

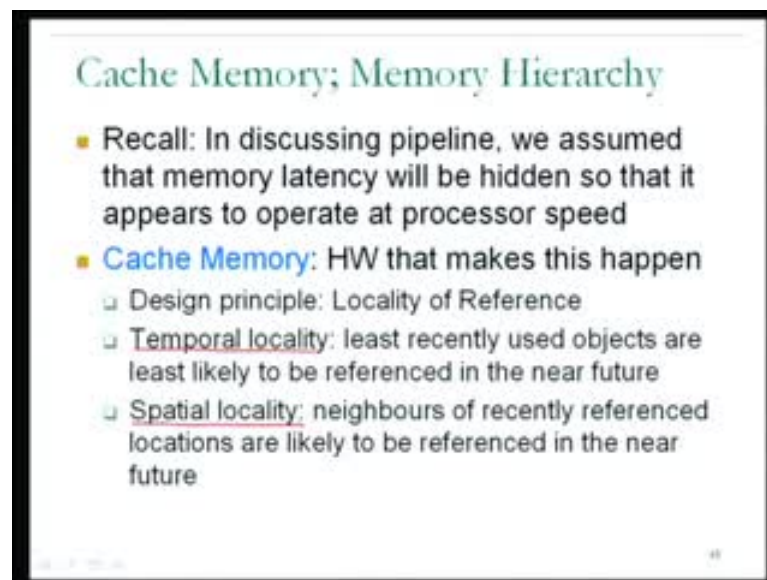
High Performance Computing
Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 06

Lecture No. # 26

Welcome to lecture 26 of the course on High Performance Computing. In today's lecture, we start on the next item of our agenda; this is item number 6, where we talk about cache memory. I had given you a general introduction to this topic towards the end of the previous lecture.

(Refer Slide Time: 00:22)



Now, this topic is important to us from our discussion of pipelining and process of architecture, because we realize that in discussing how instructions could be executed, it was important to make rather strong assumption that memory latencies would not be seen by the processor most of the time, because the main memory is so much slower than the processor. We just bypass the problem by saying that there would be some hardware called cache memory, which would solve that problem and make the assumption valid. It is now time to look into that assumption and how it is realized in the hardware.

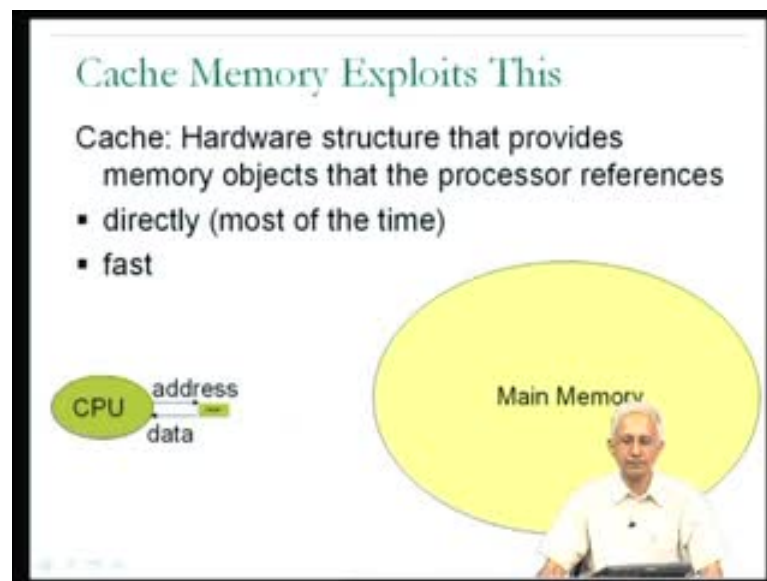
The cache memory is the name of the hardware that makes that assumption a reasonable assumption. As I had mentioned in the previous lecture, the design principle of cache memory is something that we have seen, and we talked about virtual memory, the

principle of locality of reference of which they are two aspects that it is important to understand. Temporal locality which tells us **that typical program behavior**, it is the least recently used entities, whether they are instruction or data that are least likely to be referenced in the near future.

Remember that it was important in the case of our discussion of virtual memory, in the connection with page replacement policies, to have some kind of understanding of program behavior and the sense of what it is likely to do in the near future. So, this principle gives us a model or understanding of typical program behavior from that perspective.

Second aspect of locality was spatial locality which leads us to understand there, for typical programs, the neighbors or neighboring memory locations to a location which is currently being referenced are the ones that are likely to be referenced in the near future.

(Refer Slide Time: 02:30)



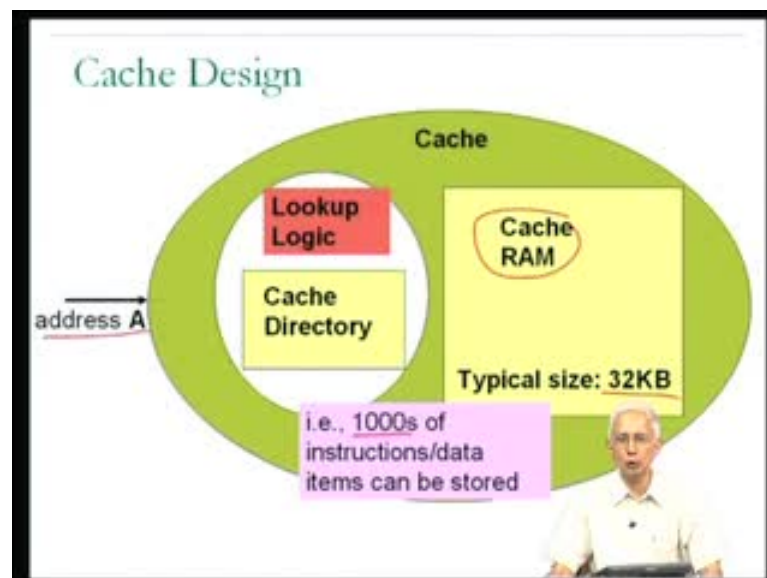
The principles of temporal and spatial locality which were of relevance when we talk about virtual memory, once again, being principles of models of how programs access their instructions or data, will also be relevant in our discussion of cache memory.

Now, just to illustrate how cache memory exploits the principles of locality of **reference**. First of all, remember that cache is a hardware entity and **it** in some sense provides what the processor wants directly, rather than expecting the accesses to be satisfied out of the

main memory, which is slow. This will happen hopefully most of the time, not necessary all the time, but most of the time at a very high speed.

The picture we should have in mind is that the CPU, as always, sends requests to main memory for instructions or data in the form of an address, and some later point in time get back the piece of data or instruction. We now understand that there is this intermediate piece of hardware called the cache memory, which typically most of the time provides the instruction of the data at high speed. I have shown the cache memory as being much smaller, significantly smaller, than the main memory in this diagram.

(Refer Slide Time: 03:26)



The general principle of how caches operate and we are blowing up that small green block into **at** the larger block, is that when an address A received by the cache hardware from the **processor...** remember that processor sends an address to the cache memory as per this diagram. When the address reaches the cache, obviously the cache must have some amount of very fast memory in order **to able** to provide data or instructions quickly, so there must be some fast memory inside the cache. But in addition, since clearly not all of the contents of the main memory could be present in this small cache, there must be some hardware within the cache which can determine which instructions or data from main memory are currently inside the cache and inside the fast memory of the cache.

Technically, what that hardware **what** have to do is, given the address A that the process has requested determine whether or not the contents of memory location A are currently inside the fast memory of the cache, which is what I labeled as, do I have it? Logic, once again, I am using the term logic to refer to circuitry.

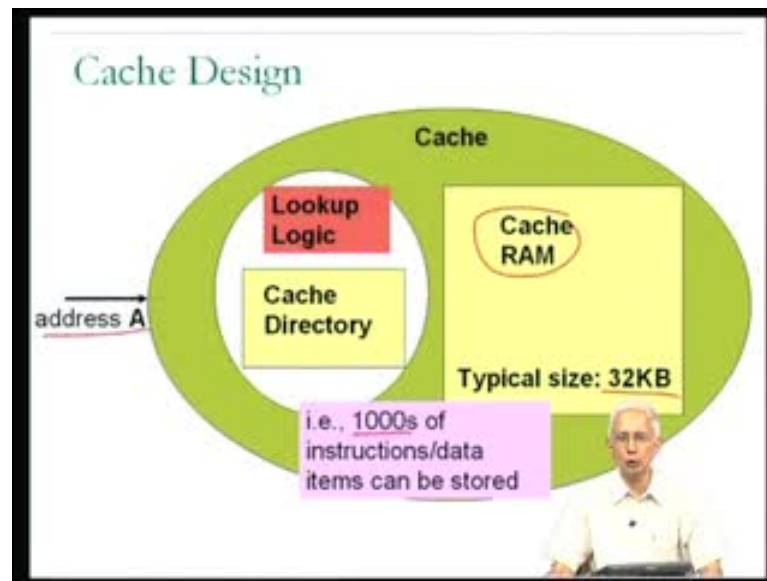
So, circuitries which can determine whether or not the address A is currently present inside the cache memory. In order to determine whether or not address A is currently present inside the cache memory, the cache will obviously have to keep track of the addresses which are currently in or represented inside the cache memory by a table of some kind. The technical terms for the fast memory, do I have it logic, and the table of addresses that I have are cache RAM, look up logic and cache directory.

The cache directory is a hardware table which contains information about the current contents of the cache. In other words, which memory locations are currently available in the cache RAM. The cache RAM is the fast memory in which those instructions or data are stored. The look up logic is the circuitry, which checks whether the address A is present by referring to the cache directory **and** subsequently can provide instructions of the data if it is present out of the cache RAM.

Now, I had given this indication that the cache is much smaller than main memory which suggests that, the amount of information that can be stored in the cache RAM is much smaller than the size of the main memory. A typical number for one of the kind of cache is we are going to look at, is as little as 32 Kilo Bytes. Remember that when we talk about main memories today, they would be a few GigaBytes in size which is 1000's of times more than the size of, I mean, what is its magnitude, but I definitely 1000's of times larger than the size of this cache RAM. So, this actually is of relevance to us and the discussion that is to follow.

We should note that if at any given point in time the cache RAM can contain only 32 kilobytes of instructions or data. Given that the typical **in our** MIPS 1 instruction set the size of one instruction is 4 bytes, then the number of instructions which could be contained in the cache is about 8000, 32 kilobytes divided by 4 bytes which is about 8000. And similarly, if I was storing integer's, 4 byte integers, in the cache then, the number of 4 byte integers which could be stored in the cache is a few thousand, about 8000, if the size of the cache RAM is 32 kilobytes.

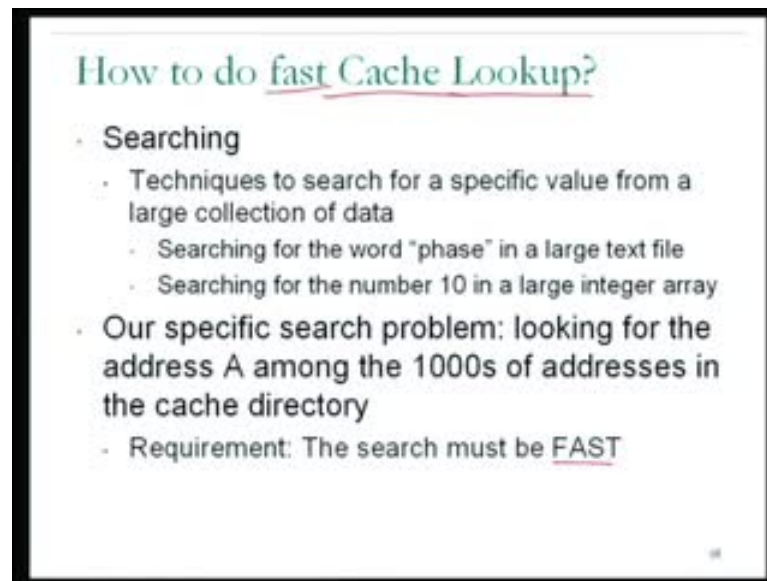
(Refer Slide Time: 03:26)



So in short... From this, we understand that the number of instructions or data that can be stored in such a cache is just a few thousands. In our current example, if I am talking about 4 byte instructions or data and the 32 kilobyte cache RAM then, about 8000 instructions or data, so numbers I describe as in the 1000's, not in the 100's of 1000's, not in the millions, but in the 1000's, a few thousands, which is something we will use in the discussion that follows.

Now, we are going to start by concentrating on what the look up logic does and how it might do it to give us an understanding of how the cache operates? Remember, the look up logic is the circuitry within the cache itself, which given in an address A determines whether or not, the entity represented by the address A is present inside the cache and it does so by looking up among the addresses inside the cache directory.

(Refer Slide Time: 07:50)



The question which we need to get an answer to right away is, how do a cache look up logic do the look up fast? It is critical that the look up be done fast, because we want the entire operation of sending an **address...**, the processor sends an address to the cache and subsequently the instruction or data goes back to the processor, we want all of that to happen in a very small amount of time. In our discussion of the processor, we were assuming that this would take about 1 clock cycle, in other words, may be a nanosecond or a fraction of nanosecond, as suppose to main memory which is going to take a 100 nanoseconds to provide a piece of data or instruction. Therefore, clearly the cache looks up itself much faster than the 1 nanosecond.

The question is how can a fast look up are done? Now, this whole question of looking up is actually a specific instance of more general problem called searching and one learns about searching in an introductory computer science course on data structures. Essentially, in searching one learns about different algorithms or techniques to search for a specific value from among a large collection of data. This is a frequently occurring problem both in the application development, as well as, we now see in hardware, because there will be situations where **searches...** particular element has been look for **in** a large collection of elements.

Now, a kind of example which might be used in data structures course, might be to talk about searching in the context **of...** let say, you have a very large text file and you want

to search for particular word let us say the word phase in the large text file and searching could be used for this, so this is the typical kind of an example.

Or if I have a large integer, I have written an application program in which there is very large array of integers, 10000 integers. I wish to determine whether or not the number 10, integer 10 is present in the integer array and if so, at which index, where in the array? So, these are two examples of search problem, which as you can clearly see will occur from many different kinds of applications.

Now, our specific search problem in the context of the cache look up, remember we are talking about how to do fast cache look up is, the address A has come to the cache and among all the addresses which are in the cache directory, the cache hardware has to determine whether or not the address A is present. From our calculation based on the typical size of the cache, we have determined that they could be a few thousands of addresses inside the cache directory not millions, not 100s of 1000s, but few thousands.

(Refer Slide Time: 10:53)

The slide is titled "Search Algorithms" and focuses on "Linear Search". It includes a diagram of an array with indices 0, 1, 2, 3, ..., n-1. A red arrow points to index 3, and another red arrow points to index n-1. The text describes the process of comparing a target address A with each element in the directory until a match is found or the end is reached. A red arrow points from the first step to the second. A small inset photo of a man is visible in the bottom right corner of the slide.

Search Algorithms

Linear Search

- Compare A with the first address in the cache directory
 - If they match, the search is successful
 - Else compare A with the second address in the directory
- If you reach the last address in the directory without finding a match, the search was unsuccessful
- Problem: Could take 1000s of comparisons

So, this is the size of the problem, a few thousands, in our particular cache example, it was about 8000 based on a current calculation, but remember that our requirement is that this search should have been extremely fast. So, we are trying to understand how search can be done quickly, and the requirement for speed comes from the fact that the entire

cache operation starting from look up and ending with the data reaching, the processor is supposed to take about a nanosecond about a clock cycle.

What are the different kinds of search algorithms that are conceivable another, the simplest search algorithms that you will hear about in, let us say, data structures course is something called linear search. In linear search, the picture that you should have in mind is one where there is, let us say, an array of integers as in the second example which I had suggested, searching for the number 10, the integer 10, in a large array of integers.

So, I have this large array of integers and the way that I could do a linear search is.., I am looking for the address A in this large array of integers. So, I compare A with the first address in this array, which in our context will be the cache directory. The index of the first address in the cache directory might be 0 as the convention that I will use is for arrays as in C, the index of the first element will be 0 and the index of the second element will be 1, and so on. So, I start by comparing A, the address I am looking for with the first element in the cache directory and if I am lucky they match.

If so the search is successful and I am done, and the search has been extremely fast, but if I am unlucky, they do not match, they are not the same and I have to continue. So, in linear search what I do next is, let compare A with the second address in the directory; in the other words, the 1 whose index is 1, and so on. So, I continue doing this until I either be successful in finding A within the directory, may be at location 35 alternatively is possible that I reach the end of the directory. In other words that I reach if the size of the directory is n elements then, the last element of the directory what I have an index of n minus 1.

If I reach the last address in the directory without having found a match along the way, then I know for a fact that the search was unsuccessful, in the sense that the address A is not present in the directory and that is the end of the search. So, the search either ends successfully or it ends unsuccessfully and either is ok, the cache hardware can be built to proceed depending on which of these is the case. So, this is the possible way to setup a search.

The problem with this particular technique, in other words linear search, is that if there are 1000's of addresses in the cache directory, in other words, in our example n minus 1

could be 8000 and something, 8195 or something like that. Then, in the worst case this number of comparisons that would have to be made in this procedure of a tutorial checking consecutive elements of the cache directory could be a few thousands and this would have to be done one after the other. Therefore, this cannot be viewed as a fast operation, cannot be done quickly therefore, one should eliminate linear search as a possibility for our context, the context of doing fast cache look up.

(Refer Slide Time: 13:45)

Search Algorithms

- 1 Linear Search
- 2 **Binary Search**
 - Sort the array of data items, say in increasing order
 - Compare A with the middle value
 - If they match, the search is successful
 - Else repeat for the appropriate half of the data
 - Much faster than linear search
 - Problem: Could take 10s of comparisons

The diagram shows an array of elements from index 0 to n-1. A red bracket highlights the middle element at index n/2. A red arrow points from the middle element to the right, indicating the search range for the next step.

Now, as an alternative to linear search, much faster search technique is something called binary search. The idea in binary search is that once again I have this array of values may cache directory let us say, but instead of just starting to search, I start by ensuring or sorting the array of data items, say in increasing order. If the array **of** already in sorted order then, I need not do anything about it, but otherwise I rearrange the elements of the array, so that they are in increasing order, let say, increasing from left to right which means that the element with index 0 is the smallest value and the element with index n minus 1 is the largest value. That is what means by sorting in increasing order, it could alternatively be sorted in decreasing order, it does not really matter.

Now, how do I proceed with the search for address A? I start by comparing A with not the first element, but with the middle element in the array and if n is odd, the middle element in the array will be n by 2, somewhere in the neighborhood of n by 2.

So, I compare A with the element at the middle of the array and if they are the same I am done, on the other hand, if they are not the same, then I could have determined during the check of comparing A with the middle value, whether A is less than or greater than the value at the middle of the array. If A is less than the element of the middle of the array then I know that my continued search for A can be restricted to the elements to the left or less than the element at the middle of the array.

On the other hand, **if** when I compare A with middle value, the value x, I found that A was greater than x, then I could restrict my search to the upper half of the array, thereby eliminating half of the array from consideration for subsequent comparisons. So, I could repeat the procedure that I have just described for the appropriate half of the array and what I mean by repeat is, let suppose, **that** I had determined that A was less than n by 2 at the element at n by 2, in other words x, then I could forget about all these elements, I just need to search among the other elements of the array and I could compare A with the element, which is at the middle of that half of the array and so on.

With each comparison I would be eliminating half of the elements that I have in contention for possible locations, where the address A could be. So, very clearly this is going to be faster than linear search, in the context of looking up, in the worst case, and not only that, I could say that it could take rather than 1000's of comparisons, it might just take 10s of comparisons. And technically, what I am talking about here is that it could take on the order of the logarithm of n base 2, which is in this case a few 10s of comparison.

(Refer Slide Time: 13:45)

Search Algorithms

- 1 Linear Search
- 2 **Binary Search**
 - Sort the array of data items, say in increasing order
 - Compare A with the middle value
 - If they match, the search is successful
 - Else repeat for the appropriate half of the data
 - Much faster than linear search
 - Problem: Could take 10s of comparisons

The diagram shows an array of cells with indices 0, n/2, and n-1. A red bracket highlights the middle element at index n/2, and a red arrow points to it from the right. The right half of the array is crossed out with red diagonal lines.

Remember that linear search was taking 1000's of comparisons which was definitely much too slow to be satisfactory for our hardware fast cache look up. Binary search is taking only 10s of comparisons which was 100 times faster, but requires that the addresses be in sorted order, which may be difficult for us to ensure, **you do not if** you know that is going to be possible, but the 10s of comparison themselves may not be fast enough for the context that we are in. Therefore, once again unfortunately, we may have a good technique much better than linear search, but not good enough for our fast cache look up.

(Refer Slide Time: 17:21)



Basically, what we need is some kind of a search technique that will be able to do the search in a number of comparisons which is not related to the number of elements in the array. Remember that it look like a linear search, in the worst case was going to take n number of comparisons whereas, the case of binary search it look like it was logarithmic in an n, but we want something which is not dependent on n, the number of elements in the array at all.

One such technique is something known as hash searching, the property of hash searching is that it may in fact, typically, just take 1 comparison for **the...**, typically not in the best case, but typically may just take 1 or 2, a very small number of comparisons independent of n. The number of comparisons required does not depend on the number of data values that we are searching among. So, this is clearly a much faster than the binary search technique and let me just explain how hash search works.

The idea of hash search is frequently use, so it is now available as a **verb** one talks about hashing and basically, hashing is a search technique that uses a hash table, it uses a table and it indexes into the table using a hash function. So, the specific idea is that there is something called a hash function and the value which was searching for will be computed on by the hash function to generate an index, which is use to look in to the hash table at one specific location of the hash table. So, the hash function is a function which is computed on the search string, the thing that you are searching for in the table.

(Refer Slide Time: 19:02)

Hash Table Example

- Example: Searching for the word "phase"
- Searching for a string of characters, $s_0 s_1 s_2 \dots s_{len-1}$
- Hash function: $\sum_{i=0}^{len-1} s_i \text{ div } len$

Handwritten notes: $len=5$, "phase" (with indices 0-4), ASCII

• But, "phase" and "shape" will hash to the same index value

• This is called a hash collision

0
1
255

"phase"

= ?

Let me just give you an example of how this operates. For this example, I will use the first search example that I had used earlier, where I am searching for a particular word, let say, I am searching for the word 'phase' as I had mentioned earlier. So, over here, I need to view the word phase not as the word, but let us say as a string of characters just to generalize this. So, in general, the elements which I am searching for strings of characters that, say, in general the string could be of length len.

So, len is the length of the string and I could talk about the individual characters of the string as being s_0, s_1 up to s_{len-1} . In the case of phase, this is a string of length 5 and I have the elements p which is the s_0 , which is s_1 and so on.

In general, if you are searching for a string of length n and you wanted to come up with a hash function, there many possible hash functions, I will just mention one particular hash function which is shown over here, you will notice that what I am doing is, I am first computing the sum of all the s_i 's.

So, s_i going from $i=0$ up to $i=len-1$; in other words, I just add up all the characters in the string what does it mean to add up characters? Characters do not have values that can easily be summed, if you view them as characters we do not think of adding the character a to the character b, but knowing that the characters are

likely to be represented in ASCII, we will recall the ASCII code, which is an 8 bit code, which is use for representing characters. We could actually do the summation of the characters by summing there ASCII values, **in** the ASCII values **are** can be viewed as unsigned integers. So, an unsigned integer addition will give us a sum and then we could divide this by the length of the string of characters.

In this particular example, I would add the ASCII value of p to the ASCII value of h etcetera, adding to the ASCII value of e and then, divide that by 5. Then, what I get would be in integer value, an unsigned integer value, and I would use this integer value to index into the hash table. So, the hash table itself is some kind of a table and array let us say, in which these different strings that I have come across will be stored.

For example, it is possible that the word 'phase' is present in my text that I am searching for in which case the word phase will be present in the hash table, where will it be present in the hash table? It will be present in the hash table entry corresponding to the value that I had computed by summing the elements as S of 0 p plus h plus a plus s plus e dividing by 5 and I would find it in that element of the hash table.

Now, over here, I have written that the minimum and maximum value in this **is** into the hash table might be 0 to 255. You will notice that this might be coming from the fact that I am assuming an 8 bit ASCII code, may be extended ASCII in which the values can therefore go from 0 to 255. Hence, when I add all the values and divide by the number of values, I will get a value which is between 0 and 255. Now, you can immediately see that if the value that I am looking for like phase is present inside the hash table.

(Refer Slide Time: 19:02)

Hash Table Example

- Example: Searching for the word "phase"
- Searching for a string of characters, $s_0s_1s_2...s_{len-1}$
- Hash function: $\sum_{i=0}^{len-1} s_i \text{ div } len$

0
1
255

"phase"

- But, "phase" and "shape" will hash to the same index value
- This is called a hash collision

Handwritten notes: len=5, ASCII, =?

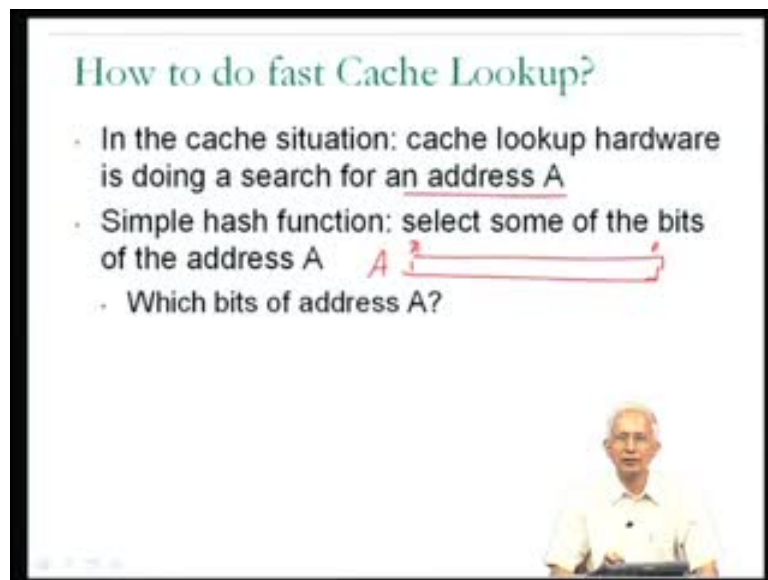
So that the very should to look at it is, if you are looking in a large text file, then before we start searching for words, you would take the individual words from the text file and put them into the hash table by computing, for each of the words, computing its hash value and then, putting that word into the corresponding element in the hash table. Of course, this would lead to a small problem in that... very clearly, if I have only 256 elements in the hash table, then I can only have 256 distinct words remembered in the hash table. This could be readily seen from the fact that, using the hash function which I have suggested both the word 'phase' and the word 'shape' will have the same hash index, because they have exactly the same characters in them just in a different order.

Therefore, when I sum the characters together, I will get the same sum and when I divide by 5, I will get the same value. Therefore, phase and shape, and in fact, a large number of other words which may have completely different alphabets, may all hash to the same entry in the hash table. This could be a problem and this is in fact, what is called the hash table collision and the implementation of the hash table would have to take care of collision by ensuring that alternative locations in the hash table could be used or some such mechanism. But in our context of the cache directory that may not be an issue in terms of the look up speed, we may be able to handle collisions or collisions may not be a problem to us as far as the look up speed is concerned. Therefore, from the perspective of speed of look up, we must view the hash table as really being such technique which

will return yes or no, with the single comparison, just comparing the word that you are looking for with...

So, you take the word that you are looking for, you compute its hash function you then, index into the hash table and compare the word that you are looking for with the word inside the hash table. If they are the same, with 1 comparison I get successful, and if they were not the same then, with 1 comparison I would say not successful, if I assume that there are no collisions. Therefore, in some sense this is the fastest possible search technique and therefore, this must be the one that is use for fast hardware look up which is in our context.

(Refer Slide Time: 24:50)



We now understand that in order to do fast cache look up, the cache hardware must be using a hash function. Now, a hash functions on what? In our cache situation, the cache look up hardware is doing a search for an address A and therefore, the hash function must be computed on the address A.

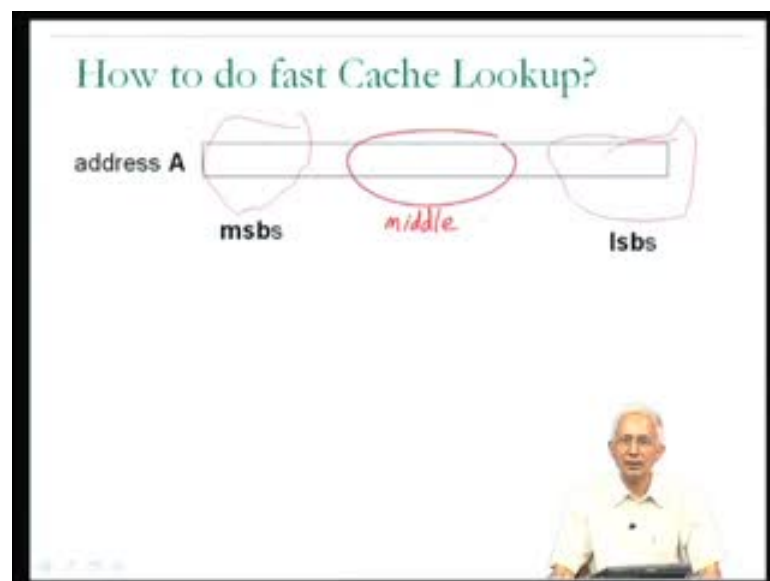
Now, a simple hash function would just be to select some of the bits of the address A, rather than trying to add all of the bits of address A, if you add all the bits of address A, you will end up if the 32 bit address you will end up with a value which is the sum of the 1's or the number of 1's in the address in that might not be a very meaningful hash function.

So, rather than that it might be interesting to think of a hash function which computes a value based on just selecting some of the bits of the address A. The picture we could have in mind is, so I have the address A, I am showing you the address A in binary, if the address A is a 1000 then, we think of it as the binary representation of 1000.

We are talking about an address A which is sent from the processor to the cache and is therefore going to be sent in binary form. So, we think of the address A in its binary form least significant bits it is most significant bit if the 32 bit address, the bits would go from 0 to 31.

The question now is, in doing the fast cache look up using a hash function, which bits of the address could be used to do the fast cache look up, in other words which bits of the address define the hash function.

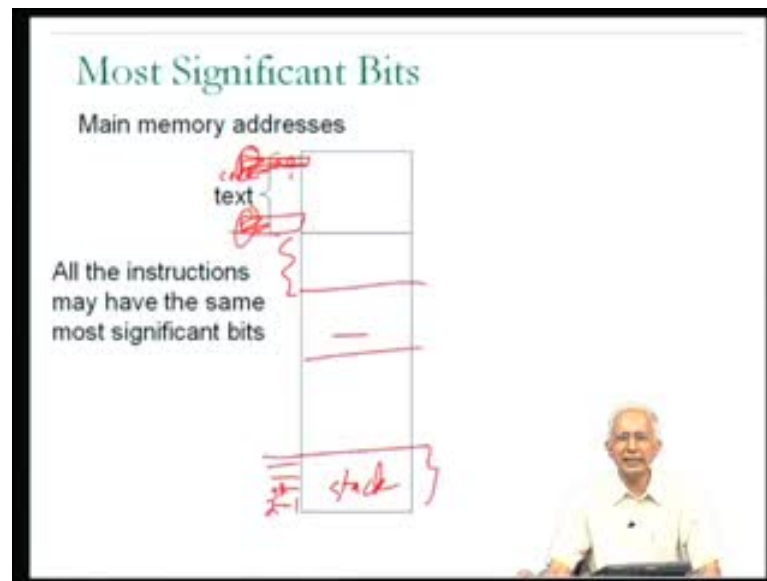
(Refer Slide Time: 26:28)



Now, there are a few possibilities, one possibilities **is** that I could use the most significant bits of the address A and other possibilities that I could use least significant bits of the address A and of course, the third possibility is that I could use some of the intermediate bits of address A.

Let us consider the two extreme possibilities first; the first suggestion is that I could use the most significant bits of address A in order to do the look up into the cache directory.

(Refer Slide Time: 26:59)



Let us try to think about to what extent this is a good idea. Now, consider your typical program, so over here what I have drawn is, I have drawn your program and it is addresses go from 0 to some maximum value may be 2 to the power of 32 minus 1 some maximum value.

Now, just consider the text part of your program, **we understood that** these are virtual addresses we are talking about. We understood that the text occurs from the **whether** we were drawing the memory image of a process, we understood that each process assumes that it starts with address 0 goes up to some maximum address 2 power 32 minus 1 . The text instructions, the code, the program part of this images occupies the early addresses, in other words the program occupies addresses $1, 2$ etcetera.

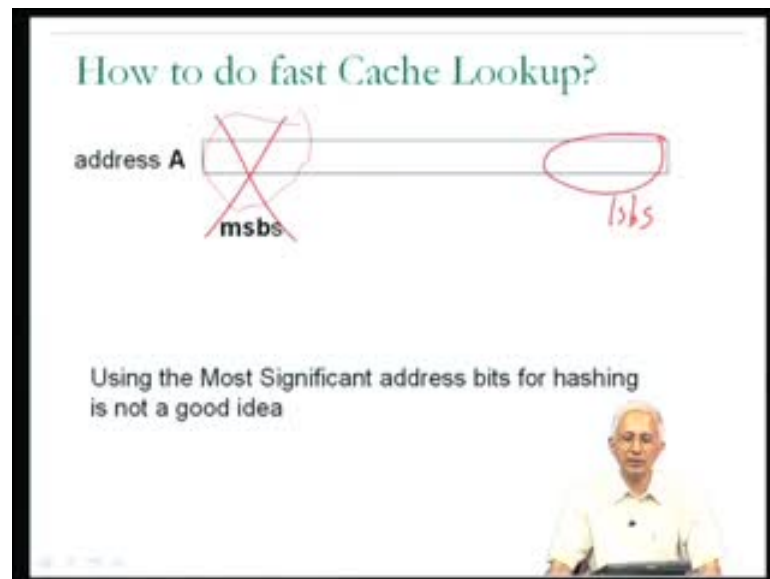
Now, if this is the case and I have a medium size program then very clearly all of the instructions of my program will have the same most significant address bits. If I consider the two most significant addresses bits, then you should notice that everything of the first quarter of this address space will have the same two more significant address bits.

Similarly, if my program is not 32 kilobyte 64 kilobytes or something like that, it will be the case that almost all of the instructions will have the same most significant address bits.

The different instructions will differ in the least significant bits, but they will have most significant bits which are pretty much the same, very minor difference between the most significant bits depending on the size of the program.

What this is going to mean is that from the perspective of instructions, pretty much all the instructions will have the same hash function, if I use the most significant address bits, has the hash function. What is happening? It means that almost all the instructions will end up occupying the same entry in the cache directory, which means that there is going to be a lot of situations which we call collisions, which means that the cache cannot be too successful in its operation as far as instructions are concerned. You can stretch the same argument for the stack. Once again, the bulk of the stack is going to have the same address bits over for the data, over for the heap; in other words, with in any region of memory, the different entities are going to differ very little in the most significant address bits.

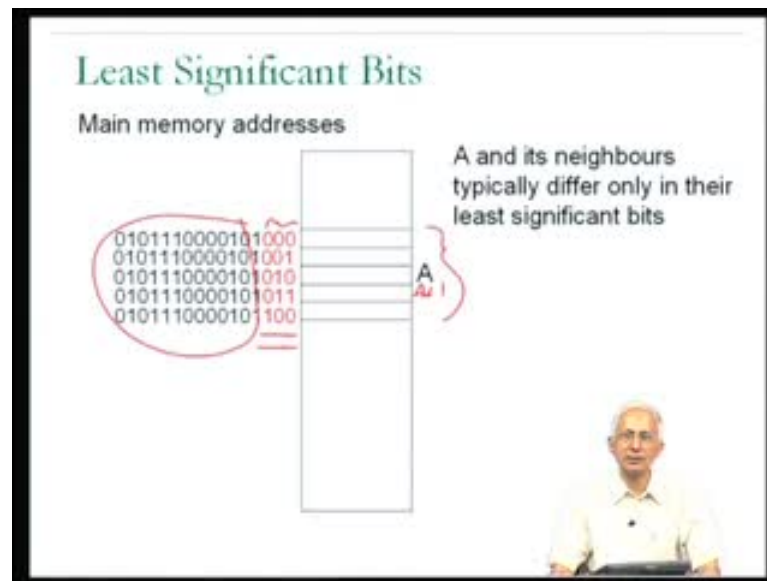
(Refer Slide Time: 29:24)



Therefore, using the most significant address bits does not seem like a good idea in the special case of small programs, it would be the case that pretty much everything would index into the same place in the hash table, they would have the same hash function. Therefore, it is not a good idea to use the most significant bits for the fast cache look up.

Second alternative that we were going to take into account was using the least significant bits of the address. Now, let us think about this a little bit, the ideas that **we will** rather than using the most significant bits, if eliminated this as being a good idea, but rather we could use some of the least significant bits of the address, this is the address A we were talking about to index into the cache directory, in order to do the look up to determine if it is in the cache or not. Now, let us think about the least significant bits a little bit.

(Refer Slide Time: 30:15)



Once, we will look at a picture of the memory image and let us consider a **particular...** this is the address A that I am concerned about, and I know that address A has some address, but this could be the address I am just putting down some, this is not 32 address it is only a 16 bit address, and in red I have shown you the least significant bits of the address A.

Now, **we know that** from the perspective of spatial locality of reference, we know that in connection with address a some of the addresses which are important in terms of likely addresses which you reference in near future are the neighbors of A. What are the neighbors of A? The neighbors of A are the memory locations which are A minus 1, A minus 2, A plus 1, A plus 2 etcetera. If I look at the addresses of the neighborhood of A or the neighbors of A, I have put them down over here. You notice that the addresses of the neighbors of A are actually similar to the address of A in the most significant bits and in fact, they differ only in the least significant bits. If I just look at the least significant

bits of A and its neighbors, I notice that they are all different, what does this mean? This means that if I had used the least significant bits of the address A to look into the cache and then next, I was to use a least significant bits of address A plus 1 to look into the cache and so on, because address A plus 1 is likely to be reference soon after A, then I would find that each of these objects index into a different cache location.

Now, in general, we understand that from the perspective of spatial locality of reference, we really want to treat A and its neighbors as 1 entity, because if there is spatial locality of reference then, we want them to come in as whole into memory. In the case of paging we talked about A and its neighbor should be coming into the main memory together. That is why we had this notion of a page, the same concept should hold in the case of the cache. We do want to treat A and its neighbors A and A minus 1, A plus 1, A minus 2, A plus 2 etcetera, as a single entity, but if you use a least significant bits of an address to do the look up then we are treating A and its neighbors as different entities most definitely, because they will occupy different locations in the look up table.

(Refer Slide Time: 32:42)

How to do fast Cache Lookup?

address A

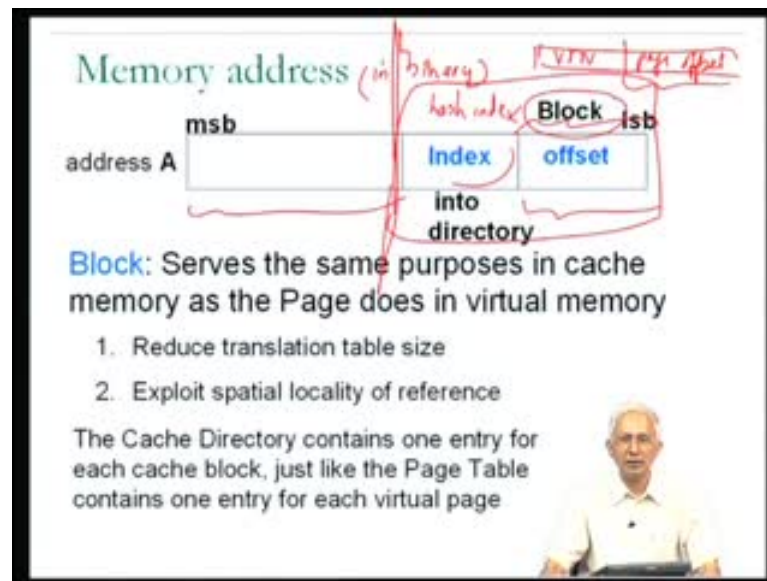
msbs middle lsbs

A and its neighbours possibly differ only in these bits; but they should be treated as one unit, not hashing into different hash table entries

Using the Least Significant address bits for hashing is not a good idea

Therefore, once again this does not look like a good idea, because if I use a least significant A and its neighbors typically differ may be only in the least significant bits in the consequence would be that **they will not** they would hash into different hash table entities and this would not be a good idea.

(Refer Slide Time: 33:09)



In a sense, we considered using the most significant bits of the address, we consider using the least significant bits of the address, rather **them** was any good. What we are left with is we have to use bits from elsewhere; in other words, some of the intermediate bits of the address for the fast cache lookup as the hash function. In other words, when we think of how the cache hardware looks at a memory address. If I consider the address A, once again, I am showing you the address A in binary, then I am showing you the address a binary with a least significant bit of the address shown on the right and the most significant bit of the address shown on the left.

Then, from the perspective of how the cache is going to look at it, the cache is going to use some of the intermediate bits to index into the cache directory, because the intermediate bits define the hash index, which is why I refer them as the index bits. They defined the hash function which is going to be used to look into the cache directory. What we are left with is the most significant bits and the bits which are less significant than the bits that we use for the computation of the index. If we just **to** look only at this region of the diagram, this will remind you of what we had in the case of paging, so for the moment just ignore the region of the diagram to the left of this line.

If you look at the region of the diagram to the right of the line this looks very much like what we had in the case of paging, wherein address was viewed as a virtual page number and what we call a page offset and this looks very much same. The virtual page number

was used to index into a page table; here, the index that we have over here is going to be used index into the cache directory. Therefore, the remaining bits are plain very much the same role as the page offset bits were playing in the case of paging.

Since, I need to use a different term from page, I will use the term block in the context of caches and I will refer to the lesser significant bits, in other words the bits which are less significant than the bits which are used for indexing as formulating a block offset, where the term block is being used for a concept like the page, but in the context of caches. So, I formally introduce the word block as a concept in caches, which seems to serve the same purpose as the page did in virtual memory. In some sense, it is providing the exploitation of spatial locality of reference, in addition to that it is providing a mechanism by which I need not maintain the translation information for each address, but rather you can be maintained a single piece of translation information can be maintained for a large number, a region, contiguous block of addresses.

These are the two objectives that the block is going to serve in the case of caching, the same objectives that are served by the page in the case of paging; it reduces the translation table size. In other words, it reduces the size of the cache directory, rather than having to have one cache directory entry for each byte in terms of addresses, I need to have one cache directory entry only for each block. Similarly, it causes exploitation of spatial locality of reference, because very clearly the cache is going to be organizing in terms of blocks not in terms of individual bytes.

(Refer Slide Time: 33:09)

Memory address (in memory) [VFN] [page offset]

msb hash index **Block** [isb]


address A [] **index** **offset**

Block into directory

Block: Serves the same purposes in cache memory as the Page does in virtual memory

1. Reduce translation table size
2. Exploit spatial locality of reference

The Cache Directory contains one entry for each cache block, just like the Page Table contains one entry for each virtual page



The bottom line is we now figure out that from our understanding of how the hashing is going to happen that the cache is likely to be organized with the cache directory having one entry for each cache block, just like the page table contains one entry for each virtual page. We are going to find out that there is a great deal of similarity between the operation of the cache and the operation of virtual memory. The time that we spent in understanding virtual memory will be a benefit to us and getting a quicker understanding of how cache is operate.

(Refer Slide Time: 36:52)

Summing up


- A cache is organized in terms of **blocks**, memory locations that share the same address bits other than lsbs
- Main memory is also organized in terms of blocks
- The cache hardware views an address as

[] [] []

tag **index** **offset**

block

cache into directory



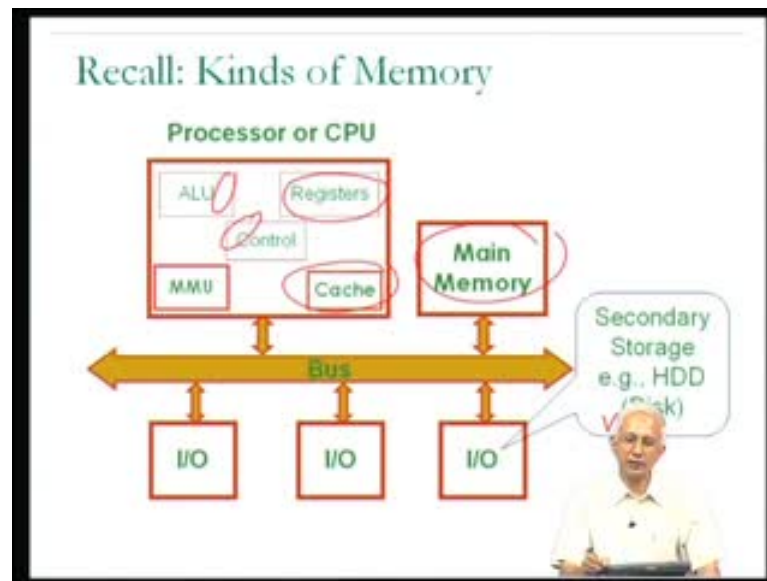
Just we sum up what we seen up to now, we figure out that from this discussion about how the fast look up is done, it let us do this conclusion that a cache is organize in terms of blocks and the blocks are basically memory locations that share the same address bits other than the least significant bits. So, they are memory locations which differ only in the least significant bits, in some sense defining a neighborhood around the middle address of the block.

Secondly, from our understanding of how virtual memory operated, we knew that the concept of pages was useful for the virtual address space, but the consequence was that main memory as well as virtual memory both had to be organized in terms of pages. The consequence in terms of caches now is going to be that if we assume that caches are organized in terms of blocks, since the data or instructions are going to come into the cache out of main memory, it must be the case that main memory also is organized in terms of blocks.

In other words, in continuing or summing up, we figure out that the cache is organized in terms of blocks not in terms of individual bytes, but in terms of blocks. Main memory also is organized in terms of blocks not in terms of individual bytes, but in terms of blocks. Finally, we understand how the cache hardware views an address, it views an address as actually being made up of the some of the intermediate bits which are used as look up value, which defined the hash value for indexing into the cache directory.

What I mean by directory here is the cache directory, do I have it table which is present inside the cache, the least significant bits formulate a block offset, they tell you within a particular block, which particular byte or word is actually required by the processor and then there are the most significant bits, I will talk about that shortly. But given for a particular cache, then we should give details about the cache, the most significant bits **we will** for the moment refer to as the tag, but for a particular cache if I give you details about the cache, you will be able to calculate how many bits are present in the block offset. In other words, how many address bits are used as block offset and how many address bits are used as index into the directory and as consequence, you will be able to calculate how many address bits are used as the tag whatever the tag is used from.

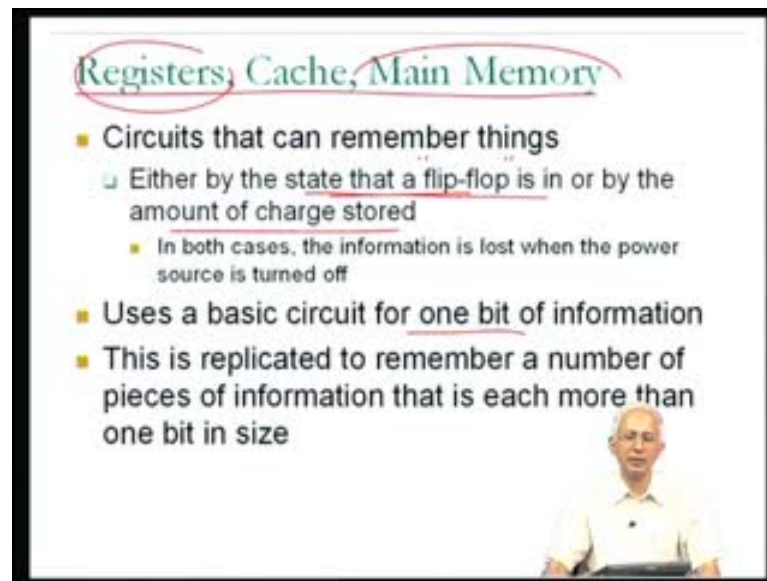
(Refer Slide Time: 39:19)



Now, moving right along, I just wanted to remind you about something that we had seen briefly earlier about the different kinds of memory present in a computer system. Now, in our high level diagram about organization of the computer system, there was the registers which were a form of memory; **inside ALU there was more special purpose registers, I am sorry** inside the control there was more registers, even inside the ALU there was some special purpose registers such as ALU out conceivably.

Then, the cache is a form of memory, main memory is a form of memory, in addition to that many of the IO devices are a form of memory such as the hard disk over VCD's, DVD's, and so on. And I won't say much more about the IO device memory which are often refer to a secondary storage, once again I will defer discussion of that for until a later lecture.

(Refer Slide Time: 40:17)



Registers, Cache, Main Memory

- Circuits that can remember things
 - Either by the state that a flip-flop is in or by the amount of charge stored
 - In both cases, the information is lost when the power source is turned off
- Uses a basic circuit for one bit of information
- This is replicated to remember a number of pieces of information that is each more than one bit in size

But, what we have seen about the different kinds of memory that I have sub listed in other words, registers, cache, and main memory is that these are all made out of circuits that can remember things. They made out of electrical circuits, digital circuits that are capable of remembering things and the way that the circuits remember is either by the state of a circuit, some of the circuits are known as flip-flops.

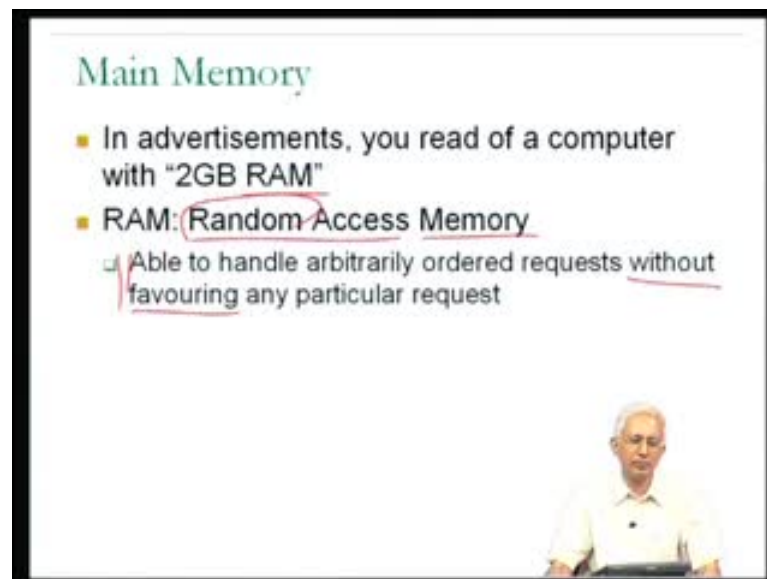
So, people talk about the state that a flip-flop is in; flip-flop is a name of a particular kind of circuit capable of remembering. Alternatively, the circuit might remember by the amount of charge stored in a capacitor. So, the different kinds of circuits they could be used in registers, cache and main memory, but in both cases, the information that is stored will be lost when the power source is turned off.

In other words, when you turn of the computer, the contents of the registers, the contents of the cache, and the contents of the main memory are all lost, why? Because the circuits that are used in the registers, the cache and the main memory all depend on their being power available. And they keep on remembering either by the state a circuit is in or the amount of charge stored and when the power is turned off, the charge dissipates and the state that the circuit **is in** disappears because a machine is in a non-powered upstate and therefore, the information is lost.

This is of course not the case with the hard disk, DVD and so on; unfortunately, but for the moment we understand that these are the kinds of circuits which information is lost when power is turned off.

And basically, the registers cache in main memory will all be made of potentially different kind of circuits, but the different kinds of circuits will all contain a sub circuit or a very simple version which is capable of storing 1 bit of information. And then, that kind of circuit will be replicated a large number of times in order to create adequate storage for the requirement. For example, if I am talking about one 32 bit register, then they might be a simple circuit which can be used as a 1 bit register and I have 32 of them together which forms a 32 bit register. Similarly in the case of main memory, the scale would be much larger, they could be Gigabytes of information that have to be stored and therefore, much larger replication of the simple circuit which could remember 1 bit.

(Refer Slide Time: 42:40)



Now, one term which is used in connection with main memory and which I have also use in this class is to talk about RAM, R A M. I used in this every lecture because, when I talked about, when I replaced, when I give you the block diagram of cache and I started by saying cache has fast memory, but then when I went to a technical mode, I replaced the word fast memory by cache RAM, R A M and I never told what R A M was.

Further, those who you have read advertisements are for computers you may have come across the term RAM there. For example, an advertisement which says that, this particular personal computer is provisioned with 2 Gigabytes of RAM, what does the RAM stands for? RAM actually is for Random Access Memory. So, it is a form of memory and the form of memory has a property of something called random access and let me just give you a rough idea, what it means to be random access. Essentially, what random access memory is a kind of memory which is able to handle arbitrarily ordered requests without favoring any particular request.

In other words, as far as RAM is concerned, each memory access is equivalent to each other memory access; **memory accesses are not in any sense** - one memory access is in no sense preferred over in other memory access, in terms of getting preferential treatment, **right**. So, that essentially what RAM could mean, but we just understand that when one hears about RAM, it is a form of circuitry which remembers and has the property that this property in some sense. And we understand that the memory inside the cache is likely to be of this kind as well.

(Refer Slide Time: 44:26)

Memory Hierarchy

- CPU registers
 - few in number (typically 16/32/128)
 - subcycle access time (nsec)
- Cache memory
 - on-chip memory
 - 10's of KBytes (to a few MBytes)
 - access time of a few cycles (1 cycle (nsec))
- Main memory
 - 100's of MBytes storage (to a few GBytes)
 - access time several 10's of cycles
- Secondary storage (like disk)
 - 100's of GBytes storage (to a few TBytes)
 - access time of msec

Now, I would like to introduce one more term before getting into the nitty-gritty details of cache. This term is memory hierarchy because, we have seen many different kinds of memory in our quick look back at computer organization starting from the CPU registers to the cache memory to the main memory and even the secondary storage like disk or

DVD and very clearly, they all form some form of storage within the computer system, they can all be used to store information within the computer system. And I am shortly going to explain why I use the word hierarchy, but let us just refresh our memories about how these different forms of memories differ from each other.

Now, one way in which they differ from each other; well, let us just spell out the properties of each of these, what do we know about CPU registers? We know that there are typically a small number of CPU registers; for example, in the MIPS 1 instructions set, we found out that they were 32 integer registers and 32 general purpose registers.

So, the number is on the order of 16 or 32 or 128 a few 10s or few, at most a few 100s of CPU registers, but did not know that they could be accessed in a very small amount of time. We used our definition of the clock cycle based on how fast one could access a CPU register and the assumption was that a CPU register could be accessed in much less than a clock cycle, which is why I talk about sub cycle access time, it takes less than 1 clock cycle to access a register.

So, it is on the order of less than a nanosecond. Now, we have not learned a lot about cache memory, but we do know from the whether I can draw things and from comments I made in the previous lecture that the cache memory has to be viewed as being part of the processor. So, if there is the processor **is** on 1 integrate circuit chip, then the cache memory is lightly to be on the same integrated circuit chip, it is integral with the processor.

I have already mentioned that the typical size of the cache is something like 32 Kilobytes. So, let just generalize that to a few 10 of Kilobytes and in the extreme case, they may be caches with a few Megabytes, but this is a reasonable generalization. Just as I generalize, CPU registers as being a few 10s in numbers, I could talk about the cache memory as being of size a few 10s of Kilobytes; 32 Kilobyte is a number which we have in mind as a typical cache.

And we have this understanding that, the access time of a cache memory is likely to be may be a cycle. Until now our assumption was that the access time of cache memory was 1 cycle; I would now extended that little bit, it might be one more than 1 cycle, it could

be 2 cycles or something like that, but we talking about something which once again is on that nanosecond time scale, may be 1 nanosecond, may be 2 nanoseconds.

Registers were a fraction, a part of a nanosecond. What we know about main memory? We know that main memory is much larger, we could be talking about a few Gigabytes, 2 Gigabyte main memory in the advertisement that I refer to in the previous slide, may be fortunate to have 4 Gigabytes of main memory in your PC at home or those may be less fortunate, I have only half a Gigabyte 512 Megabytes of main memory and from what we seen the access time is several 10s of cycles, may be 100 nanoseconds are there about.

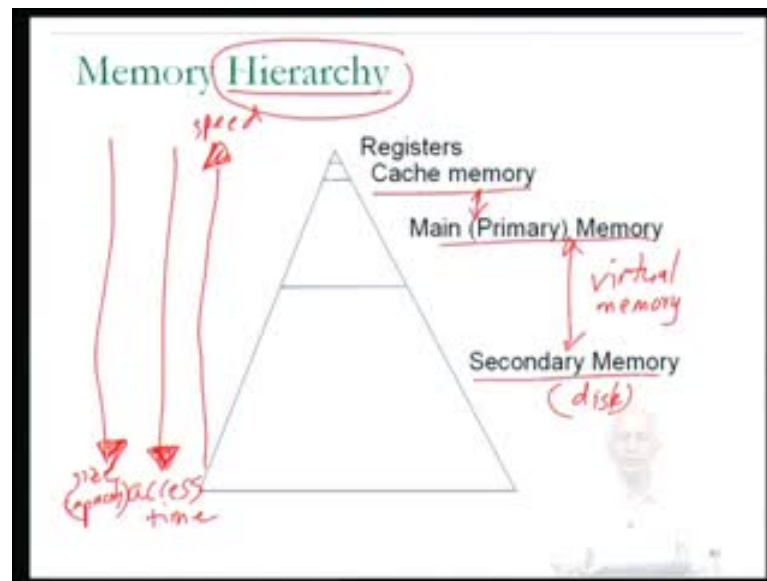
(Refer Slide Time: 44:26)

Memory Hierarchy

- CPU registers
 - few in number (typically 16/32/128)
 - subcycle access time (nsec)
- Cache memory
 - on-chip memory
 - 10's of KBytes (to a few MBytes)
 - access time of a few cycles (1 cycle (nsec))
- Main memory
 - 100's of MBytes storage (to a few GBytes)
 - access time several 10's of cycles
- Secondary storage (like disk)
 - 100's of GBytes storage (to a few TBytes)
 - access time of msec

Finally, secondary storage like disk we do know that, I have talked about disks which could have 100s of gigabytes or even a few terabytes of capacity. And that the access time is not on the nanosecond time's scale at all, but in fact, on the millisecond time scale.

(Refer Slide Time: 48:30)



So, these are numbers that we have encountered until now, I am making only minor adjustments to that, the kind of minor judgment is that, they may be caches which take more than 1 cycle to be accessed, other than that it is pretty much the same numbers that we have been talking about before.

Now, if I had to actually represent this information on a single diagram, the way to represent that diagram might be in the form of a triangle. The general idea here is that, if I wanted to represent **both** all of registers, cache memory and secondary storage devices on a single diagram, I would give very little important to registers because, there the registers provide so little storage they may only be 32 registers which might be able to contain only 4 bytes of information.

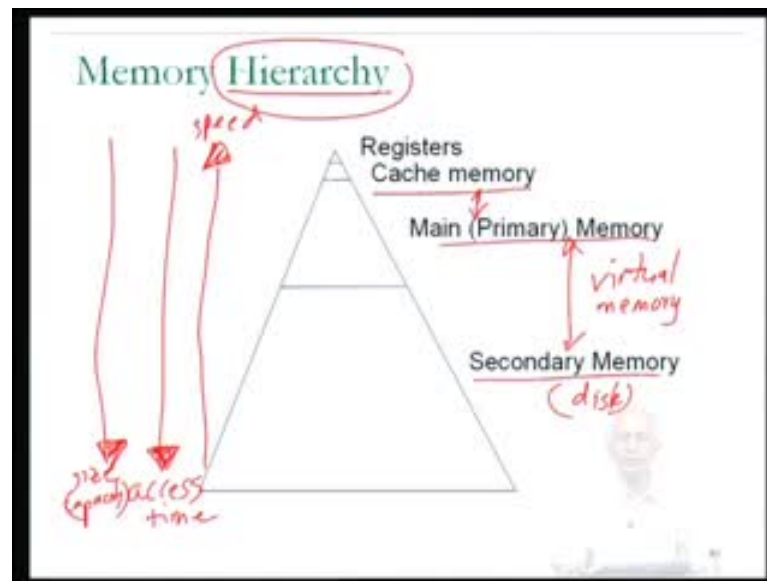
That is just 128 bytes of information as oppose the cache, which can store 32 Kilobytes of information or the main memory, which can store 4 Gigabytes of information or the disk which can store 1 Terabyte of information. So, the registers occupy very small amount of the area of this triangle. The main memory being that Gigabyte capacity occupies a reasonable amount the area whereas, the secondary storage occupies the bulk of the area and if one had to scale this properly, the line for main memory would be much further up, but this will give us a little bit of clarity in the description.

Now, what are we talking about in this diagram, it looks like the area occupy relates to the capacity of the device. But, if you look at the diagram from another perspective, we realize that, at the top we have the registers which can be accessed in **sub nano sub cycle** in between, there is a main memory which takes 10s of cycles, at the bottom there is the secondary memory which takes essentially 1000s of cycles. So, in some sense, if I had to draw a line over here, which takes about the amount of time that it takes to access something from that kind of memory, then the access time increases as I go down **the down** this triangle, registers are very fast, memories are less fast, secondary memory is least fast. Therefore, the access time is increasing; by the same token the capacity increases the size or capacity, as I go down this diagram that is why the diagram serves a purpose.

Now, given the diagram, we can sort of fit the cache memory into this diagram, we would put the cache memory somewhere over there; towards the top in that, it has much faster access time the main memory, but marginally slower than registers and it has larger capacity then the registers, but much smaller capacity than the main memory. Hence, this is the kind of diagram we would end up with, a diagram which gives us this idea about the differences between the capacities and the speeds or access; if I had to draw arrow for speed, the arrow for speed would be in this direction increasing speed as I go up the hierarchy, registers are the fastest, cache memory is the second fastest and so, on.

So, this kind of diagram is useful, it is still not clear as to why it is labeled with the word hierarchy, but very clearly this kind of a diagram will be useful for us in understanding the interplay between the different components of different kinds of memory in a system and we know that there is significant interplay. Recall that when we talked about virtual memory, we realize that we were talking about the use of main memory, but since main memory was not began off to whole the virtual address spaces of all the processes, which might be running. We had to resort to secondary memory to actually store the virtual address spaces of all the processes and then at any given point in time, some number of pages from those virtual address spaces would be present in the main memory.

(Refer Slide Time: 48:30)



So, there was interplay between main memory and secondary memory from the perspective of virtual memory. We are now talking about cache memory and we realize that what the cache memory is containing in its cache RAM is, some blocks out of the many blocks that are present in the main memory and not all the contents of the main memory can be present in cache memory at a given point in time, given that the cache memory is so much smaller than the main memory.

So, once again we are talking about interplay between the cache memory and the main memory in terms of what happens when the processor initiates a request to access something out of memory. The request could be made to the cache memory, the cache memory might provide the data or instruction, alternatively it may be determined that the data or instructions is not in the cache, in which case it will have to be fetched from the main memory into the cache memory, before it can be given to the processor from there.

And this is similar to the relationship that we saw between the main memory and the disk. When the processor generates a request and it went to the main memory, the way that we discussed it when we are talking about virtual memory, if it was determined that there was a page fault, then the request had to be satisfied like copying the page from the disk into the main memory before the data could be accessed in main memory and provided to the processor.

So, the interplays between these different levels of memory present in a computers system is clearly integral to understanding what happens when our program executes. We do need to understand how the hierarchy work first and we proceed to do this in the next lecture.

So, I will stop here for today. In today's lecture, you will recall that we have started looking at the operation of cache memory; cache memory is an integral part of any computer system as we had found out the basis for our understanding of how a processor works. Without the presence of cache memory, our understanding of processors were conceivably have be quite different, because any time an instruction or a data had to be fetch from main memory, they would be in (()) it delays and therefore, the cache memory is an integral part of the processor.

In today's lecture, we have looked at how cache memories even though they are small, can be built to do very fast look up or in fact, as a consequence of being small, can be built to do very fast look up, in order to determine which of the few main memory blocks are actually present in the cache. We proceed to look at other intercrises of cache hardware in the lectures to come, thank you.