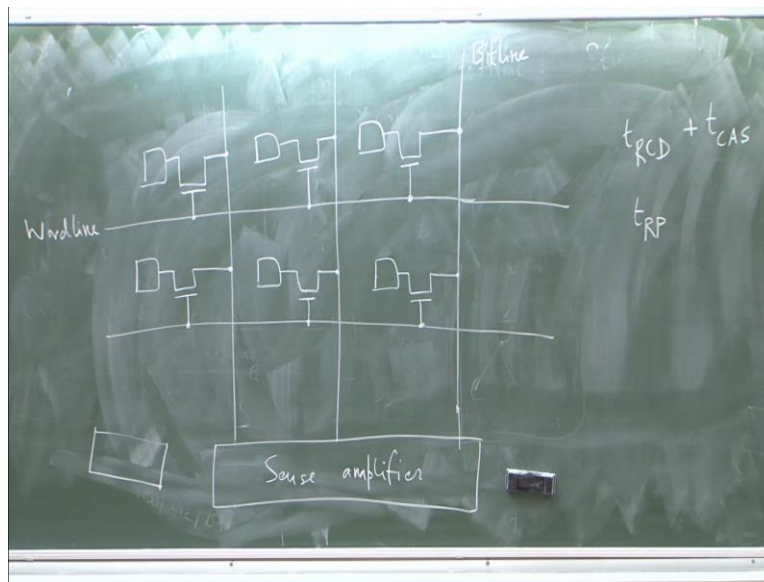


Computer Architecture
Prof. Mainak Chaudhuri
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

Lecture - 30
Topics in memory system, DRAM and SRAM Technology

(Refer Slide Time: 00:14)



Here is the architecture. ((Refer Time: 01:04)) So, we have multiple such rows, multiple such columns. So, this is called word line, this is called bit line, ok, and these are called ((Refer Time: 01:22)) transistors. So, the idea was that, so just, so the way DRAM is accessed is, that equals the access of full row and we have an array of sense amplifier here.

So, you access the full row ((Refer Time: 01:53)) into the sense amplifiers and then, you can access the ((Refer Time: 01:57)). So, after you sent the row address, which we essentially, essentially will be decoded and will activate all of the word lines, ok. And from then on, until you can access the column, it is essentially called the row to column ((Refer Time: 02:17)) or t_{RCD} . And then, time to access the respective, the required columns will take time t_{CAS} , right. So, that is what we mentioned, ok.

And you also said, that as long as you have, so it also maintains a register here, each essentially records the currently accessed row, alright. So, that is the open row. So, whenever request comes what we will do is you compare its row with the currently registered row here. If they match, then they do not really need to activate any word line. You can just directly do and do the CAS from the sense amplifier, ok, alright. So, you can eliminate this particular ((Refer Time: 03:02)) case. Yes or ok, that is what we mentioned last time.

So, so, that is how you enjoy first open row accesses to the DRAM array. The other thing that will happen is, that the currently, currently access row does not match with the current open row. In which case what we have said was, that you need to close this row and open a new row, right, ok. So, what really happens is, that you, you precharge these bit lines, ok, at that time. So, that, so that takes some extra time call, t_{RP} , that is, row precharge. So, the question is why is it really needed this precharge operation?

So, this sense amplifiers, actually what they are doing is, that they are trying to, so whenever you activate a word line the charge of the bit line is supposed to change, ok, by whatever amount. And the sense amplifier's job is to magnify the change on the right side, ((Refer Time: 04:04)), ok. So, precharge proportion what it does is, that precharge, it is all the bit lines to the midpoint of 0 and 1, alright. So, that is the stable state of the DRAM where the DRAM is ready to erect. So, that is what this precharge ((Refer Time: 04:17)) actually does.

And then, when you activate a word line, depending on the charge stored here in this particular cell, that will disturb this particular charge at ((Refer Time: 04:29)) because this is a capacitance actually, storing the charge. So, very small capacitance, it will have a very small charge. So, from the midpoint the charge will, the charge ((Refer Time: 04:39)) swing a little bit on the higher side or little bit on the lower side depending on the charge stored here ((Refer Time: 04:45)). Sense amplifier's job is to magnify that to the full voltage, either to 1 or to 0, ok, that is what it will actually do this. So, precharge formation is actually readying the DRAM array for the next both access, ok, otherwise you cannot even sense the voltage from the bit line.

So, that was, that was about the DRAM details, how you access and what the ((Refer Time:

05:09)), etcetera. So, we spent a lot of time last time discussing these things. Any question?
Yes...

Student: Sir, right also ((Refer Time: 05:16)).

Professor: Yes, right also happens the same way. You force certain charge on the bit lines and that it transferred to the, to the, to the capacitance here, alright. So, this is a single transistor cell. That is what I am showing here. So, there are many other ((Refer Time: 05:34)), there, there is a three transistor cell also where we normally has separate word line for ((Refer Time: 05:41)). So, like this, this is normally used as the most dense device.

The major architecture actually goes here. The most of the engineering, how you really design the cells so that it can retain the charge longer because if it, if it can retain the charge longer, you can ((Refer Time: 05:55)) the refresh cycle. Refreshing, as we discussed last time, essentially involves letting out each row with the sense amplifier and closing that row so that takes time.
Yes...

Student: Sir, why do we need to have the sense amplifier? Is it because the voltage change on the bit line was small?

Professor: Yes exactly, exactly.

Student: But what is if we have, Sir that will be because the capacitance in these cells is small.

Professor: Exactly.

Student: So, why cannot we have larger capacitance?

Professor: So, they may lose density. So, last time we mentioned that for DRAM, the primary goal is density. So, how many bits can it pack in, in the ((Refer Time: 06:34)).

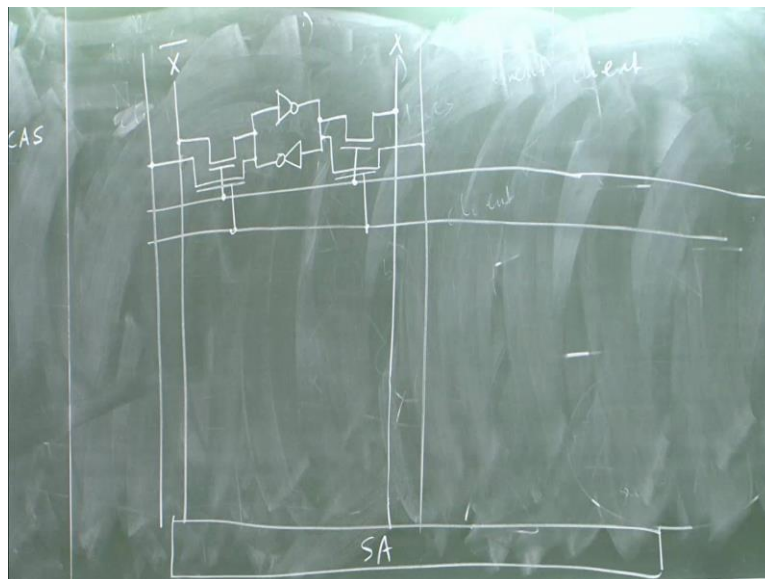
Student: Larger capacitance...

Professor: Exactly, yes, exactly

So, we will see one more thing when you go to SRAM, we will quickly discuss right now, actually. ((Refer Time: 06:43)) actually has for each column two bit lines, the pair of bit lines. One is, one, one carries the actual value, the other carries the inverted value. So, that actually makes your sense amplification much faster. Here, it is, it actually takes little bit time to amplify the voltage, but DRAM is not much concerned about ((Refer Time: 07:03)), density what matters.

So, if you make a pair of bit lines, density will even go down because it will take a pair of bit lines. You cannot pack as many bits in ((Refer Time: 07:12)). Any other question over here? Ok, alright, so what I will do is I will spend a little bit time explaining SRAM.

((Refer Slide Time: 07:34))



So, they look very similar in architecture. So, these cells are, maybe I can draw it separately

there. So, this one you know, ((Refer Time: 07:38)) inverters give you 1 bit memory cell, right. ((Refer Time: 07:41)) in a digital design course because that is how flip flops were actually designed, ok. So, this is a typical SRAM cell.

And then, you have the ((Refer Time: 07:56)) transistors and the bit lines and the word lines and it is exactly same. And this is, replicate it on this side and this side, so ((Refer Time: 08:16)) SRAM, alright. And of course, we have the ((Refer Time: 08:20)) for, so here you can easily argue out, that one of these lines I can, if I call x , other one will be \bar{x} ((Refer Time: 08:40)). So, here what happens is, that the precharge operation, precharge, which is both the lines to 1, so therefore, high, high voltage. And then, when we activate the word line what happens is, that one of them will essentially sway ((Refer Time: 08:59)) and that difference is amplified by the sense amplifier.

Now, here it is much faster because there is nothing much to amplify is not it. Sense amplifier, which has read the difference and has registered the current ((Refer Time: 09:12)). And here also, there is nothing like a cache, normally nothing like a cache in RAM because what would do is, you read down the entire book. For example, when we are designing register file of say 32-bit processor, it essentially has 32 cells, that will be 1 row. So, we could always read out the entire book. There is nothing, nothing like a column access within a row.

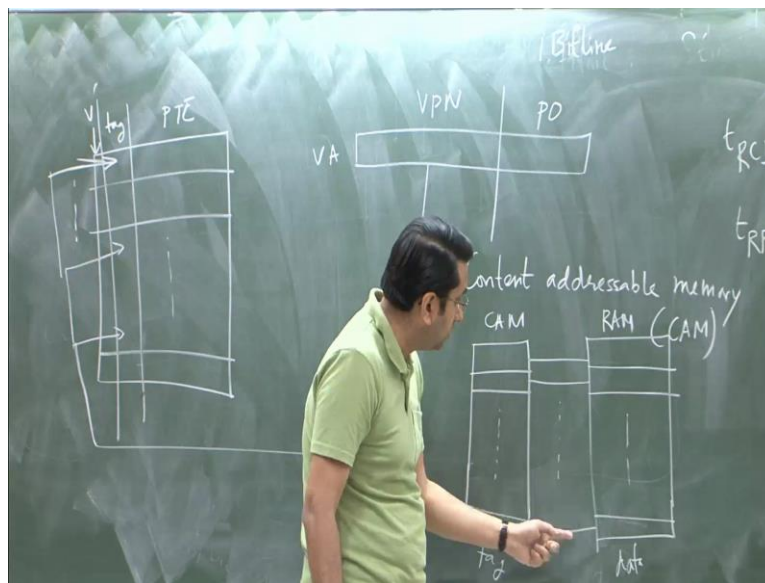
So, essentially, here only the precharge operation that is needed, which will give the paired up operations and there is no concept of ((Refer Time: 09:50)) or anything because there is no concept of column access. What you get out of sense amplifier is ((Refer Time: 09:58)) and immediately, you precharge the array again for the next composition, alright.

And just to remind you how you multiplex the RAM array, so we discussed this also last time. So, this is a single ported RAM in the sense, that I can access only one row at a time. If I access both the rows the values will get ((Refer Time: 10:21)) here because this value and this value will essentially clash and will get something, I mean, some garbage. So, if you want to make it multi-ported you will have to add, dual ported, you will have to add one more access transistor. Sorry. This will have a new bit line also, this said will require that ((Refer Time: 10:58)) and will

be there for all the ((Refer Time: 11:02)).

So, now you can, not only access the same row drawings simultaneously, you can access two different rows also. So, this will be replicated, that is a SRAM. SRAM is used for designing your caches, register files, any ((Refer Time: 11:20))

((Refer Slide Time: 11:59))



Alright, so one small thing that is left slightly connected to this. So, we talked about this ((Refer Time: 11:42)) caches and ((Refer Time: 11:44)) where you essentially need to make multiple comparisons. The question that comes to mind is, let us take the t_{NB} example ((Refer Time: 11:57)).

So, if you have fully associated ((Refer Time: 12:01)), just remind you what the structure was. You have a value grid, you have a tag and we have the page tabulation, that is the organization ((Refer Time: 12:14)) and you have multiples of entries, alright, ok. So, what you do is, if it is a fully associated array, you would take the virtual average, chop it into page offset and virtual page number and you compare the virtual page number with all the tags here, right, ok. And the hope is that ((Refer Time: 12:46)) will match ((Refer Time: 12:48)).

The question is, how do you get the design in such a ((Refer Time: 12:56)). It is not a SRAM clearly, because you are not really accessing 1 array. You are accessing, no, I am sorry, not, not only, not just accessing 1 row, you are accessing all the rows, ok. So, how do you really do this? So, these are called content addressable memory or CAM for short.

So, here what you do is, you separate the tag and the data, ok. So, data is designed as a conventional SRAM, just like that, the data array, ok. The tag is designed in a slightly different way. So, tag is designed in a CAM and what it does is, so this is a tag, a bunch of rows here and this is the data, these are RAM. So, this is just like this and you, and you connect these things. So what happens here is, that when you get this tag you compare all the elements here, alright. So, essentially what happens is, that there is a line through which you will pass this particular tag. There is a comparative rating of each of these cells in the CAM in each of the rows and this line tells me whether there is a match or not, ok. So, this will be either 0 or 1.

The value on ((Refer Time: 14:46)) and this line will be connected with the word line with the corresponding row. So, any line that matches will activate the corresponding row in the data array and will read out the ((Refer Time: 14:58)). For example, if the 10th row has the corresponding tag, the 10th line will go high, everything else will be low. So, we will read out the page ((Refer Time: 15:11)) of the 10th row, which is the ((Refer Time: 15:12))

Student: What is the size of CAM?

Professor: Size means?

Student: How many tags will it...?

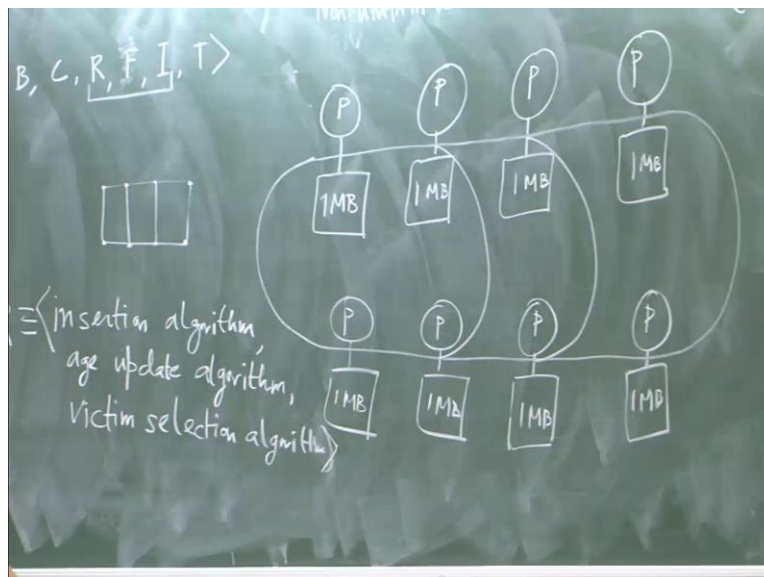
Professor: How many...?

Student: How many tags will it store?

Professor: Well, that is decided by your ((Refer Time: 15:23)), maybe 100, 200, maybe 2, 4, 8, right, maybe small, maybe large. Now, if it is large, it is going to be bulky because each cell is pretty large. Each cell contains not only this bit, ok, it will carry the line that will carry the external, external bit, that has to be compared, and there has to be the comparator also. And based on the comparison it will give out the corresponding line of... So, so that is how you design this, this tag arrays.

You connect the word lines of the data array with the comparison ((Refer Time: 16:03)), ok. So, what I will do is, I will, there is a remaining half an hour, so discussing some of the ((Refer Time: 16:12)) problems in caches and memory. This will hopefully recap also what we did on the memory side.

((Refer Slide Time: 16:33))



So, let us start with the caches, ok. So, we talked about three main parameters in caches, these are the A, B, C parameters, associativity, block size and capacitance. What else is there that determines the cache performance? There is a replacement policy, R. There is an index function, which decides, which set the index. The cache could be inclusive or exclusive with respect to the next lower level and there is something called inter-polar, which I will explain now. So, cache can

be roughly seen today as the ((Refer Time: 17:15)).

Associativity, block size, capacitance, a replacement policy, an index function, inclusive or exclusive and interolarity. So, we know about these six things except the topology, ok. So, what has happened today is, that caches have grown big now. So, you will routinely find on chief caches, which exceed tens of megabytes, ok.

Now, the problem is, that if you design such a large ((Refer Time: 17:51)) it is going to be very slow because you can guess what your bit lines are going to be long because there is a span of very large number of ((Refer Time: 18:00)). So, in a cache one sec is essentially a 1, 1 block is essentially 1 row ((Refer Time: 18:05)). So, essentially a bit line length will be roughly proportional to the number of blocks in a cache. Although it is not done in that way, it will be reorganized in a certain in a slightly different way. ((Refer Time: 18:19)), which is why as the cache get bigger, it will become slower, ok, alright. Here, the bit line lengths increases and the time to, to read out the value from here is directly proportional to the length because that determines how long it will take to stabilize the data ((Refer Time: 18:36)).

So, today what people have done is, that they have divided the cache into smaller chunks, which are called banks just like ((Refer Time: 18:46)). For example, if there were 8 megabyte cache, you will probably design it as 8 banks each of 1 megabyte, alright, so that if you can decide what bank to access, there will be a small cache that you will access, just 1 megabyte. So, topology determines how you exactly layout these ((Refer Time: 19:05)) banks, ok, with respect to the processes on the chip.

So, so let us take an example. Suppose we have today, you know, today we have multicore processors where on each chip you have multiple processes. So, that is how you have 8 processors and 8 megabyte cache with 8 bags. So, typically the way it would be designed is, that a processor will be given a 1 megabyte of cache, 1 megabyte bank, ok. The question is, how do I connect them?

You can think of it as, you know, ((Refer Time: 20:26)), connect them in whatever way I want, as long as it remains a ((Refer Time: 20:30)), alright. So, that is the topology. So, one connection, one possible connection ((Refer Time: 20:36)). Another possible connection could be a mesh. The problem is how do I make it? That is a two cross ((Refer Time: 21:07)).

So, what are the implications? Suppose this processor wants to access a piece of data, which is here. You will have a long ((Refer Time: 21:38)) compared to if he has to access the data here. So, that leads to something called non-uniform access cache architecture or NUCA, ok, right. And you should be able to appreciate, that compared to a ring, in a mesh the access taken is actually smaller. For example, this processor instead of going all the way here, connection of ((Refer Time: 22:22)), ok.

Shortens your average talk time. So, these are called actually a hop. So, this, this one will have a switch here, so which will decide an incoming packet will go which way. It can go either to the, to this bank it can forward, it shall forward round this direction or it can go like this. So, topology essentially decides your average access links that is the impact of it. So is it clear everybody?

Student: Sir, it is analogous to distributed shared memory?

Professor: Yes, very good, yes, exactly. ((Refer Time: 23:00)), these are essentially local memory at the processor. Yes, exactly.

The whole problem is now ((Refer Time: 23:09)). The only difference is, that their each hop maybe much more costly than the hop here. These are essentially short links, switches are simpler and normally faster, ok.

So, what are the research problems here? So, we have to ((Refer Time: 23:29)) some of the details and you know, extract some of the research problems here. So, what are the problems? That, that is interesting is, if we fix, let us say, ABC and T, ok. So, let us suppose, I ((Refer

Time: 23:49)) a cache and tell you that this is ABCT, what is the best you can do? So, then essentially our degrees of freedom were R, F and I. So, we can decide whether inclusive or exclusive. We have discussed, that the implications of that in trade off between inclusive and exclusive.

We can design smart index functions, ok. So, what are the implications for the index function? Can anyone please guess ((Refer Time: 24:15)). Why should I at all want to spend time designing an index function? Sorry.

Student: Speed of access...

Professor: Speed of access?

Student: Yes

Professor: Can you give an example?

Student: Well, resolving the address, at that time...

Professor: Resolving the address? What is the implication of the index function?

Student: ((Refer Time: 24:48))

Professor: How?

Student: It is like, it should be uniformly distributed across the...

Professor: What should be uniformly distributed?

Student: The row address whatever, I mean, whichever row we are accessing, we get that address
((Refer Time: 25:09))

Professor: What is it called?

Student: A set

Professor: Set.

Student: We decode the set from it, it should be uniformly distributed.

Professor: Set should be uniformly distributed! What does that mean?

Student: The value of the set should be...

Professor: What is uniformly distributed?

Student: The data should be like, uniformly distributed across the...

Professor: Data? What is it that is uniformly distributed, first you make that clear. There is a bunch of sets, which are fixed by the way number of sets is equal to. You can easily calculate, right. C over A times B , statistics, ok. So, what is in you want to distribute it, what is the range of index function that, right, ok, alright.

Student: So, whatever the output be, that is, only index function should be uniformly distributed.

Professor: So, my, my domain of the index function should be uniform distributed on the range; that is all. You really want the addresses should be uniformly distributed across the sets so that each set is equally balanced. ((Refer Time: 26:15)). So, that is, that is, that is what index function

I is. If you balance all the sets so that whatever conflicts that will happen, will be uniform across the sets.

And the replacement policy, as you can guess, of course, it will have huge implication on the performance of the cache. So, usually the way the replacement policies are designed or the replacement algorithms are designed, there are three distinct pieces of replacement algorithm. So R is usually decomposed as there is an insertion algorithm, which I will explain what that is there is an age update algorithm. There is a victim selection algorithm.

So, what is the insertion algorithm? That decides when you have a new block coming in into a set, what age should it get. That is the insertion algorithm. So, then, that means, what would it be in rank relative to the currently deciding elements in the set, ok. So, for example, in LRU replacement policy what you do is, you give it the highest priority so that it becomes a MRU block. ((Refer Time: 26:54)).

The second component is age update algorithm that says, whenever you have an access to an element into this set how are the ages of the element of the set be updated, right. For example, the LRU policy, the element that gets access, will become the highest priority element, that is the MRU element, alright. And the other ranks remain essentially unchanged. So, the MRU element will now be ((Refer Time: 28:24)).

And the last one is victim selection that is based on these ages currently ((Refer Time: 28:33)) to may use of a new block, ok. So, it is possible to architect each one of these separately. You can do that. You can keep the victim selection algorithm fixed, you can keep your insertion algorithm fixed and try to ((Refer Time: 28:47)), that is possible. You can ((Refer Time: 28:52)) combination. So, I, I really have time to go into details of these. There is a, you know, two, three details of ((Refer Time: 29:04)) design replacement algorithms. So, if you want I can give you some of the recent references, yes.

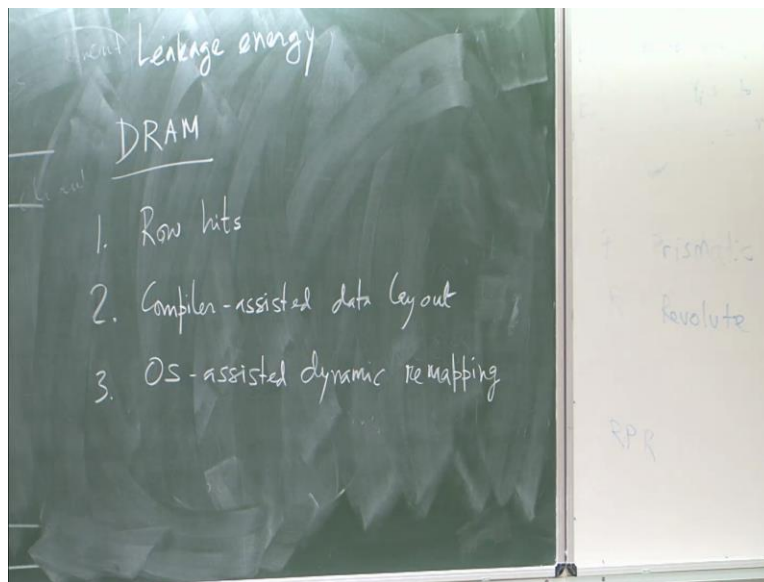
So, these are pretty much the ((Refer Time: 29:16)) ordered research problems on caches, that is,

architecting these three. Now, of course, lockers, I mentioned here to be concept because this course, in this course we did not really discuss about much, discuss about this topology much, ok. But that is also a very important research problem, that is, how do you really lay down the banks property, so that as you have already mentioned, that access ((Refer Time: 29:43)) gets effected, right. How do you really lay down the...

So, one algorithm that people have explored is, still ABCT are fixed, alright. To mitigate this problem, that is, this processor ((Refer Time: 29:56)) data here, it have to traverse a long distance, ok. So, what that, what these algorithms do is, they actually try to indirectly tweak the index function. So, at one time what they do is, they monitor the accesses to these banks from the top and they calculate the affinity matrix, that is, which processor is trying to access the data, which data the most, alright. And then, try to remove this data dynamically closer to that processor. So, essentially, in effect it is changing this index function. It is saying, that how should this data be indexed into a cache, alright. So, instead of indexing this data into this bank, it should be indexed into this bank, actually. So, this implicitly changes the index function.

So, that is one way of making this problem of ((Refer Time: 30:45)). So, that is about your cache components. The other one, the other one important problem in caches is ((Refer Time: 30:54)), that is, usually we talk about cache energy especially for large caches.

(Refer Slide Time: 31:00)



So, here essentially the problem is, that as the transistors are getting smaller, they are getting leaky. Leaky, in the sense, that even if you turn off the input voltage for transistor, it does not turn off completely. ((Refer Time: 31:23)) from the high supply voltage to the ground current continues to flow, that is the weakest current.

So do not get confused, do not confuse this with the leaking of charge in DRAM, ok. It is not like the cells are leaking in the cache. It is not like that. It is just, that there is a path from your high voltage to low voltage that is always on. So, that is the leakage current ok and this is a weak problem when you have large ((Refer Time: 31:52)), ok.

So, caches have become so large, ((Refer Time: 31:53)), ok. So, caches have become so large, that they are the biggest consumer of leakage energy in a chip, ok. So, how do you really optimize the leakage energy? So, some of the common techniques, that people have tried is, you can, architecture techniques. Of course, there are many circuit techniques, I would not go into that.

So, you have, let us say, you know, an ((Refer Time: 32:19)). You are running an application, so

you start monitoring the heat rate of the application that it is drawing from the cache. Suppose you shut down ((Refer Time: 32:31)), so you, you will have a cache with $A - 1 \theta$, ok, instead of A . Thus, the heat rate of the application change if the access is known. Then, you have essentially saved one working portion of the leakage energy but shutting down one, one way without losing the performance, you can keep doing this. You keep on monitoring and keep on shutting down the ways of the cache, ok.

And of course, you have to do the other way also. We find, that you are losing in performance, then you have to turn on ((Refer Time: 33:04)). The problem is, that turning on actually takes time. It is not, it is not the 0 time limit. So, you have to be very careful when you are shutting down the ((Refer Time: 33:14)), so remember that. You cannot bring up the performance as fast as you can, that you will ((Refer Time: 33:19)).

The second approach that people have tried is, that although you have an 8 megabyte cache, it may be, you do not need the full cache at any point of time, ok. You may need only a small portion of that but the problem is, that the way the data is mapped on the banks, it is all scattered. So, some data is here, some data is here, some data is here. So, some of the data, so a subset of data is always on at least 1 bank. So, to keep all the banks on all the time, so what we can do is, that again you can apply the same technique that we are talking about. You can figure out the application and you can figure out ((Refer Time: 33:57)) data access together. So, you can plaster them all in small number of banks and shutdown all the remaining banks ((Refer Time: 34:04)). So, both of these techniques require online monitoring and often people employ machinery techniques to figure out what needs to be done, that will, that will have the optimal effective notion, ok. So, that is about caches.

I will just talk little bit about DRAM although we have discussed these things already. So, in DRAM there are, you know, so in DRAM, the access ((Refer Time: 34:42)) is determined by row hits. So, that, that is the major research problem, that is, how to maximize the number of row hits. So, we have discussed one, one possibility in the class last time, that the memory controller could actually cluster requests that go to the same row. So, that will make sure, that ((Refer Time: 35:06)) row hits. But the problem is, that this may lead to lots of lack of ((Refer Time:

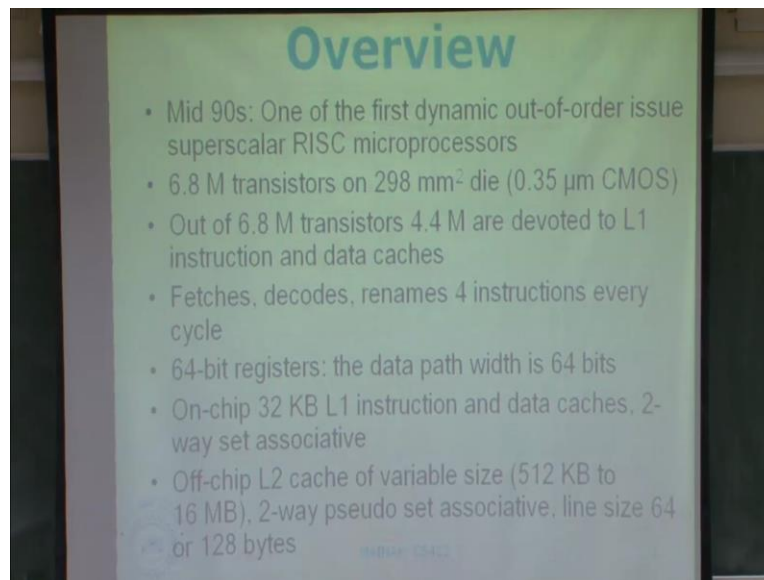
35:13)). What may happen is, that you may end up giving the DRAM to one particular thread of your computation, which, which essentially happens to have a sequential access pattern.

So, it have a lot of ((Refer Time: 35:29)) cluster, the rows of that ((Refer Time: 35:31)) cluster, the rows of that, you know, cluster the accesses of that thread ((Refer Time: 35:36)) and you keep on scheduling. Other threads will keep on ((Refer Time: 35:39)) simply because they do not have enough row ((Refer Time: 35:42)) at some point, you will break this. So, there is a tradeoff between thread list and ((Refer Time: 35:50)).

This one tries to maximize your throughput ((Refer Time: 35:54)). So, you come up with the fairness matrix here and to figure out how to promote that. So, essentially what I am saying is, that you just cannot keep on scheduling request from particular thread just because it has, it maximizes the number of row hits. You have to get into other threads also. And other problem is that, so that is the, that is the, that is the prospect of memory controller. The second way of attacking this problem is, that you could lay out the data in such a way that applications would access in the particular locality of the program. The data will be sequential and they will be accessed to maximum row hits.

So, here you will require some, the ((Refer Time: 36:40)) compiler for locating the data. Compiler-assisted data layout or you could try out dynamic technique, just this it could figure out that. Currently the application is trying to access data from four different rows, ok. You could dynamically change their addresses, remap them through the operating system and put them in a single row, that will ((Refer Time: 37:12)). So, I will put that also here, OS-assisted dynamic remapping alright ok.

(Refer Slide Time: 37:56)



So, roughly for the next three or four lectures what we will do is, we will look at some of the commercial processors. We will not look at instruction set of them, we will look at the micro applications, how they actually implement instruction set in a processor.

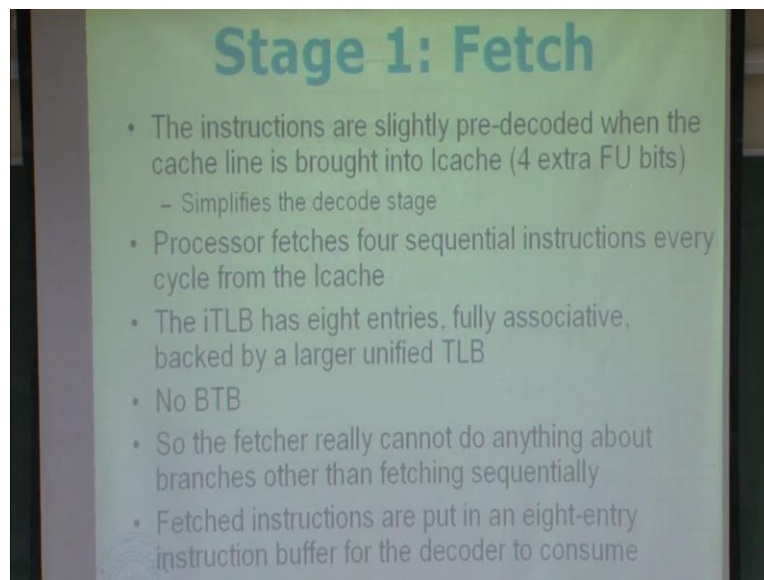
So, the first one is the MIPS, goes back to mid 90s. This was one of the first dynamic out-of-order issue superscalar RISC microprocessor. So, here is some information about this processor. It has 6.8 million transistors. So, you should compare this number with today's high-end processors, which are routinely 1 billion, 1 and a half billion, something like that. A 6.8 million transistors on a, 280, 298 millimeter square die. There is a sign which it on 0.35 micro CMOS. We are, today, what is this today? Anybody? Size of... ((Refer Time: 38:34))

So, that is in production. The leading chip manufacturers have already demonstrated much smaller transistors, ((Refer Time: 38:50)). So, out of 6.8 million transistors, 4.4 million are devoted to L 1 instruction and data caches. So, you observe this ((Refer Time: 39:00)) today actually. So, so percentagewise how much is this roughly? Two-third, right. So, today actually it is even more. It is about 90 percent transistors become caches, 8 percent goes to logic. So, that is the deviation in memory and logic.

It fetches, decodes, renames 4 instructions every cycle, which has 64 bit registers. So, the data path is 64 bit wide.

There is on chip 32 kilobyte L1 instruction and data caches 2-way set associative; off chip L2 cache of variable size, which you can reconsider; 512 kilobyte and 16 megabyte 2-way pseudo set associative. If you have forgotten this one, you can lookup past lecture notes, will also, I will also refresh your mind when we discuss this L2 cache. A line size 64 or 128, it is again reconsider ((Refer Time: 39:58)). So, that is a summary of the processor.

(Refer Slide Time: 40:05)



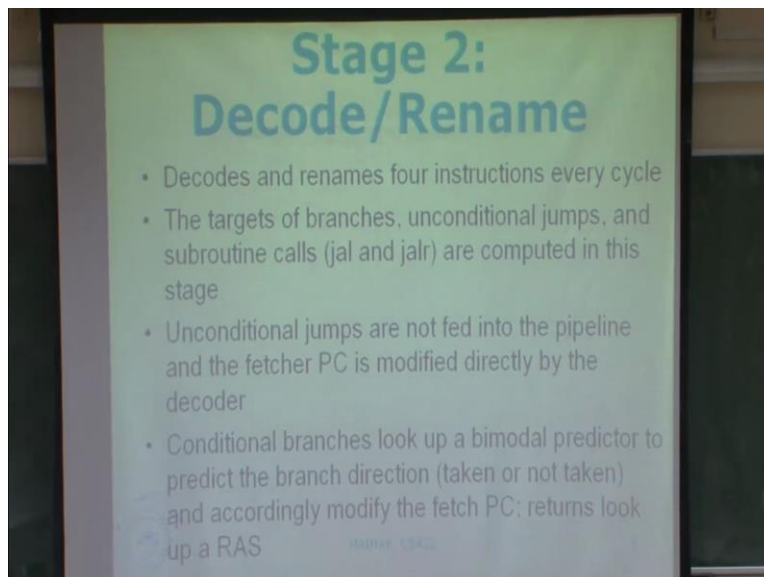
So, what we will do is, we will walk through each of the pipeline stages. So, in the fetch stage, instructions are slightly pre-decoded when the cache line is brought into instruction cache. So, essentially what happens is, that before you fill in the instruction cache each instruction is appended with 4 extra functions in 8 bits. So, essentially it says, that to which function unit this instruction should go, alright.

So, that simplifies your decode ((Refer Time: 40:40)) instead of that before you put the instruction, instruction cache should do this. Actually it is a very simple thing to do so that the

((Refer Time: 40:50)) can leave this particular instruction, you have got to decode it every time processor fetches 4 sequential instructions every cycle from instruction cache. So, remember, that it has fetch decodes real ((Refer Time: 41:01)). So, you have to do this for to be able to feed your pipeline properly.

The instruction TLB has 8 entries fully associative backed by a larger unified TLB. So, there are two levels of TLB. The level 1 instructions will be small, but the second level will be quite large. There is no branch target buffer, so the fetcher really cannot do anything about branches other than fetching sequentially. ((Refer Time: 41:31)), so fetch sequentially until you have some ((Refer Time: 41:35)). Fetched instructions are put in an 8 entry instruction buffer for the decoder to consume. So, that is the fetch stage. Any questions?

(Refer Slide Time: 41:49)



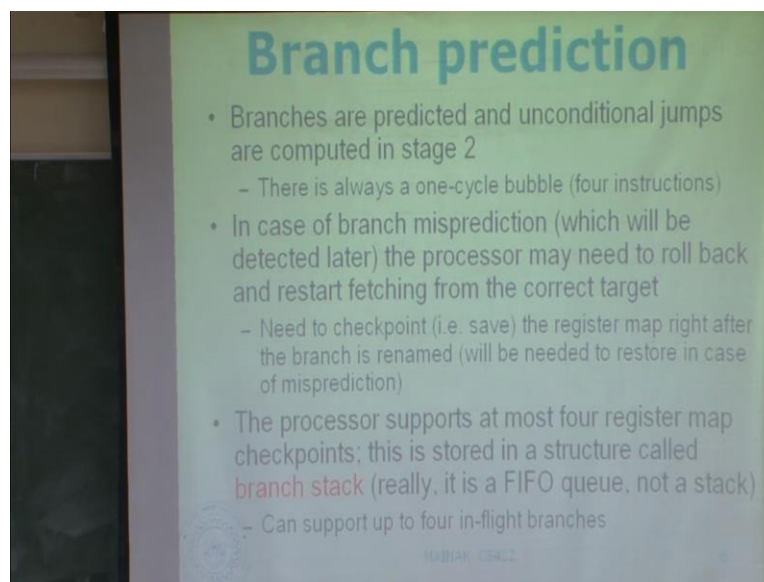
Stage two, decode/rename, decodes and renames 4 instructions every cycle. The targets of branches, unconditional jumps and subroutine calls are computed in this stage. What this means is, that you will actually have a ((Refer Time: 42:01)) at this particular stage to do this because a target of conditional branches will require in ((Refer Time: 42:08)) because you have to add ((Refer Time: 42:09)).

For unconditional jumps you need nothing. The decoder will give you the ((Refer Time: 42:15)) part of the instruction. Subroutine calls also will come greatly from instruction, are computed in this particular stage, ok. So, so these are all available here. Unconditional jumps are not fed into the pipeline and the fetcher PC is modified directly by the decoder. So, the unconditional jumps do not make any further ((Refer Time: 42:35)), right. Here, the decoder tells the fetcher to fetch from the target program, ok.

Conditional branches look up a bimodal predictor to predict the branch direction taken or not taken and accordingly, modify the fetch PC and the return instructions will look up a return at this time. So, we have discussed all these gadgets, what they actually do. In fact, for the homework you should be working on the bimodal predictor and see how, how accurate that is. Any question on this?

So, this is the stage where actually fetcher gets the first information about what to do on a branch because ((Refer Time: 43:11)).

(Refer Slide Time: 43:16)



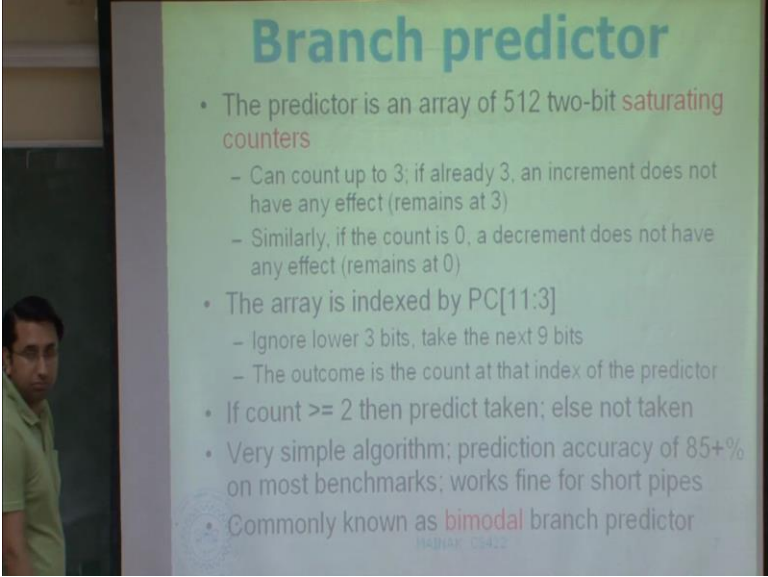
So, little bit on the branch prediction, this seems you already know. The branches are predicted

and unconditional jumps are computed in stage 2. So, there is always a one cycle bubble. ((Refer Time: 43:27)). So, here is what that the branch ((Refer Time: 43:31)) the legacy fits the, continues there, alright. But it is only just the one instruction after the branch, the remaining three are not in the branch ((Refer Time: 43:40)).

So, essentially you can think of it as one cycle bubble constitute three instructions ((Refer Time: 43:45)) provided. Of course, the branch instruction was the last one in the previous ((Refer Time: 43:54)) of the 4 instructions. In case of branch miss prediction, which will be detected later, the processor may need to roll back and restart fetching from the correct target.

So, how to do that? You have discussed this also in the class. You need to check, point the register map right after the branch is renamed, which will be needed to restore in case of misprediction. The processor supports at most 4 register map checkpoints. This is stored in a structure called branch stack. It is really a FIFO actually, I do not know why they call it a stack. It is just a FIFO queue. So, which means, it can support only up to 4 in-flight branches because every branch is required a check point at register map and it clear only space for 4. A 5th branch will have to stop if the previous 4 branches have not been resolved.

(Refer Slide Time: 44:45)



Branch predictor

- The predictor is an array of 512 two-bit **saturating counters**
 - Can count up to 3; if already 3, an increment does not have any effect (remains at 3)
 - Similarly, if the count is 0, a decrement does not have any effect (remains at 0)
- The array is indexed by PC[11:3]
 - Ignore lower 3 bits, take the next 9 bits
 - The outcome is the count at that index of the predictor
- If count ≥ 2 then predict taken; else not taken
- Very simple algorithm; prediction accuracy of 85+% on most benchmarks; works fine for short pipes
- Commonly known as **bimodal** branch predictor

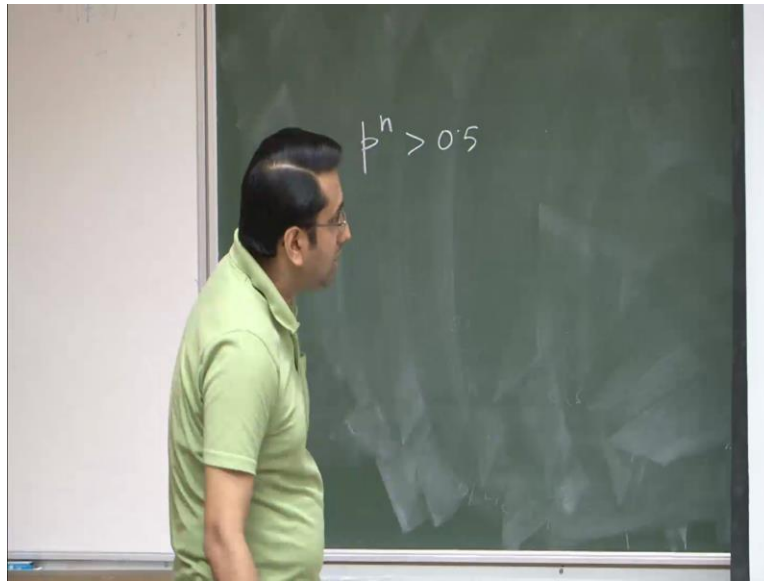
The predictor is an array of 512 two-bit saturating counters. So, that is the bimodal predictor ((Refer Time: 44:53)). Can count up to 3, so that is the definition of saturating counters. So, count up to 3. If already 3, an increment does not have any effect, remains at 3. Similarly, if the count is 0, a decrement does not have any effect, remains at 0 ((Refer Time: 45:07)).

The array is indexed by this bits of the PC, bit 3 to 11. So, it actually disturbs the lower three bits: 0, 1 and 2. Why is that, any idea? We said, that it should discard two, the lower two bits because instructions have 4 byte ((Refer Time: 45:26)). Here, they discard three. Why is that? What is the next instruction to the branch? It always execute, right. ((Refer Time: 45:46))

So, can I think of the branch instruction to be actually used form of these two instructions, right? Because I know, that the next instruction to the branch will always be executed and the restrictions that, which ((Refer Time: 46:04)) is, that it cannot ((Refer Time: 46:05)). So, that means, I can actually assure, that with branch instructions effectively ((Refer Time: 46:12)). So, I can actually remove these lower three bits. So, ignore lower 3 bits, take the next 9 bits. We take the 512 bit array and the outcome is the count at the index of the predictor.

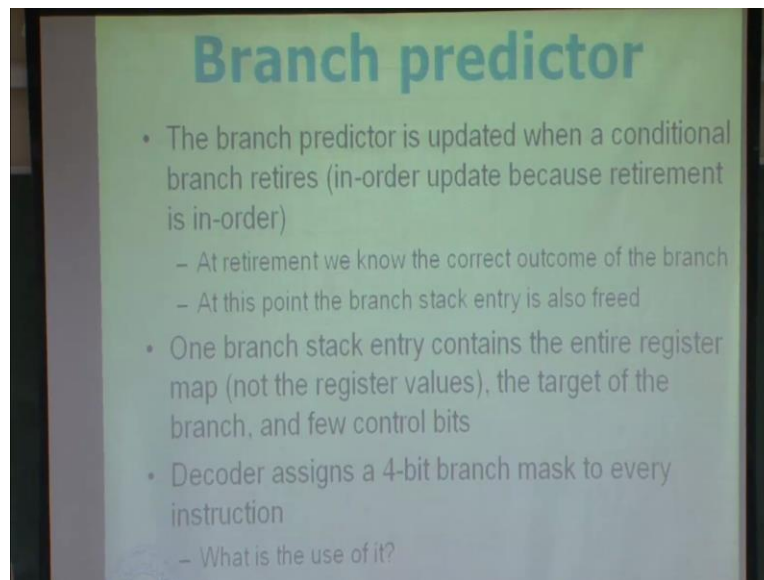
If the count is, is 2, then ((Refer Time: 46:31)). A simple algorithm prediction accuracy of 85 plus percent, that is what it showed, for there, for there benchmarks they used. Works fine for short pipes. So, this also we discussed actually.

(Refer Slide Time: 46:52)



How this prediction accuracy actually has implication of pipeline lengths because we say, that if we have a prediction accuracy of p and you allow n branches concurrently in-flight, then I, then I want this to be at least ((Refer Time: 47:00)). So, actually if you do this, in this case, in this 4 because we allow 4 branches, right, that is what we were just saying. If you have 4 outstanding branches already, the 5th will stop. So, any support p is pointing at 5. So, actually you have find, you can calculate that. You were actually above 0.5, 0.85 ((Refer Time: 47:24)).

(Refer Slide Time: 47:25)



The branch predictor is updated when a conditional branch retires. So, in order update because retirement is in order, ok. At retirement we know the correct outcome of the branch at this point. The branch stack entry is also freed ((Refer Time: 47:37))

One branch stack entry contains the entire register map, not the register values, the target of the branch and few control bits. So, target has to be saved in the, in the, in the branch stack because that will be needed to update the predictor ((Refer Time: 47:54)).

The decoder assigns a 4-bit branch mask to every instruction. Why is it needed? So, the pipeline there can be at most load in-flight branches, that ((Refer Time: 48:08)) I am saying, that each instruction goes out with a 4-bit branch master. Anybody guess what this is. First of all, what is this branch master?

Student: This is an array, an array of 2 or 3 bits.

Professor: Yeah, I know. What is the, what is it holding actually?

Student: It is the ((Refer Time: 48:27))

Professor: No, it will take, that you attach this mask to every instruction ((Refer Time: 48:37)).
So, can someone define a branch master?

Student: ((Refer Time: 48:45)) 4 branches and that branch is, I mean, that branch is taken, then this instruction is executed. So, you have the bit for that.

Professor: Yeah, so essentially what you are saying is, that if I have an instruction i , tell me which branches it is control dependent on. So, that is this mask. ((Refer Time: 49:16)).

Student: It may be depending on all of them.

Professor: Yes, right. So, it may have all the bit set. So, it means, it has gone through 4 branches. So, the fate of instruction i depends on the fate of all these branches, right. So, why here on my interaction list?

Student: ((Refer Time: 49:31))

Professor: How, how is it useful?

Student: Sir, there is some restriction and we may have to, if the, if we execute a ((Refer Time: 49:37)).

Professor: Exactly, so what I do is, whenever I miss predict the branch i , let us say i ((Refer Time: 49:56)), the second branch.

(Refer Slide Time: 49:57)



So, I prepare a mask like this, 0100 and send this mask to the pipeline. So, this mask will be ended with the masks of all the instruction. Only the non-zero outcomes will actually ((Refer Time: 50:10)) because we know, that those are the branch, those are the instruction that are control dependent on this particular branch, ok.

So, it might be helpful if you are willing to do the thinking. ((Refer Time: 50:21)) before you come to the class. It will help you understand lectures. So, we will, after ((Refer Time: 50:28)).