash is how really the cache should handle variable updates. That is what should be the policy based on which cache data should be updated back to the memory. Now why is this important, because you bring data from the main memory to the cache, you load the data to the registers. You do some operation you update the register content, which now need to be return back to the cache.

And further back to corresponding memory location. So when do we update the memory location, after updating the cache. Now one simple policy can be write-through, that means, always write the data into both the memory and the cache, so whenever I update our cache location. I also update the corresponding location in the main memory. Now, is it good. Not really, because it's the conservative policy.

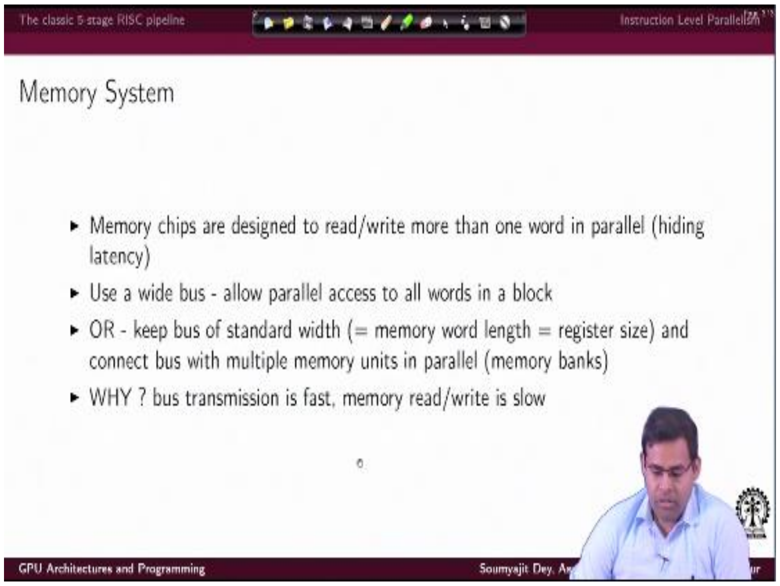
Because then every time I do a right on the cache. It's accompanied by a right on the main memory it's going to slow things down. Alternatively, you have a separate write buffer, so that in the write buffer you sequence pending memory updates. So by default you have update cache. And in the right before you sequence these pending memory updates for the main memory, and when the buffer is full.

Then you carry on the updates, or should be the buffer size can be decided by the memory speed, right? Because if the memory as a high latency The buffer needs to be big to hide the latency. An alternate buffer policy will be write-back. That means, by default, whenever a program is updating variables you update in the cache. Now you update in the main memory. Only when that specific cash block is replaced.

Why is this good? Is good because whenever the variable will be referred in the effort in the future. You will anywhere refer from the cash first, where the variable value is already updated. The issue will come only when some other cache load is going to override this data, that means this current updated cache block is going to be replaced. Then this update needs to be transmitted back to the main memory.

So this is like doing a lazy right. Doing it isn't where it's needed, and the need actually comes with the cash blocks gets replaced. This gives definitely better performance because it's not conservative, more so in case of frequency rights, because then you do not write back to the memory that often, but the bad thing is, it's more complex to implement the hardware needs to do a lot of stuff so it's more complex to implement.

(Refer Slide Time: 29:44)


The classic 5-stage RISC pipeline
Instruction Level Parallelism
GPU Architectures and Programming
Soumyajit Dey, A

Memory System

- Memory chips are designed to read/write more than one word in parallel (hiding latency)
- Use a wide bus - allow parallel access to all words in a block
- OR - keep bus of standard width (= memory word length = register size) and connect bus with multiple memory units in parallel (memory banks)
- WHY ? bus transmission is fast, memory read/write is slow

Now, with respect to the memory system. How is this issue of reading and writing data from the memory, arranged. Now, memory chips are fundamentally designed to read right, more than one word in parallel. Why of course you want to hide the latency, cache is fast. The main memory is slow. So that is why whenever you transact with the main memory, you need to. You like to transact more number of things in parallel.

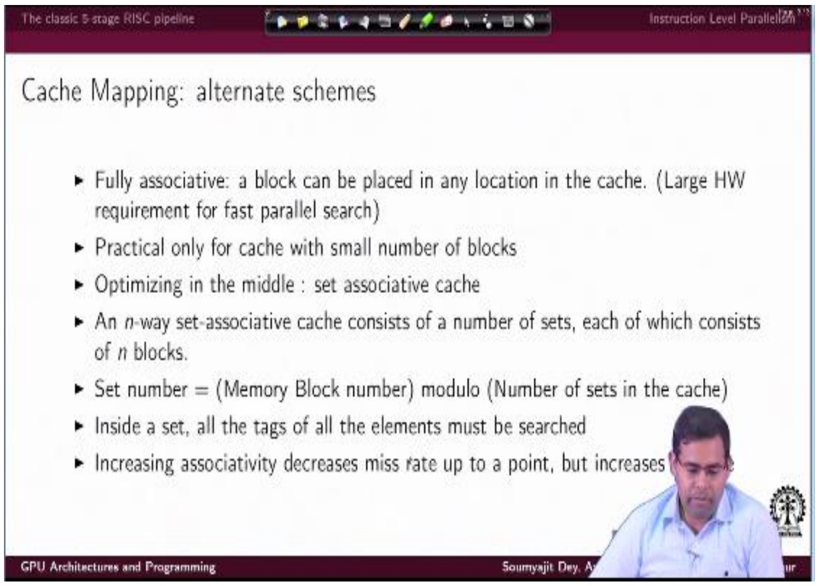
So that whenever you do an access to the main memory, you bring more things, or you write-back more things. Now, how can this really be done in the physical hardware, one option is , you use a wide bus. That means you are doing parallel access to multiple words in a memory. let us say you are accessing multiple words, all of the words in a block in parallel, but then you need the bus to be wider. Maybe you have a 32 bit system.

But the bus is not 32 bit in this scheme you require a wider bus. Alternatively, you can keep the bus of standard with, which is equal to the memory word length or equivalent to the register size. But then, although the bus is of standard width, you connect the bus with multiple memory units in parallel. That is the memory bank, the memory chip is not one single chip. There are multiple small memory chips together.

And read and write operations can be done in parallel, from all these different memory banks. Why is this a good optimization? The reason is, when we talk about memory access, we talked about the total time that means accessing the data from the memory, reading or writing, as well as transmission through the bus. The bus is fast. The issue is more with the memories read and write, which is slow.

That is why it may make sense to keep the bus of standard width and creating banks in the memory so that I can read or write in parallel.

(Refer Slide Time: 31:46)



The classic 5-stage RISC pipeline | Instruction Level Parallelism

Cache Mapping: alternate schemes

- Fully associative: a block can be placed in any location in the cache. (Large HW requirement for fast parallel search)
- Practical only for cache with small number of blocks
- Optimizing in the middle : set associative cache
- An n-way set-associative cache consists of a number of sets, each of which consists of n blocks.
- Set number = (Memory Block number) modulo (Number of sets in the cache)
- Inside a set, all the tags of all the elements must be searched
- Increasing associativity decreases miss rate up to a point, but increases

GPU Architectures and Programming | Soumyajit Dey, A

Now coming back to the cache mapping. We discussed how cache mapping can be done in a direct map where there can be alternate schemes, for example, instead of doing a direct map. I can do a fully associative mapping of a cache. So what's that? A fully associative mapping would mean any block of data from the main memory can be placed in any location in the cache. If you remember in direct map.

We were doing a modulo number of cache blocks operation and deciding what should be the location in the cache, where a memory block gets map right? So in that way for every memory block there was a unique location of the cache block. Coming to fully associative, we simply say that no you can load anything anywhere in terms of blocks. But why is that a bad thing because then when you were referencing data from a cache.

You have to search all the cache locations. Because there is no mapping, there is no mapping that look at this memory address. If it is at all present in the cache. It should be located exactly in this location, it's not so. So you need a large hardware for doing the parallel search. that is why it may not be practical. It is practical only for caches with very small number of blocks. If you consider a big cash pool is it is a bad scheme.

So, we can optimize in the middle between these two extremities of direct map And fully associative, we go for an in the middle approach we decide decide that let us say that mapping scheme can be what we call a set associative cache is defined a set of associative cache. We say a cache is in with set associative. That means, cash has got a number of sets, and each of them consists of n blocks.

So now instead of talking about cache. As a thing, containing a set of cache blocks. We say that it's a hierarchical hardware containing things that we call a sets, and each set contains n blocks. That means every memory block can get mapped earlier in a direct map scheme, it was mapping a memory block to a cache block. I do not do that right now. I map a memory block to our caches set by the formula.

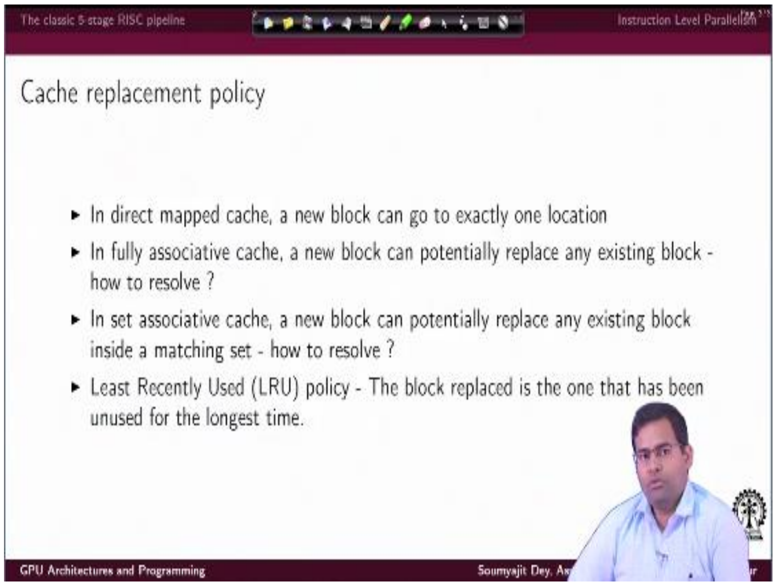
The number of the set is the index of the set is, the memory block number modulo number of sets in the cache, why do I do that? Because now keep an memory block number. I know in which set of the cache, it should be in case it is present at all. And inside that set, I have, n different locations to search for instead of searching the full cache. I searched, only a corresponding set. So in that way, I have an in the middle approach.

The scheme is not fully associative, I do not have a complex search by hardware. I have to only search inside a set. At the same time, we are elevating one of the issues with direct map cache. That means, in direct map. If there were multiple memory locations, which were getting mapped due to the module operation. The same cache block number one, getting loaded to need to replace the other, but now I can have due to this idea of sets and one set containing n blocks.

I can have multiple memory locations mapping into the same set, staying together in the cache. So this is the advantage we get by doing the trade off. And in that way. The, there is a related question that how do you decide this n the value of this n? Now it's more of a design question. You have to choose a suitable associated to value, based on what is your target design criteria. Look at this important things like if you increase the associativity.

It decreases miss rate up to a point. Why is that so? Because if you increase set associativity then coming back to the earlier point. I have the possibility of storing more a number of memory blocks mapping to the same set together. So, maybe it will reduce the mystery up to a point. But what is the disadvantage? It increases the heat time. Why, because again I have to do some amount of search inside the set. So, if the set size goes on becoming big, the heat time increases.

(Refer Slide Time: 36:41)



The classic 5-stage RISC pipeline — Instruction Level Parallelism

Cache replacement policy

- In direct mapped cache, a new block can go to exactly one location
- In fully associative cache, a new block can potentially replace any existing block - how to resolve ?
- In set associative cache, a new block can potentially replace any existing block inside a matching set - how to resolve ?
- Least Recently Used (LRU) policy - The block replaced is the one that has been unused for the longest time.

GPU Architectures and Programming — Soumyajit Dey, As

Now, another important thing is the replacement policy. So what is that. Now, as we have discussed that suppose in a direct map scheme, new block comes. And this block is going getting mapped to some cache block position where there is already some data. That means that data has to be replaced. So, in this case there is no problem, right, because every memory location, every memory block has a unique mapping with a block in the cache.

So whenever it is being referenced. But the corresponding position is filled up, you have to replace it with this thing. So there is an issue. But how about fully associative? Then a memory block can potentially replace any existing block right? Because anything can go anywhere. So then comes the question that how to resolve this that suppose I have loaded, a memory location. And then I have to load, another data.

I have to find out whom to evict because there is no rule, any data point, any block from the memory can be mapped to any block location in the cache, there is no rule. So how do I resolve whom to evict. The same question also comes in case of set associative cache, because now I suppose I am trying to load a specific memory block. It can potentially replace any existing block inside a matching set.

Because the operation if you remember, was this. If you are given a memory block number you identify the corresponding set number. Now inside the set you have n possible blocks to replace. Which one to replace the most standard policy that is used is Least Recently Used policy, which means the block replaces something which is being unused for the longest time, that means in with every cache block.

We have an idea that okay for how long it is lying there without being used. So there is a least recently used block right? So you will replace that one. So this is kind of a summary of the different pieces of information. Which is really important with respect to memory system design its hierarchy, its access mechanisms. Its mapping schemes and replacement policy, which are pretty much used, and we will learn concepts in RISC processor. So with this will complete the present lecture. Thank you.