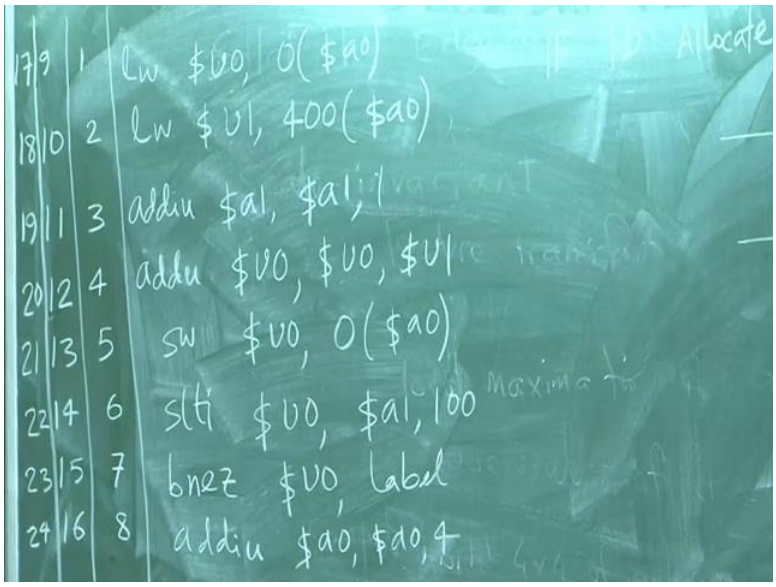
Computer Architecture
Prof. Mainak Chaudhuri
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

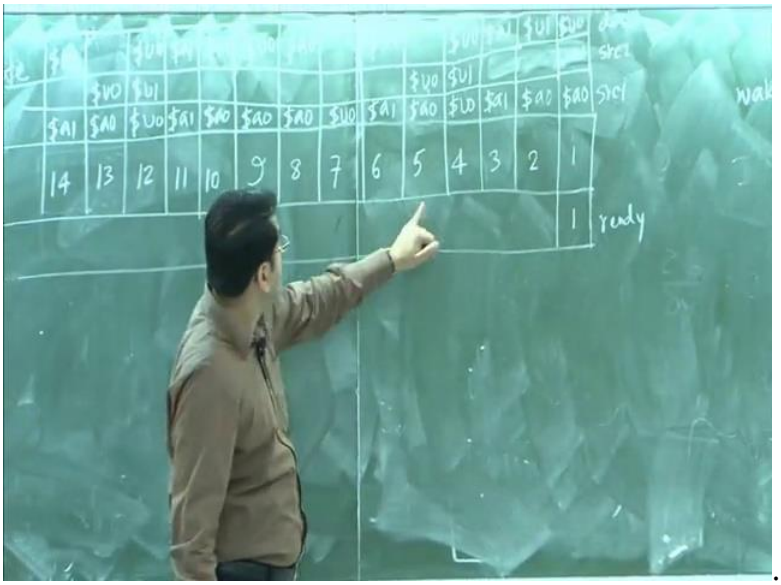
Module - 1
Lecture - 21
Dynamic scheduling, speculative execution

(Refer Slide Time: 00:18)



I will look at the detail of algorithm that we discussing yesterday. So, this is the code slip that we looked at yesterday, one iteration of the loop. I have also numbered the instructions as they will appear in the dynamic sequence. So, this is one iteration the first one, this is second iteration, this is third iteration and so on and so forth. So, essentially we are going to execute this on our model, that we discussed yesterday. We have a single issue queue. So, this is my issue queue. So, I have not shown what it really contains. So, the idea is that; instruction to be fetched in that order, as numbered, will be decoded in that order, and will be allocated in queue in that order. So, essentially instructions will sit in the queue exactly in that order, in the in the order of the instructions shown there. And then from the queue will select instructions and execute them. So, any question on yesterday's material, or any doubt that you might have on this model.

(Refer Slide Time: 01:55)



So, what I am going to do is. So, each queue entry is going to have, of course, it will have a slot id, which is the id of the queue entry. It will have a bunch of source registers, and at most one destination register. So, let us take the first instruction there. So, its number one; source dollar a 0, and destination dollar v 0. So, for execution purpose of course, I will have to also remember the up code etcetera, any other argument like for example, the immediate argument also I will require. I am not showing them here. So, I will put the destination at the top; that is my first iteration. So, this is source 1, source 2 desk etcetera. So, that is the order in which instructions will be allocated in the queue. Now when you allocate an instruction, and you know if the instruction can. So, let me tell you the remaining pipeline stages. So, that might help you, to visualize what is going to happen in future. So, next stage is called wakeup, then there is a select issue stage, and then usual execution memory write back. So, the wakeup stage essentially wakes up any waiting instruction that might be waiting for the operands.

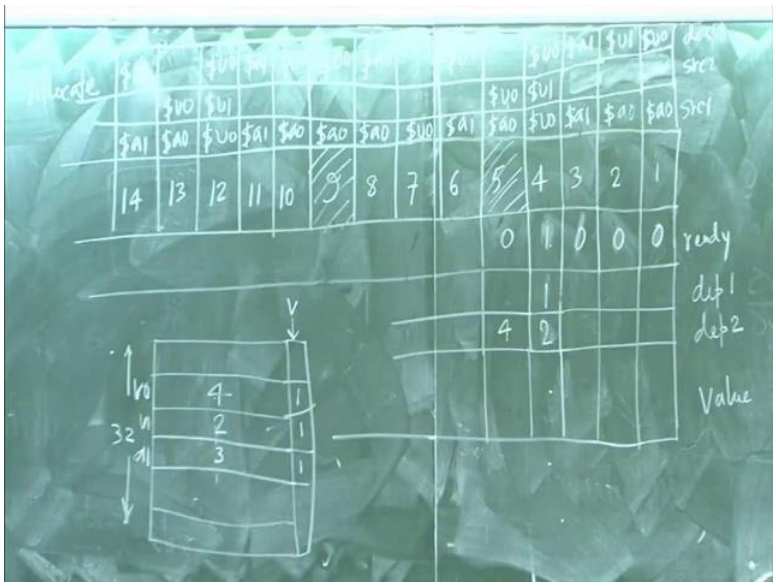
So, when I allocate an instruction, I need to know, if the instruction can be selected in the next cycle. So, the instructions which are ready, they will actually skip this particular stage, there be wont any wakeup stage for them. So, how do I know that, when I allocate a new instruction how do I know if it can participate in the selection in the next cycle. So, how do you figure this out. So, let us suppose that I am fetching in this order, the first instruction comes, queue is empty so nothing needs to be checked. So, I will actually put a ready bit here. So, in this case a ready bit is going to be 1, meaning that next cycle it

can execute, no constraint on this. So, these are ready bit. What about the second instruction. So, I take dollar a 0 and compare with, why is that, because this instruction is currently sitting in the queue with the destination v 0.

So, will that be a correct algorithm all the time. Should I compare with all the instructions before me, the destination of all of them. Does everybody see that. So, at any point in time whenever I put an instruction in the queue, it needs to compare its source cells with the destinations of all the instructions before it. Is there any way to optimize this. It is a lot of large actually. For every instruction you have to do, you know a linear search, and whatever way you want to implement it, there will be a linear number of comparisons, any way to optimize it.

Remember that we discussed last time that as soon as the instruction executes I may not write it to the register file. I have to wait until its turn comes for example, 7th instruction may execute now, but you will you will not be able to write it until everything else before it, has written to the register file. No, it is not the question of bypass, I cannot write it to the file, because there may be an exception that I may have to handle, the 7th instruction writes to the file, fifth one takes an exception, how do I recall for the register file now. On a write bag I make the binary as 0. So, what he has suggested is that, I prepare a table, what does the table store. It tells me for each register so each entry contains. So, let us suppose in MIPS we have 32 registers. So, these going to have 32 entries, is that what you are saying.

(Refer Slide Time: 10:05)



A particular entry; for example, let suppose the entry for register one, will tell me the slot id in the queue, for the instruction which generated this register for the last time, which was the last producer of this particular register. So, whenever an instruction shows up with this source register, I look up the table and immediately know who I depend on, because all I care about is a last instruction to produce this value. I do not care about anybody else before it actually, is that correct.

So, for example, here, this instruction cares only about this particular instruction. It does not care about anybody else, in this particular sequence. So, let us try to populate this table. So, the first instruction comes. The entry for dollar v 0 will get populated with 1, and will also have a valid bit, so that is known a binary thing. So, is valid bit is normally 0, I will mark it one, whenever an instruction shows up with these particular. So, I am assuming that this is an entry for dollar v 0, whatever that is. So, when I bring this instruction in, it looks up this table, goes to the entry for a 0, finds that its actually invalid, which means I do not depend on anybody. So, essentially this instruction is also ready to execute.

Next I bring in this one. So, also I establish dollar v 1 entry. So, let suppose this is v 1, this becomes 2 valid. So, then this instruction comes. So, it is dollar a 1. It requires dollar a 1 produces dollar a 1, looks up the entry for dollar a 1, its invalid does not depend on anybody, so it is ready to go. So, let us suppose this is my dollar a 1 entry. I mark it 3

make it valid, because it is it solves the producing dollar a 1. Next instruction comes it needs to look up two things; v 0 and v 1. It looks up v 0, so it finds and depends on 1. It looks up v 1 it depends on 2, so that is what it marks here. So, it is not ready to execute, and it has a dependence list. At most two things it can depend on a particular instruction, because there are two sources. What are the dependencies; 1 and 2.It depends on these two slots. So, I depend on these two things. What about the next one. So, it produces v 0, now something interest is going to happen. So, it is going to overwrite v0. So, I want to change this entry now, because a new inclination of v 0 is actually 4.

Then these one comes, it also needs two things a 0 and v 0. So, it looks up, a 0is free, but v 0 comes from 4. So, this is also not ready to execute, and it is a 0 is free, but v 0 has a dependence on 4so on and so forth. You as the instruction come, you look up this table. So, store does not produce any value. So, I do not change this table on this instruction. You look up this table find out which depends on, and that is how you fill up this queue. Is this step clear, what the allocate stage does exactly. It is just picks up the next free entry in the queue, and fills up this particular slots whatever it is, and that is it, that is what the allocate stage is doing. Any question. So, will come to this stage very soon, let us keep over to the select stage. So, the select stage what it does is that, it picks up the entries with the ready bit on, and select the subset of those for execution. It can select any subset for that matter. So, it discussed that also couple of lectures ago that, instructions which are ready can execute in any order.

So, let us suppose that for this particular cycle it selects these three instructions. So, it assumes few things, when you say that it selects these three instructions; that mean, I have. So, what are these three instructions; 2 loads and 1 add. So, for these three instructions to execute together in this particular cycle. So, essentially they will move on to the execution stage in the next cycle. I have missed one stage. So, in these stages essentially the selection hardware says that, you can issue these three instructions in the next cycle which means they will go to go and access the register file. So, these three instructions do not depend on anybody. So, they will go and access their source register. So, in this case they need dollar a 0 dollar a 0 and dollar a 1. So, as soon as you say that, you implicitly say something about the register file organization, you say that the register file is designed in such a way that, it can source three operands in a cycle, because in this case you need three, and in fact, when you say that I can issue three instructions every

cycle in the worst case, you should be prepared to source 6 operands. So, in this case if just happened that these instructions did not have a second source. Is it clear to everybody, this particular thing. So, essentially what I am saying is that, how much you can issue in a cycle, depends on number of available ports in your register file. How many source operands you can get in a cycle. So, that is about your register file.

Now, we want to execute 2 loads and 1 add instruction in a cycle .So; that means, what, what kind of functional units do I need, to be able to do that. Three adders, why three adders exactly, two for generating the address of these two instructions, and one for actually computing this instruction, so I need three adders. So, that is another implicit assumption that we are making when you say that I can actually execute these three in a cycle. So, what it means is that it puts one more constraint on the selection hardware. It should be aware of what are the function limits that are available in the machine. It cannot just pick an arbitrary subset. So, gradually restriction the subset you can see that. First we say that you depend on number of read ports in the register file, how much you can read in a cycle, and then you are saying that how what mix of function units you have, that also decides what subset you can actually issue. So, unit 3 adders what else. What else do I need to execute those three instructions in a cycle, 2 memory ports. So, I should be able to do two load instructions in a cycle. So, I need 2 data memory ports. So, then if I have three adders and 2 data memory ports, then I am ready to send those three instructions to the execution, and they can execute concurrently without any problem. So, what happens after they finish executing.

So, essentially we are going to turn these bits to 0, ready bits, because they have essentially executed. They are not eligible for participating in future selection. Now the question arises that where do I put the values that these three instructions produce. I cannot send them to the register file, because it may not be time yet. So, it creates one more entry here which is the value. So, here it will store the value of the load. So, the values will be filled up here here and here. What else has to happen, you have to wake up the waiting instructions, that are waiting on me. So, how do I achieve that. So, what happens is that whenever instruction finishes, it will broadcast its queue slot id, overall the entries. So, whenever this load instruction completes, it will essentially send this particular entry id to everybody in this queue. And everybody else job, is to pick this slot

id, and compare with these two entries here. If there is a match; that means, one dependence has been resolved. This particular operand is ready.

So, for example, in this case, 1 will match this one, and essentially this will go away; that means, this is no longer dependent on this instruction. This instructions will completes, will broadcast to and this will also go away, and when both go away this bit will become one. So, this is ready to execute now. So, in the next cycle it will actually contained for selection. So, this is the wakeup cycle, that this instruction goes through. And you can see that an instruction can have at most two wakeup cycles, in a particular cycle; one particular dependent dependence may get resolved, in another cycle another dependence may get resolved. I am coming to that. So, where is it going to read the value. So, in this case it needs v 0 and v 1. So, essentially these two slots, will give the register value. So, it is not going to read from the register file, but it will be actually going to read from the dependent q slots. Remember that the values are stored over here. Well it was recording the dependence here.

No not yet, will read the value in this stage. So, I am coming to that. So, I am avoiding the hard corner cases. I am just trying to describe their basic algorithm. So, this is ready to go, this is not yet ready to go, and there will be many other instructions which are ready to I have not shown actually, you can fill up the queue and see. So, in the next cycle, this instruction is now eligible to participate in selection. So, let us assume that the selection hardware selects these particular instructions. So, now in the next cycle this instruction has to read the register file. So, now, the question arises, the value may not be in the register file, the value maybe sitting here actually in these two slots. So, essentially now, you interpret these two entries in a different way. You figure out if these two entries essentially tell you where to get the value from. So, in this case you will get the value from these two slots, and go to the execution unit, complete execution, broadcast your slot id, now you will essentially going to wake up this one, because this has 4, so4 will match actually. And that is instructional issue, and finally time will come when you want to write back, and the write back will happen in this order exactly.

For example these instruction; the first one is eligible for write back as soon as it completes, because it is the head of the queue. So, when the time comes to write back, essentially what will happen is that will pick up this destination id, pick up the value, send it to the register file, and this queue entry will become free. So, if you design it as a

circular queue, the pointer will move, that entry will become now eligible for allocation again. Is the basic algorithm clear. So, any question. So, one point to notice is, that we have implicitly done renaming; that is one point, because essentially we are using this queue entries as an alternate name for the destination registers. For example, here I could execute two instructions with a same destination without any problem, because they essentially hold the values in two different slots in the queue, and they will write back to register file in this particular order.

So, I can concurrently execute them. Of course, in this case there was a dependence, which is why I could not concurrently execute them, but otherwise if there was no dependence I could actually. Also notice that, I can also resolve memory dependence, because the store value here. So, what happens is that when the store instruction executes, this particular one, the value actually still remains here. It does not go to memory yet, because it cannot. Only when the store comes to the head of the queue this value will be moved to the memory. So, then the question arises, what does it mean to issue a store instruction then. So, in this particular model it actually does nothing. You cannot do the store at this point why is that?

There might be a load before it. There might be a load after it. What is the problem then. Yes it should need a new value; that is fine.

Student: (( ))

So, that is one problem. So, for now, let us assume that because of exception I cannot send it to memory. So, I am coming to your point very soon. So, store cannot really modify memory yet, because there may be a chance than an instruction here may take an exception, then you cannot modify memory, until it turn comes. So, that also gives you memory renaming essentially, because now I can have two store instructions in this queue, going to the same address. They can sit in two different slots. So, when I introduce cashes will see that actually it make sense to issue a store early. There is some meaning, but for now there is no meaning actually, it does not do anything. It just sits there with the value, when it comes to the head it will send it to memory. Now what he has mentioned, we discussed this also yesterday that suppose. So, these are store instruction, this one here. This particular entry the store instruction, it tries to some

address which is this, and we do not know until the store instruction is issued, and it executes. So, in this particular pipe stage, I actually get to know the address of the store.

Now, let suppose that. So, here our ninth instruction is a load; this one, ninth instruction is a load instruction. And the address of which is also I do not know, until it issues and executes, but on the face of it, if you just look at it. I could have actually issued ninth instruction along with 1 2 and 3, because it would be ready, because dollar a 0is. Sorry, yes it cannot, because it depends on this actually. in this case I have a dependence, So which is why it cannot, but if I did not have a dependence then I could have issued actually9 with 1 2 3, skipping over the store here, which maybe a problem, which you discussed yesterday, because it may be a problem if these two addresses are same.

Then the ninth this load will then get a wrong value from memory, it should be getting the value from this store only. The problem is that I cannot really resolve it until, both of these are issued and executed, then only I get to know the address. So, that is a big problem, and which is why often what you do is, you would not issue a load if it has a store before it. So, you will wait until fifth instruction is gone from the queue, it has executed so that you know its address. You can store the address in a slot in a queue, and this instruction can compare and can figure out, whether you can issue or it has to wait or whatever. Is the basic algorithm clear, what is happening. So, this called Tomasulo's algorithm, after the name of Robert Tomasuloan IBM engineer, who came up with this particular systematic way of maximizing ILP for the IBM 360 machine. It has many different descriptions; this is just one of those many. The book has a different description, which we will go through very soon, but the tracks of it is this, that you keep track of dependencies, you have a table to know who you depend on and soon and so forth.

Now, there are small issues that are left, which actually can be troublesome, which you have to think about. The first question is, let suppose that the instruction that produces. So, currently v 0 is produced by 4, and for some reason what happens is that, this instruction is not fetched, nothing beyond it is fetched. So, this is the last instruction that produces v 0 in a queue, and everything onward is not fetched yet. So, eventually time will come when this instruction will write back. So, at that point essentially what it means is that, v 0 is no longer held by 4,it is in the register file. So, at that time I actually turn this off, this bit goes to 0. So, that is what he has mentioned that when you write

back to register file, you can mark this entry as invalid. So; that means, v 0 is not in queue anymore, if you want to read v 0 its ready in the register file, you can get the value from there. Any question. So, the second question is, it may happen that. So, we say that when you allocate, you read this table, to find out who you depend on, which slot is producing your sources. So, there may be a race going on. It may happen that one of the sources, that you require, the instruction currently being allocated requires, is currently being written back, in the same cycle, that can happen, because in the same cycle many things are happening, something is being allocated, something is being executed, something is being written back

So, an instruction requires register r, which is being allocated in the cycle, and the instruction that produced r; the last instruction which produce r, is being written back in the same cycle. So, this instruction comes looks up the table, finds up the entries one, things that well, it depends on this particular slot, and it goes there, and waste forever, because it is never going to be woken up, because that instruction has been written back. So, how do you resolve this. Is the race clear to everybody, because in the next cycle if you checked, of course, this entry would be 0, but in the cycle that you have getting allocated, there is a race going on. You are being allocated; same register is being written back. Are you remember that the way pipeline works is that, in a particular pipeline stage it will sample all its inputs, and then work on those inputs, during the cycle, and then make the modification at the end of the cycle. So, at the beginning of the cycle it will sample the table, find out that this entry is one, and will think that it is supposed to get the value from 4, how do you solve this problem.

Student: we have to timeout to check (( )).

Time out, you have to be very careful if you rely on time out, because there are, the memory instructions have non deterministic delay, you do not know how long they are going to take. What is this register is being produced by a memory instruction, which maybe a legitimate dependence actually. If you time out two early then… So, yes that works. So, you are saying that periodically why do not we check this particular entry, that make sure that will make forward progress eventually, but we may end up losing a lot of cycles, any better solution.

Student: We divide the semicircle in two parts. First part we do the write back, and second part we actually check this table.

So, you are saying that you want to face the execution.

Student: In the first phase we (( )) and in the second phase we would actually check this table for (( )).

No see the wakeup happens when an instruction completes execution. You want to modify that. So, before I go to his solution. So, what he has suggested is that, why do not we have a phased execution, that write back happens in the first half of the cycle, and my allocation happens in the second half of the cycle, then the problem is solved, because by the time allocate works, the table is up to date with this cycles write back. So, I will always get a consistent state in the table. That is one solution, but if I want a very high frequency processor, I may not be able to accommodate my write back in half cycle.

Student: We can have wake up even after (( )).

You want to wake up after this. before after is very fuzzy in a cycle, when you are within a cycle there is no meaning of before after, these are all concurrent, they can happen at any point in time. So, this one works, so what you do is, you divide your write back essentially into two parts, spread it over two cycles. In a first cycle, you update your table. So, you mark this entry to be 0. So, that is what this write back stage is doing, and if you want you can write the value also to the register file. In a second cycle, you broadcast this entry again, over the queue.

So, in case, in the previous cycles somebody had a race, that guy will now wakeup, and this cycle if somebody is being allocated, it will actually see as 0 value, so it will not wait. So, that takes care of the problem actually. Is it clear, splitting the write back in two cycles. There is only one broadcast per write back. So, when this particular v 0 is written back, in the first cycle in the WB stage, it will reset this bit, and it will write back the value also to the file. In the second cycle it will broadcast 4, actually in this case will broadcast 4 along with the value, if the value will also be needed. So, if anybody miss that bit in the last cycle will wake up now, and pick up the value from the bypass and will go. Clear to everybody.

No problem. So, you said that this wakeup stage is needed, because this will wake up any waiting instruction, which are waiting on some dependents, so that they can now participate in the selection, in the next cycle. So, you can have the same race there between allocate and wakeup. You are being allocated, and your source register is also being generated in this very cycle, you are being woken up actually. So, what will be the downside if I miss this particular wakeup, because there is a chance that I may miss it, because I am being allocated in this cycle. So, what will happen is that. I will go and look up the table, table will say 1 here let say. So, I pick up 4, but in this same cycle 4 is being completed, but I will miss that wakeup signal. So, what I am going to do, I am going to wait there, until this is going to get written back, when I will be woken up, because of that broadcast, of the second broadcast that we have put there. No it will be reset only when it is written back. Yes that is what I am saying. So, I may allocate an instruction in this particular cycle which needs v 0.The instruction which is producing v 0 is also completing in this cycle, and is broadcasting this particular slot id. So, this instruction is going to miss this particular signal, because there is a race going on. In the same cycle you are being allocated, and you are being woken up. No the point is that. So, maybe I should clarify. So, let us go back to this. So, these are ready to go, and this is not yet ready.

So, assume that this instruction is not yet allocated. So, now what is going to happen is that, let us see. So, v 0 is being produced by 1, v 1 is 2fine. So, 1 2 3 are issued, they are executing. They go to the memory stage, they done executing. So, now, they complete. So, essentially I am going to broadcast 1 2 and 3 they are complete now. So, when these broadcast happens in the same cycle, 4 is being allocated in the queue. 4 looks up this table finds and it depends 1 and 2. So, it goes in there, sits there waiting for 1 and 2 which will never happen, because 1 2 and 3 have been broadcast in this cycle. So, this instruction which could have executed in the next cycle, will now wait until these two are written back, when they will be woken up. Is the problem clear to everybody, or you still do not see it. I am saying is that you are now depending on this particular wakeup, to know that you are ready, instead of this one, because you miss this signal. In the same cycle you are broadcasting 1 2 and 3, which all the instruction supposed to compare, against their dependence list, but these instruction is not yet here, to compare anything against. It is still being allocated in this particular cycle.

So, you look up the table, will get to know its dependence correctly, it will mark 1 and 2.Will mark itself as not ready, and will wait there, until these two are written back, when they will be again broadcast, and now they will pick up based on this. So, there will be some lost cycles which we actually do not need. Is it clear, how do you solve. This one more bit in the table. You want a ready bit. How does it help. When is this modified. As soon as it issues, or when it completes execution, which cycle. While execution may not be one cycle, it may multiple cycles last cycle. So, then what it is a difference I am that point I broadcast, the wakeup also goes out. I see yes. So, you want to do this one cycle early. So, will be actually doing the same solution, we are going to split up the wakeup in two cycles. So, we are going to change the table entry here. So, it is a ready bit which says that, whether this particular value is ready or not. It may not be ready in the register file, but whether is ready in this particular slots or not, queue slots.

So, whenever the instruction executes, I may not be able to do it in the same cycle, maybe the next cycle. I will first change this table entry, to say that; yes this register is now ready, meaning that the value is ready. And then in the next cycle I will do the wakeup. So, what will happen is that, in this cycle if somebody misses this, will be woken up in the next cycle, and the next cycle if anybody comes, will actually get the correct value from the table, and all of them can now participate in the selection in the next cycle. So, essentially what have done is that, I have introduced one more cycle here. So, I will just call it wakeup 1 and wakeup 2. So, essentially what I am doing is that, I am essentially splitting the wakeup into two state changes.

So, there are two states associated with the wakeup now, or write back. So, that gives you essentially what; an 11 stage pipeline. So, these are pretty much a functional design, these are going to work, and the only limited to your ILP is the, length of the queue, how many instructions you can see at any point any time. If you have more, you will discover more independent ready instructions, which can participate in the selection. And of course, you are limited by your register file ports, you are limited by your number of functional units, how much can be issued. Also we have not discussed one thing; that is you are limited by the rate at which the queue can drain, and that depends on what, write ports, the number of write ports in your register file, because if I want to drain three entries every cycle, I better have three write ports in my register file, because I may have to send three values to register file.

So, if we have an unbounded queue, you will be limited, and if you have unbounded register file ports, if you have unbounded functional units. You will be limited only by flow dependence, nothing else, only data flow will be the limiter. You will get all other ILPs possible in your program. And you notice what is going to happen with this particular piece of code. Your queue will actually fill up, even though the 7th instruction is a branch instruction. Assuming that this branch predictor here, which will be looked up here actually whenever the 7th instruction is fetched, and if it tells you the right thing, will actually keep on filling the queue in the right direction with right instructions. And then your selection hardware will actually in this case, although it may not be possible because of this dependence. These two loads will depend on this particular one, and then you have various other dependent instructions, but otherwise nothing stops you from picking up ready instructions from different iterations. Any question. So, your book gives slightly different description of this. It has a different organization of this queue, and it also maintains this particular table in a slightly different way, and will actually. So, by the way this is actually not implemented in any processor, in this way ever.

Can anybody guess what is the problem? So, here I have a single queue, length of queue yes. I want a very large queue, so what cannot I have one large queue.

Student: It is not bounded.

No it will be bounded to some length, let us say 200 entries.

We have a long queue in the broadcast (()). We have to. All of them have to check. You have to update all the entries, that the last value.

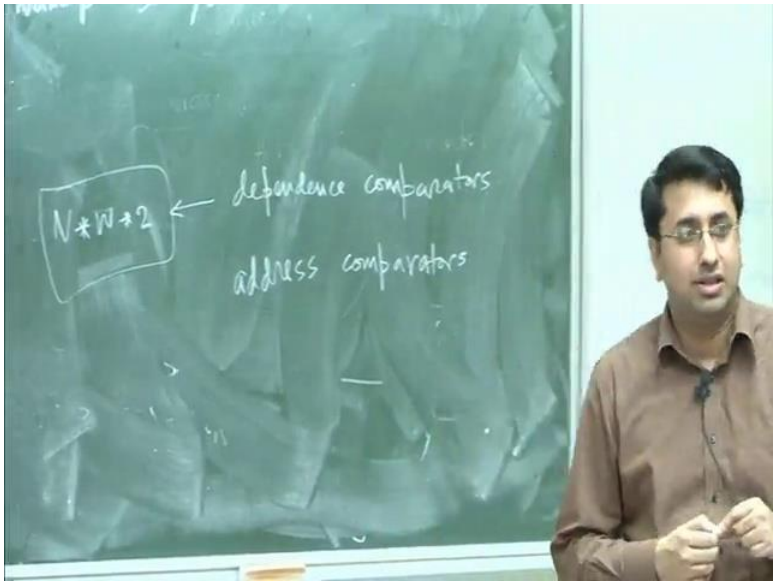
So, how many comparisons.

Student: number of entries into maximum number of dependencies.

How can yeah how many. Why is it.2 into number of entries. So, per entry I can do at most two comparisons; that is the worst case, multiply by number of entries. So, in the worst case every cycle in a 200 entry queue designed, I will be making 400 comparisons every cycle. And possibly to accommodate this 400 comparisons in my short cycle time, I will be doing all of them in parallel. I have to do that, which means I need 4 hundred

comparators, in my particular design, which is probably out of question. , but per entry can make only two comparisons.

(Refer Slide Time: 48:56).



Compare with a every broadcast id exactly. So, what is the number. Suppose I can write w entries in a cycle; w into n into 2. So, does everybody see why is that, whenever I broadcast one entry that will be compared against both of these for every entry. So, n into w into 2.So, that is pretty large actually, very large. So, how do I solve this. I cannot get rid of the broadcast; that is fundamental to this design actually, that everybody needs to compare, that cannot be taken away. So, number of comparators. Can it be reduced. There is more, other than this wakeup, there are other comparisons, involving load store operations, I have just mentioned. So, suppose that I have a constraint that load instructions, will not issue, until all the stores before it have completed, so I have these particular constraint. So, whenever a store is issued what it does is that. It does not write the value to memory, but it computes the address. It goes to the execution stage computes the address and stores it in the separate filled in this particular entry.

So, whenever a load instruction wants to issue, it first takes that all the stores have been completed before it, that is the first constraint. And let us suppose that yes they have all completed, but there are bunch of stores before it actually, which have not yet written to memory. So, what it will do is. It will issue compute this address, and then broadcast the address over all the queue entries for comparing against the stores which have

completed. And if anybody matches, the load has to be called back. The load has to be called back, it will take the value from the stores entry. It cannot take the value from memory. Is that clear to everybody, because load is supposed to consume this value, not something that is in the memory. So, that actually increases your number of comparators even further, you need address comparators. So, these are queue id comparators. So, these are dependence comparators. In addition to that you need address comparators. So, what do you do, how do you reduce this overhead. The storage is not a problem; a 200 entry queue is ok actually. So, you want to list of dependence here, as oppose to storing who I depend on, you want to store who I need to source, but this list maybe unbounded. So, how do I size it then, because this has to be done in the silicon which will be fixed at design time. You cannot increase it at runtime saying that, I need more.

Student: (( )) size of the (( )).

Which is gigantic? Yes hash table, will that reduce your number of comparisons. It will reduce w.

Student: it is the organization of a table.

How, what is the organization of a table. No what is my hash keyids. Tell me the organization of one entry, what does it contain, all this things same. So, then what is the difference. So, we will continue from next time. So, I will stop here today. Based on the register; means. Can you little bit more. So, you want to attach a list of dependence with the register, but the list of dependence is unbounded, I mean not unbounded, but its. So, that is still large. You mean so many fields, not too many. See I am saying storage is not a problem actually. The problem is a operation that you are doing on the storage that is the problem. You have gigantic cashes on that is not a problem really. The problem is a kind of operations you are doing. So, anyway we will continue from next time. We will see how today's processors actually handle this problem, but keep in mind that this is the model. Now, essentially we will do small tricks on this, to make sure that this becomes implementable.