Parallel Computer Architecture

Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

Week - 02
Lecture - 12

Lec 12: Cache Basics

Hello everyone. We are starting module 2 today and the topic is memory hierarchy. Today's lecture is on cache basics. Most of you must have done a basic undergrad level course on computer organization and architecture so you are familiar with it, but we can treat this as a recap lecture about cache and its concepts because as part of this subject we are going to look more at cache and memories and hence we need to recap the concepts before we can plunge into detailed topics. Alright. So, what is memory hierarchy? From the CPU to the disk, we all know that we start from the registers, then we have several levels of cache, then the main memory and then the disk.

And why does this hierarchy exist? Because the memory which is closer to the processor is smaller and faster whereas the memory which is farther is slower to access and because it is slow to access we would want to have a larger storage also available to it. Hence in any memory hierarchy, the registers are the fastest. So this slide is showing some values related to servers and at the bottom part of the diagram we have personal mobile device related parameters. So, registers are the fastest, they store very less amount of information, but very fast.

Then you have the first level cache which can be accessed within less than nanoseconds moving from L1 to L2 which is the second level cache and then to the third level cache, we remain in the range of nanoseconds here. Once we go to the main memory which could be DRAM or any other type of technology, so the main memory references are slower and further the disk storage is even slower. But as you move away from the processor, the size increases. Okay. So, CPU if you move far from it the size increases and reverse direction the cost increases. So, the closer you come to the CPU the cost of the memory increases and the speed increases.

So, it is fast and it is costly and the farther you go it is slow and bigger in size. Okay. So that is the memory hierarchy. So, let us look at the concept of why this memory hierarchy came into place and how does it actually work. Okay. So, I will take very familiar example of our day to day processing. So, where do we store our money? Okay. Forget the e- wallets.

Because you go 5 to 6 years back you only deal with cash. Okay. So, where do you store your money? You store it in a bank. That is the biggest storage of money you ever have apart from the gold storage which you might have, but this is the place where you are going to store a big amount of cash. So, the your money is in the bank every time you want to buy something, do you go to the bank to draw cash? You do not. Where do you go for drawing cash? You go to the ATM machine. Okay.

So, if I just try to give some numbers suppose I want 10K rupees I will go to the bank and if I want 1K I might go to the ATM machine, but every day when you go to work or to college you want to buy some items right? from a vending machine or from a tea vendor you want to spend 10 or 20 rupees. So, are you going to go to the ATM machine to draw that? Because you would go there only to draw some little bigger amount. So how do you pay for a day to day requirements which are of small denomination? You pay it from your wallet. Right? So, this is the wallet in which you are going to keep small amount of cash to use for everyday purposes and then so I will go to the bank if I want to draw more amount of money I will go to the ATM and if I just use similar analogy I will say that for the amount of 10 rupees or something I will use my wallet. Okay. So, when I am using my wallet for 10 rupees does it mean that I only have 10 rupees? No, I may have 10,000 or I may have lot of money in the bank, but right now I do not need it.

What I need is small amount of money so that I have in my wallet. Tomorrow if you want to do some bigger transaction you want to pay fees of your college or you want to buy something bigger you will want to go to the ATM for drawing some cash or if your wallet is empty you might go to the ATM to draw the cash and depending on the requirement you will go to a storage from where you will obtain the information. Okay. I will extend the analogy to a different scenario. Suppose you are given a assignment to write a term paper and for that you visited a library, you are sitting at the library desk, you want to go to the shelves to issue some books related to the topic. So, you go and you issue there is a shelf in the library you will go and issue one book at a time.

So, you brought one book you kept it you started reading you took some notes, then you realized you want some more information. So, you will again go to the shelf, bring another notebook or another book and eventually you will have a pile of books on your desk which you will use to write your term paper. Okay. So, if you suddenly feel that these four books are not sufficient I want some more information, you will again go to the shelf and when you go to the shelf for accessing a new subtopic in your term paper you will see a book and maybe the close by books are on related topic by some different authors. So, you will think, okay author A's book is on my required topic, but maybe author's B books may also have some relevant information. So, let me take these two books with me because it might help me writing this particular subsection. Okay.

So what you tend to do when you go to the shelf because you have to walk from your desk all the way to the shelf you want to save your time. So, when you go there you tend to bring close by books because they may be useful to you. So, you bring multiple books with you. Now when you bring those books to your desk your desk is already full. So, you want to remove books from your desk which you do not need now.

Why? Because you finished writing about them. So, you will remove some books from your desk and you will replace them with other books which you want to use right now. Okay. So now moving this concept to the concept of locality. So what is locality used in memory hierarchy? We have two concepts, locality of reference which is spatial locality and temporal locality. In this example temporal locality was the book which you have brought right now will be needed on and off again, right? until these four books which you have brought you might be using them one after the other again and again while you finish that subsection of your term paper. Okay.

So, that is temporal locality. Then the concept of spatial locality is when you went to the shelf to bring a book you brought two or three more books which were close by because libraries tend to stack them closer to each other on a given topic. So when you went you saw two three books on the same topic you brought all three of them with you because you thought it will be useful these three books may be useful to me. So that is spatial locality, in space you found information useful and temporal is time wise locality that is you will need that information again and again. So the concept of memory hierarchy derives from this analogy of writing a term paper you are sitting at your desk you bring the most needed books from the shelf and you go and fetch newer books when you finish a subsection and then when you go to the shelf you tend to bring more than one book because it might be useful. Okay.

So the most needed book satisfies the temporal locality and the more books which you bring satisfies the spatial locality. So with this term paper example the whole library is accessible to you but you are not going to consume that information in one go you are going to consume it one by one. So this concept gives us an illusion that a large amount of memory or a large amount of bookshelves are available to me it gives me an illusion of that but at the same time I am able to utilize them in small amount in a faster manner. Okay. So moving on this principle of locality states that programs access relatively small portion of the address space because when you write a C program you have arrays, loops even the instructions which we write they will be executed one after the other. So within the program we see lot of locality. Loops essentially say that the same type of instruction gets executed again and again giving you temporal locality. Loops also give you spatial locality because you access array A of I where I keeps changing so you are going to access consecutive locations in the memory giving you again spatial locality. Okay.

So arrays, loops and even the instructions which execute in a sequential manner give you spatial locality because close by instructions get executed one after the other. So within the context of a program we do have temporal and spatial locality. In the memory hierarchy we have multiple levels of memory as we discussed in the first slide each has each hierarchy level having a different speed and a different size. The faster memory is kept closer to the processor because yes the processor needs data very quickly so the fastest is closest to the processor and if anything is fast it is very efficient it is very expensive at the same time. Okay. So the faster memory is expensive compared to the slower memory. Okay.

So because it is expensive we tend to keep it small in size. Okay. So what is the overall goal of the memory hierarchy? To present the user with so much memory that it is the biggest amount of capacity available at the cheapest technology and at the same time which gives you speed which is equal to the fastest memory. So I want speed of a fastest memory and I want the capacity of the cheapest memory. Okay. So that is the illusion we want to create overall in the memory hierarchy. Okay. So when we say hierarchy we have multiple levels and then there are levels we want to bring data from one level to the other level.

So we have one level, second level, third level which is even bigger. So the data does not get transferred across levels, we only transfer data from one level to the other level. So from here to here, from here to here. So between two levels the data gets normally transferred between two adjacent levels. Okay. And the level which is closer to the processor, the closest one is called the upper level and the others are called lower level.

The upper level as you see every upper level is a subset of the others. Okay. So every upper level is a subset of the next level which is further away from it. And essentially the complete amount of data is stored in the lowest level could be main memory or the disk and so on. Okay. So the memory hierarchy makes sure that data are subset of each other and they get transferred between two adjacent levels. Okay. So some terminologies to remember here because we have small levels of memory like this, the smallest one is closest to the processor you may or may not have everything in every information there.

So when we go to read or access a data item we should be sure whether it is present or absent. Hence we need to understand these terms. The first one is called the hit that, is the requested data found in the upper level? If you find it, it is called the hit. If you do not find it is called a miss.

Then we have the concept of a hit rate. Hit rate or hit ratio is the fraction of memory accesses which are found in a particular level. So if you send 100 accesses to the memory and out of that 60 were found you will say my hit rate is 60 percent. And miss rate is 1 minus hit rate which is the fraction of accesses which were not found in the current level which you are accessing. So that is hit rate and miss rate.

Next is hit time. So hit time is the time to access the upper level. So you want to access the upper level, first go read it and then identify whether it is a hit or a miss. So there are two things to be done here. First access and identify hit or a miss. So this whole thing is called a hit time.

And the second one is miss penalty. This happens when there is a miss. So if you did not find the data item in that level you have to go to the next level and bring that data from that level, store it in the upper level and at the same time move that data to the processor or the previous upper level. Okay. So go bring the data store it here and give it to the requester. So this whole trip is called miss penalty. Okay.

So we have to replace the block in the upper level. So go to the lower level, bring the data and deliver this block to the processor. So this is similar to going to the shelf and bringing a new book. Then memory as we said is very critical for performance because most of the programs spend their time accessing memory. All things are centered around the memory because depending on the memory performance the OS is going to manage the memory and the IO the compiler is also going to generate the code depending on how good your memory is and it will eventually affect the applications. Right?

So the way the memory hierarchy or the way the memory system is built it directly affects the performance of the system. So it is very critical to the performance. Okay. So therefore all the computer designers what do they need to do? They need to pay attention to how can we improve the performance of memory system. So paying attention to memory systems is very important. Okay. So moving on in the levels of the memory hierarchy the first level was after the registers we had the cache.

So what is a cache? It is the highest and the first level of the memory hierarchy encountered once the address leaves the processor. Okay. So once the address leaves the processor it encounters the cache. Cache is the smallest and fastest memory between the processor and the main memory of processor and other levels of cache. So first level cache is like that and blocks are brought from the main memory into the cache. So we have main memory which is a big memory and if I assume a single level cache which is sitting here, so data is brought into this and then accessed by the processor. Okay.

So this is the cache. So it is the smallest fastest memory closest to the processor, brings data from the main memory. Even here because it is small we need to take care of hit or miss we need to know whether the data is present or not present and this also works on the principle of temporal locality and spatial locality as we discussed because the same item may be needed again and again and nearby items may also be needed in near future. Okay. So what is cache? It is nothing but a sequence of storage elements. So here on the left hand side I have shown some array or some storage locations which has randomly loaded some information. Right?

These are some addresses. This is an address of which the data gets loaded in this location and so on. Next you want to access address X_n and to access that X_n we go to the main memory, bring the data and maybe write it here because this was a free location. Right? So there was a free space here and hence we added X_n to that location. So this is how we bring data and keep it in the cache. Okay. So seeing more details about the cache design, I will take an example.

Suppose we will take a small cache which can hold only 4 entries. Okay. And it is associated with a very large main memory. Right. So when the program starts reading information, the processor sends some request to load the cache and it goes to the main memory, brings the data. So let me write some alphabets here. So this is the data I am interested in. So this data item comes here, sits in this location, then the B sits here, C and D.

So this is how the cache will get loaded with A, B, C, D because the processor accessed A to D in that sequence. Next it wants to access E, F, G, H something and now this E, F,G,,H are say sitting here. This is address 000 to 4 and let me assume that this is address 8 onwards. Okay. So that is the binary address of these locations. So when I brought A, it came from memory address 0.

Next I want to bring E, F, G, H. Now E has to come and replace somebody here. So we will see these details later but suppose E comes and sits here and F, G and H. So these are going to remove the A, B, C, D and themselves sit in these locations. So later on if we come to the cache and find out, is A present? If I ask you this question, is the data item A present in the cache? So how will you answer that? Because initially it was there when E, F, G, H were read after that A got deleted. Okay.

So if you come here to identify that we need to know the address of that. So when I say is A present? essentially I am saying is the address or is the content of address 0000 present or not. So when I load this cache with that I also need to associate the address of the values. So when A came in, I said that this belongs to the address 0000 and this and so on. So if I ask you is A present? you will come and see this label.

So I am going to check this label. If this label is all 0s then yes A is present but what would happen is when I loaded E, it replaced this and the label became 1000. Okay. So when you came for looking for A, you will never find four 0s but you found this and you will say that the data is not present. So if the label associated with that box, this is the box in which I kept the information, there is a label associated, we are going to compare this label to identify whether that data is there or not. And why do we have to do this? Because here I have only four locations which might map to any number say 200 locations or 2k locations in the main memory.

So only four of these can come here at a time. So which four have come has to be identified and hence this label is very important. Okay. So how do we identify where should A sit? and if I say bring F, so how will you know to keep F in the first box or the second box or the third box? So which box in the cache should an address go? So given an address, you need to identify which box within the cache is it destined for. So you simply take that address and modulo that with the number of boxes in the cache. Right? So if you mod that with the number of blocks, so this is the formula, take the block address, modulo the number of blocks in the cache, you will get the value. Okay, suppose I want to load the address D into the cache and D has the main memory address as 0011 and I will do a mod 4 with this and what do you get? You get the value decimal 3 and so D gets stored in decimal 3.

So this is 0011. So this is the location number 3 in which value D will go and sit. So every time you get an address, you do a modulo with the number of blocks and identify which box that value will go into. Okay, so the next question is does the block have correct values? So how will you identify whether the information you have is present or not present? In the previous example, when we came to the cache looking for A, right, we came here to find out if A was present, we wanted to compare the label because the size of the cache was small, hence we kept a label which was keeping track of the source address of A. Okay, so this label is nothing but a tag which we associate with the entry. Now how do you identify the size of this label? So this label in my previous example was 4-bit label because I was keeping the complete address, but every time you did not keep the complete address, we can actually not use certain bits because if I am using 4 blocks of cache, then to identify which block in the cache, I am already consuming the LSB side bits, right.

So this will decide the block number. So the block number is decided by some part of the address which will always map to some block. Hence, the part of the address which is not part of the block number is what I am interested in. So I need not keep a label which is this long, okay, the address could be that long. So this information is redundant, we can remove that information and only a part of that big address I will store as the label which is called the tag.

So we will do examples to understand how to derive this. For example, if you have a 5-bit address, then and you have 8 entries in the cache, you are going to use 3 bits to identify the block number that is which box of the cache you are going to go and hence 5-3 that is 2 bits are going to be stored as the label or the tag of the block, okay. Next is even if the tag or the label matches, how do you make sure that you have valid information? That is initially when we started the computer, the cache was empty and it was logically empty, but physically it had some storage. Suppose it may have A, B, C, D written here, but this is not what the program wrote, this was pre-existing in the cache when you started the system and probably the tag which is the label, it also had some values written here, may or may not be useful to us. So when if I say is A present in this? you will come here and yeah, A is present, the tag is also matching, should I treat this as a hit? This is the first time we have come to read A and we do not know whether the program has really read A or not because the system has just started. So when you start the system, the cache may have garbage values preloaded inside that and this may or may not match with your intended data and if it matches again, it is not valid because you did not load that value.

Hence to identify garbage from real data, we add another bit to the cache called the valid bit. So this valid bit is when you start the system, we reset all these valid bits so that no accidental hits can happen inside the cache. So all these valid bits are made 0 and when the program itself loads block A, this valid bit will be turned to 1. So that is the use of a valid bit. Okay. Because when you start the system, your cache may have garbage tag values, garbage data values and if it matches, it is still not useful to us because the program did not load it. Hence to identify that we add a valid bit which is set to 1 when the program does a read or a write to the cache.

When the program loads a value into the cache, the valid bit is set accordingly. Okay. So you would also need the valid bit when there is a big cache but only 4 entries are filled, right? sorry only 2 entries out of these 4, we have filled this and this entry but the other 2 entries we have not yet filled. So for those, the valid bits will remain 0. Okay. So the valid bit is useful to identify the blocks which are really present and updated by the program and all the others will have the V bit equal to 0. Okay.

So we will quickly do an example to see how we will fill the cache. In this table, you can see on the left hand side is the index, index is the block number. This is the block number, V is the valid bit, T is the, this is the tag which is the label and as we discussed, this tag is a part of that address, I am not going to keep the complete address and what is this going to do? We are going to load these many memory locations inside this. Okay. So here, the tag is a 2 bit tag and index is a 3 bit index because we have 8 entries. Okay. And every entry will store 1 byte. So if I look at this address, the last 3 bits of an address out of these 5, this will be used as the index field to go into the cache and these 2 fields will be used for tag. Okay.

MSB side 2 bits for the tag and LSB side 3 bits for the index. So let us start filling this, I will do a few and you can pause the video and do it for yourself and then check the answer. So I will start with an empty cache and the cache is empty and hence the valid bits are all 0, if you can see it here. Okay, we will start with the first address. Take the first address and here the last 3 bits are the index, so 110. So this is the row in which this value will go and in this location, the remaining 2 bits are the tag.

So under the tag, I am going to write 10 here, sorry, let me erase this. So the tag is 10 and here I am going to, whatever is the data, so I will just say the address loaded, the actual address I have loaded from is 10110. So the data in this field has come from this particular address. Suppose I directly jump to a new address, suppose I am loading this, you can fill the others for yourself, then here the index is the last 3 bits, so this row, the tag is 10, okay and the address I have loaded here is 1 followed by 4 0s, okay, right. If I take something else, say this one, so 010, so we will go to this location, write the tag as 10, the address I have loaded here is 10010 and so on.

And you will clearly understand that once I have an index of 010 is my index, the tag field here in my example was 10. But if there is any other address, can you generate an address which will go in the same row but have a different tag, okay. So which all addresses will go in this row which have the same, sorry, which have the same index but a different tag, okay. So if you solve this, what you will understand that your index is same and these two bits will be different. So can you count it? So here the tag is on the MSB, so tag will change, tag will have 00, 01, 10 and 1 1 as the values followed by the index which is 010, 010 here, 010, this and this, okay.

So these are the four different addresses which will all be eligible to sit in this particular location. I hope that is clear. And the same thing is depicted here. So this type of a cache is called a direct map cache because every address has exactly one location in the cache. The grey ones, these grey boxes or grey addresses map to this location here, 001 index and the orange ones map to 101.

So multiple orange boxes map to a single orange box in the cache and multiple grey boxes also map to a single grey box in the cache which we saw here. Multiple such addresses were mapping to the same entry in the cache. So that is the concept of a direct map cache where every address has exactly one position in the cache, okay. So now we will extend the design. In the previous design we assumed that there was just one entry and every entry just stored one bit of information like here if I go back, here, I had just stored a single alphabet A, single alphabet B.

But this cache entry which we had drawn, I had, so if this is the cache. Each entry need not be a single alphabet or a single letter, it can be multiple letters. So I can have a complete spelling or a word written here, right. So instead of single alphabets, here I have written 4 to 5 alphabets in every box. So now my entry, here if I want to read A, I need one particular address.

I want to read R, I need a different address. So within every block, within every block you have multiple bytes of information. Initially I had just one byte because we simply wrote alphabet A. In this example, I have 3 to 5 bytes of information. I can have a 3 byte word or I can have a 5 byte word, right, a small word or a big word. So we have multiple values which can go inside every box and hence the address also needs to be extended accordingly.

So in the previous example, we had a tag, we had the index and that is all because there was just one alphabet sitting there. A tag of 00 and the index of this, this was all we had. But right now, when with this I will only reach say with this index I cannot reach for here I have the index as 11, okay. So with 0011, suppose I come with this information, I am going to use this as index, go to this box where I have written the word water.

But to read the word water, there are 5 bytes written inside it. Suppose I want to read the alphabet T, so I also need the address for this alphabet T. So essentially inside that I need the byte address also. So which byte I want to read inside this box. Initially we had just one byte. Now when I go for multiple bytes, there is a tag, there is an index and after the index we also have multiple bytes to read.

We can also complicate this by saying that I not only have a single word, but every entry will have a collection of words. So I will put all letters, all words starting with A here, okay. And then another one, all the words starting with B kept here. And within this collection of words, if I want to access the word apple, this is the second word.

So I need to know which word to read within this. So apart from the bytes in a single word, I also need to know which word to read, okay. So I need to know which word first and once you read the word, within the word, which byte or which alphabet of the word. So two more levels of information will get appended to the address. So we will take an example to clarify this. Okay, so here I am using a main memory of 16 bytes, cache of 8 bytes, but the block size is 2 bytes.

So instead of 1 byte, I am keeping 2 bytes of information here. So let us find out the tag index and the byte offset. Okay, so what is the address size? Because of my memory is 16 bytes, the address is 4 bytes. So we have 1, 2, so 4 bits, 3, 4, this is the 4 bit address. Within this 4 bit address, every block is going to have 2 bytes.

So if this is my cache block, inside this I will write two alphabets A1 and A2. These are my two alphabets covering 2 bytes of information and they will be stored in the memory one below the other, this will be stored in the address 0000 and this is 0001. They will be stored in consecutive locations. So each alphabet has got a different address and I will need to know which? this last bit is going to decide that. So this is going to tell me which byte I want to read from these 2 bytes. So I have A1 and A2 as 2 bytes, from these 2 bytes which byte I am going to read is decided by the last bit. Okay.

How many blocks do I have in the cache? How many such boxes I have? We have 8 byte cache, so 8 bytes can be kept. In each box I am going to store 2 bytes. So I am effectively having 4 such boxes available in my cache. So 4 boxes means I need 2 bits to store the information. Right? So these 2 bits will identify the box, which box I am going to use to store the data and the remaining one is the extra information which I will call the tag. Okay.

So what is the answer for this? Tag is 1 bit, index is 2 bits and byte offset is 1 bit. So this is how we can calculate. There will be formulas for this but as an illustration, I hope you it's clear that within the box if there are multiple bytes of information you need the byte offset. Clarifying it further, the same example is here, the same memory which we discussed here 16 bytes memory, 8 byte cache. Okay. So 16 bytes memory is in the grey picture on the left hand side, all 0s to all 1s, so four 0s to four 1s is the main memory. On the right hand side, you can see this is the data portion, the green is the data portion and the red or the orange one is the extra information.

Extra information is just telling what set id that is the block number which is also called the set in the cache, whether the block is valid or not and the tag bit. Initially we loaded the address all 0s, so I loaded this. Once I load this because the cache has got 2 bytes of information, it is going to bring B also along with it. So even if I read A, it comes with B, so 2 bytes get brought in at the same time, address 0000 and 0001 both come in and if you know the index, index was the middle 2 bits.

So middle 2 bits of this, this is the index and hence they sit in the set number 00. The block is valid because we have just brought it in and what is the tag the MSB which is 0, getting it. Alright, so next I want to load 0010 that is C and D. So you can try it for yourself. When I go and read C, the D also comes with it.

So you can pause the video and check and try to fill this cache and later also try to fill K and L. Okay. So you try to solve this and at the end, your cache will look something like this. Alright, so we filled consecutive locations. Next what I want is, I want to load a new address 1110. 1110

It is here at the bottom. In 1110, this is the index, middle two 11. So we will first check whether this is there.

So we will go to the set 0. Okay. Set is this. Is the data valid? Yes, it is valid. Now let us check the tag. Is the tag matching? What is the tag here? Here the tag is equal to 1 and here the tag is equal to 0. So they do not match. Hence, this is a cache miss. So address 1110 is not present in the cache because you cannot look at G and H to see whether this is the data you want. You have to compare the orange portion, the index, the valid bit and the tag to confirm whether you have the data or not.

Alright. So once we have a cache miss, we need to bring the data from the memory. So now when I go for bringing 1110, it comes with the second byte also because we are loading 2 bytes at a time. So where will they go? They will go in the same location. But what changes? Valid bit remains as it is. This tag bit will become 1 because the current address's tag is 1 and the G and H get replaced with P and Q.

Okay. So this is how it will change. Okay. So as an exercise, I won't go into details of this but leave it for you to try it out. Here I have drawn the same cache in a horizontal manner. This green portion, it was vertical. Now I have just spread it out in a landscape manner where block are in this direction, block 0, 1, 2, 3 and the addresses are in the right hand corner.

Then we have the set numbers. Valid bit and others I have omitted here because we will take it as intuitive information. The previous example, we tried to load this. So 1110, it wasn't there because the tags did not match.

Hence it was a cache miss. Once the cache miss occurs, we are going to load this information. The new blocks got loaded. And we can continue this example with newer addresses.

So I will just do one of these and then remaining I will leave it for you to try. Okay. So I want to load 1100. So first identify the set ID. The set ID is the middle two bits 10. So this is the set I am interested in. What is the tag of the requested address? It is equal to 1.

And if you look at this set, this is the current information here and here. And the tag of this doesn't match because the address loaded here has a tag of 0. So this is an abstract diagram because you infer the tag. I have already written the addresses in these cells.

So the data is what is stored at this address. So 0100, this is the address of the data I have kept here. And the data I want to bring is from 1100. So the tags don't match. Okay, so if I say my next request is for 1100, I will say that this is a cache miss.

It is a cache miss. And where will this miss get satisfied and loaded? It will go in set 10. I will write 1100 here because this is the address which will now sit in this box and 1101. Both of them will come and sit here. So you can continue this hand simulation and declare hits or misses. So if you do we have 1000, maybe even this will need to be loaded because there is no entry with the tag equal to 1.

So this will also result into a cache miss. Later on when you come for this one, 0011, it will result into a hit. So for this you will get a hit. This was a cache miss, this was a cache miss, this was a cache hit and so on. Okay, so you try it for yourself to get a feel of how to fill the cache using the index and the tag and decide the hit or miss. Okay. So one small point left, we have looked at multi-byte block size, that is every block was storing here.

Every block was storing 2 bytes of information. Okay. But it was still one entry. Can I have multiple words in the box? Instead of one word, I have 1, 2, 3, 4. Okay. As I had given an example of air, apple, etc written inside that box. So multiple words can go.

Okay, so we extend the cache design from 1 byte information to 1 word. So 1 byte is now translated to 1 word. And suppose every word is 4 bytes long, I further extend the design and say that I have M such words, each of 4 bytes, getting it? So I have these are M words in the cache box or block and each word is and 1 word is equal to 4 bytes long. Okay, and of course, how many such boxes do we have? How many such boxes do we have? We have N such boxes.

So we have N boxes or I can say N blocks. So the cache has got N blocks. Each block has got M words. Every word is 4 bytes long. So so much information. Now how do we distribute the address across this and how do we identify the tag and the index? Okay, so if let us go bottom up. Let's look at the word.

The last LSB side is the word. After the word, I have the number of words, right? So in the word, I have two things. One is look inside the word. And here, how many words do I have? So inside the word, I have 4 bytes of information. How many words do I have ? M such words Okay, I have M words each of 4 bytes. So if we translate it, how many bits do we need to represent this? So for 4 bytes, how many bits will you need? To identify at byte level, okay, not at bit level.

There are 4 bytes, so I need 2 bits for this. So 2 bits for identifying which byte I am going to use. So this is which byte, okay. These four, 2 bits will tell you which byte to read. Then for these M words, how many bits will you need? $log\ m$, right? So $log\ m$ number of bits will tell which word.

The last 2 bits say which byte. Once we come out of this, then we go to the box level. We have n boxes or n blocks. For this, we will need n bits to identify which blocks to go. Getting it? And now once you have covered all the blocks, whatever is remaining in the address remains your tag. If I take this as a longish address, the 2 bits to identify which byte, the next $log\ m$ bits to identify which word, which byte, which word, the next $log\ n$ bits to identify which block and whatever is remaining is the, this is the tag. So when you say which block, this is nothing but the index field. So if I have an address of size A, so it is A, A is equal to this, so how many bits will be there in the tag? It will be total address bits A minus all of these $log\ n$, $log\ m$ and 2. Okay.

So we have completed that example and here if you see, as we discussed in the previous slide, we have the tag bits is the address bits minus the index, the $log\ m$ and the 2. So this way we can have a bigger cache designed. Thank you so much.