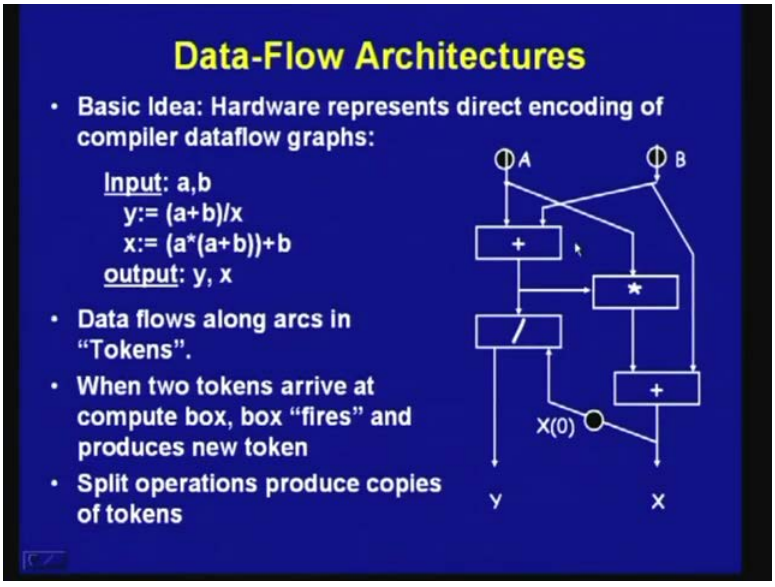
High Performance Computer Architecture
Prof. Ajit Pal
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 18
Dynamic Instruction Scheduling with Branch Prediction

Hello and welcome to today's lecture on Dynamic Instruction Scheduling with Branch Prediction. In the last two lectures, we have discussed various techniques of branch prediction and earlier, we have discussed the dynamic instruction scheduling. Now today, we shall see, how these two can be combined to achieve higher performance in a processes.

(Refer Slide Time: 01:27)



But, before that, let us have a quick recap of, one very important concept that is, data flow architecture, which is used in dynamic instruction scheduling. Earlier, data flow architecture were proposed with basic idea that, hardware represents direct encoding of compiler dataflow graphs. So, for example, you have to perform this computation y is equal to a plus b by x and x is equal to a into a plus b plus b and inputs area and b, and output is y and x.

Now, one can create, in data flow architectures, a data flow graph is created, so here this is the dataflow graph corresponding to this computation. Here, you can see, a and b are the inputs and y and x are the outputs and various operations like addition then, your

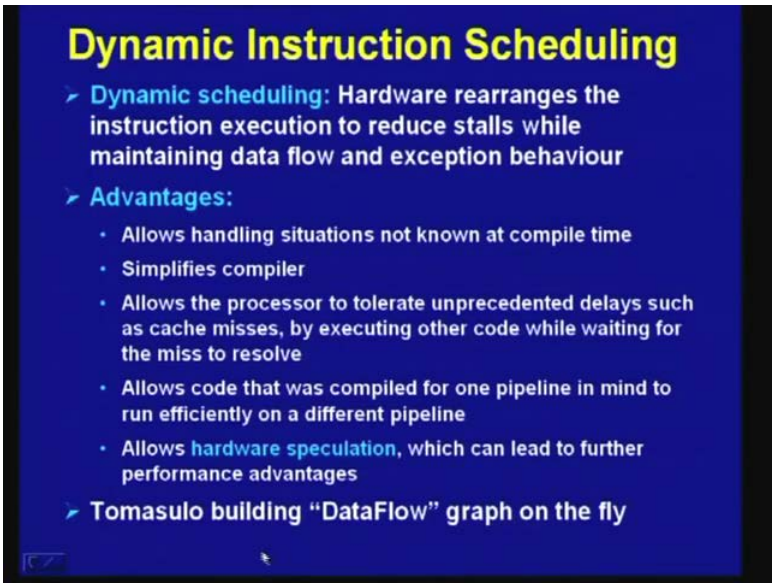
multiplication and this division y and x and this addition. So, you can see, all these are given here, now what is the basic idea of the dataflow architecture. So, dataflow along arcs in tokens, you can see here, two circles are given, these are known as tokens.

When both the tokens are available then, when two tokens are arrive at the compute box, so this is your compute box, adder is a compute box and then, the box fires and produces new tokens. So, that we seen, there are a two tokens I mean, if there is a token on this arc, if there is a another, there is also a token on this arc then, this compute box will fire and produce a token at the output. And if their a spilt operations like this then, copies of tokens are produced that means, here for example, A and B tokens of their.

So, this box will fire, produce a token here and that token will be available at this input on this arc as well as on this arc. Now, we can see tokens on A and tokens on this arc are available, so this multiplication multiple I mean, multiple compute box will fire and it will produce a token here. Now, we can see, token is present on this arc, token is already present on this arc, so this add execute unit will fire, compute unit will fire. And as the token is available here and earlier, there was a token here, now tokens are available both these arcs.

So, this particular compute unit that is, divide will fire and finally, it will produce a token here. So, you see, this is how, the dataflow operation of different computations are controlled with the help of tokens in dataflow machines. Somewhat similar thing is done in dynamic instructions I mean, scheduling and particularly in tomasulo's approach, what is being done, this dataflow graph is implemented, while executing instruction and hardware does it.

(Refer Slide Time: 05:07)



That means, this dataflow graph is created within the hardware, with the help of reservation a stations and other functional units to do it. So, this is one very important concept and we have already discussed about this dynamic instruction scheduling, hardware rearranges the instruction execution to reduce stalls, while maintaining dataflow, you can see dataflow and exception behavior. Here, as I mentioned, I have briefly explain the operations of the dataflow architecture, same concept has been borrowed here.

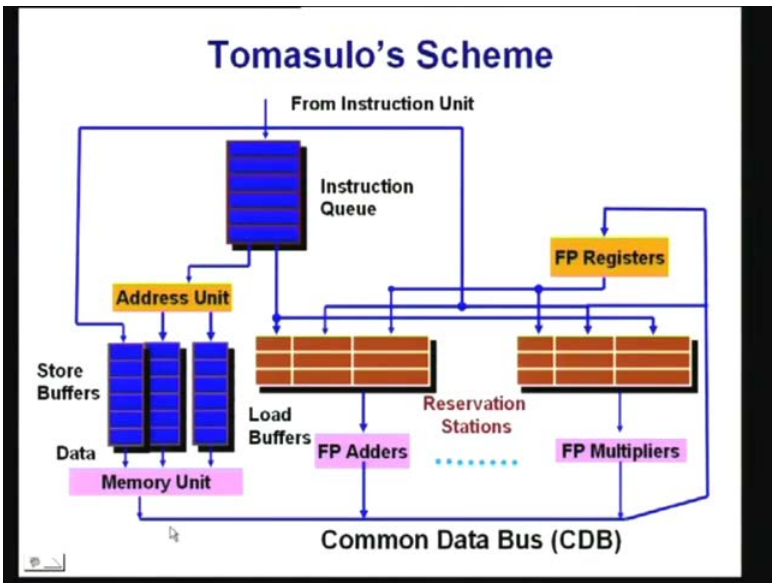
And of course, it also maintains exception behavior, which is also important and it has got many advantages. As you have already discussed, it allows handling situations not more than compile time that means, it is better than compile time instruction scheduling and it simplifies the compiler. It allows the processor to tolerate unprecedented delays such as, cache misses by executing other code, while waiting for the miss to resolve.

Because, it allows you, how to code for execution, so if some instruction, if it is delayed then, other instruction, which is out of order can proceed, particularly in processors in the in the early years, when cache memory was not present or even when cache memory is present, it may be delayed. So, if cache memory is not present, if it is a memory instruction, it will take longtime, on the other hand if there is cache memory, if there is cache miss, it may take longer time, because of cache miss.

So, this type of problem is resolved that means, executing other codes while waiting for the miss to resolve. So that means, the different delays of different instructions are tolerated then, it allows code, that was compiled for one pipeline in mind to run effectively on a different pipeline, as elaborated earlier. Now, in my next lecture, I shall discussed about another very important technique that is, hardware speculation. You will see that, hardware speculation goes hand in hand with dynamic instructions scheduling.

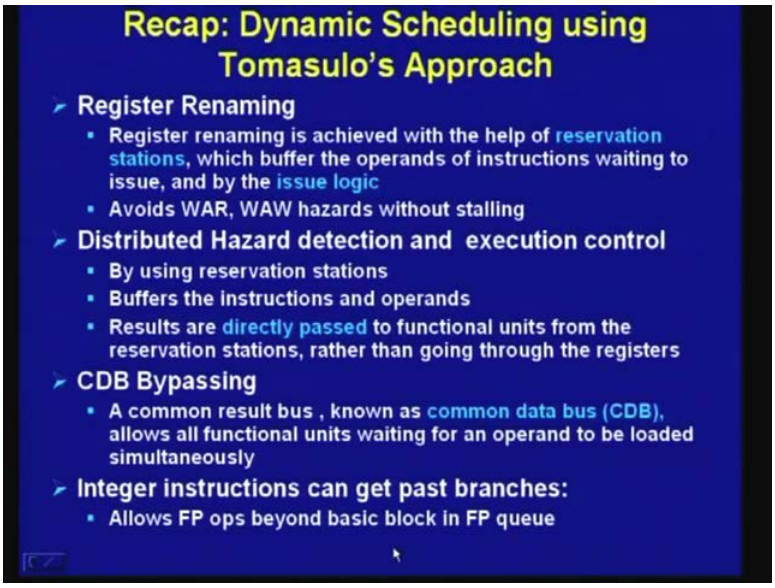
So, this dynamic instruction scheduling along with hardware speculation, which I shall discussed about in my next lecture, which can lead to further performance advantage. And particularly, as I have already mentioned, tomasulo's building dataflow graphs on the fly. So, that means, this dataflow graphs, which is created here is created on the fly in your tomasulo's scheme.

(Refer Slide Time: 07:52)



And we have already discussed about the basic structure that you require to implement tomasulo's scheme. You have been instruction queue then, these are the floating point registers, when innovation is the availability of reservation stations as we have discussed and you have got load buffers and store buffers.

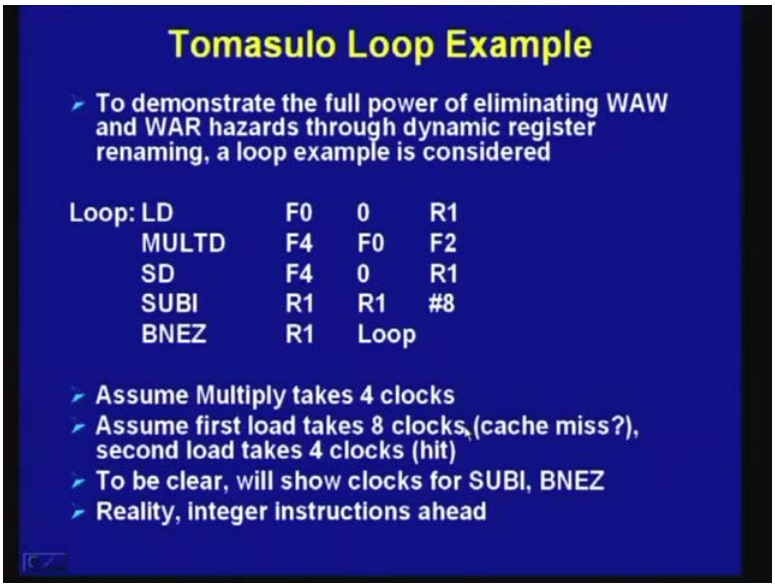
(Refer Slide Time: 08:20)



And the main features is register renaming, that is achieved with the help of reservation stations, which buffer the operands of instructions waiting to issue and by the issue logic that you are already discussed and this avoids WAR, WAW hazards without stalling. And distributed hazard detection and execution control by using reservation stations, buffers the instructions and operation cooperands that means, with the help of this buffering, you can pass the results directly to the functional units from the reservation stations, rather than going through the registers.

And this common database bypassing is done, because the result is broadcasted and the common database, which is received by all the functional units waiting for the operand to be loaded simultaneously. So, these are the key continuation of tomasulo's approach and another thing is, integer instructions can past branches. So, this allows floating point operations beyond the basic block of FP queue and this can be done provided, we have got branch prediction, only with the help of branch prediction, this can be done as we shall see.

(Refer Slide Time: 09:50)



So, let us illustrate this with the help of tomasulo's loop example, this is a loop example, earlier we have seen how dynamic instruction scheduling can be done over the basic block. And you can increases the size of the basic block by loop unrolling and all those things, but now you will see that, loop unrolling will be done dynamically with the help of branch prediction, so this is the example.

And these are the assumptions we shall make, assume multiply takes 4 cycles, assume first load takes 8 cycles, because of cache miss and second load takes 4 clocks. Because, as you know, later on I shall discussed about this cache memory and you will say that, whenever there is cache miss trying to read a particular word, if that particular word is not present, the entire block comprising several words had transferred to the cache memory.

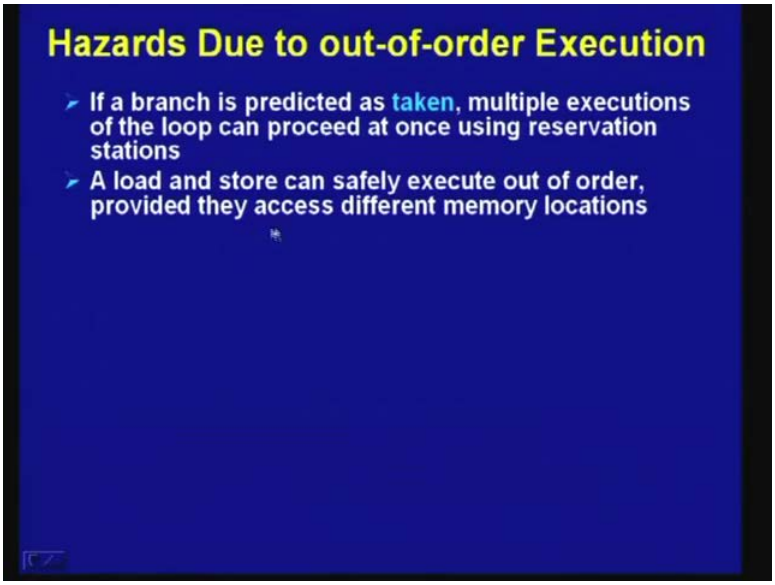
And as a result, subsequently if you read another word from that block then, it will lead to a hit. So, first time there will be a miss, subsequently there will be hit, so that is what is explained here and later on, we shall discuss in detail, how the cache memory really works and this terminologies like cache miss and cache hit. But, for the time being, let us assume that, first load will take 8 clock cycles, because of cache miss and second loads will take 4 clock cycles.

And also we shall see, although I mean, for the execution of this, we shall show clocks for these two instructions, which are essentially used for the purpose of housekeeping for

loop computations, you know you have to subtract R 1, which is incremented I mean, R 1 is incremented gradually, which is acting as a pointer to different elements of an array and this branch is done by checking care of, how many times the loop will proceed or not.

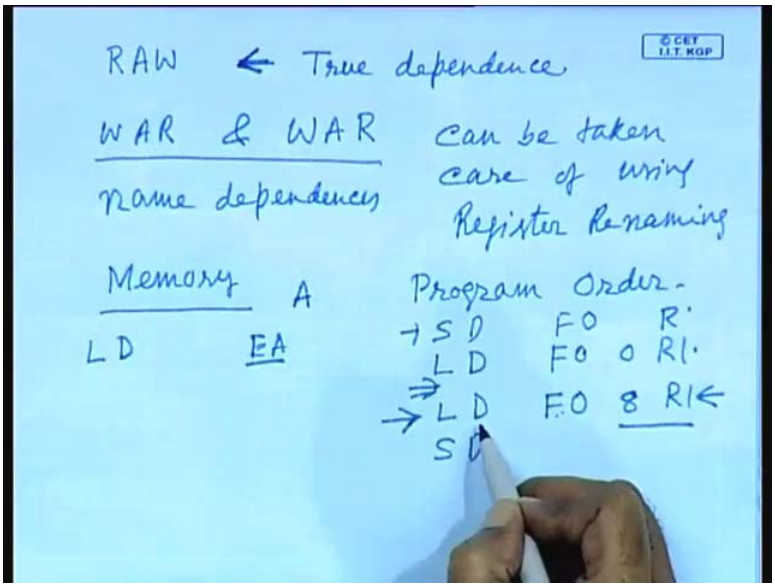
So, by creating it to 0, taking it whether it is equal to 0 or not, so these two was the clocks, for these two will be also shown. And we shall see that, integer instructions will proceed ahead compared to the floating point, but we shall primarily focus on floating point of a reasons.

(Refer Slide Time: 12:51)



Now, hazards due to out of order executions, if a branch is predicted as taken, multiple executions of loop can proceed at once using reservation stations. So, normally the execution is restricted to a basic fire block but now, if you know that a branch predicted that means, that loop will be taken I mean, that branch will be taken loop then obviously, the instructions of the other iterations can be carried out. So, this is the basic idea here that means, if a branch is predicted as taken, multiple executions of loop can proceed at once using reservation stations, how we shall discussed later. And a load and store can several execute out of order, provided they access different memory location.

(Refer Slide Time: 14:06)



So, this is another thing we have to take care of, we have already discussed about the read after write of hazards, which are essentially arising out of true dependency, which will be taken care of by tomasulo's algorithms. We have already seen how it is being done, now there is other two hazards, right after read and write after write. If they involve registers that means, if the read after write and write after write operations are involving registers, that can be taken care of by register renaming, that we are already discussed.

And in tomasulo's algorithm, this register renaming is done dynamically without actually involving registers, that we are discussed in detail. Now, question naturally arises, if this hazards arise involving memory then, how it can be taken care of. So, write after read and write after write hazards I mean, arises out of name dependences. If they arises involving memory then, you have to take care of, you may have a series of load and store instructions present in a program.

So, the way the sequence, in which they appear in a program as you know, this is known as program order. Now suppose, you have got, this is one load and you may have another load say, let it be F 0, 0, R 1 and there can be load and store, there can be several other load and store before and after hit. Now, let us consider load and this load, let us consider and suppose, this is also F 0 and it is involving a memory. Let us assume here, it

is some something 8 R 1, but in between the value of R l may have changed, which have not let me here.

Now, if these two involve the same memory then, from the same memory we are loading then, what can happen. So, if you change the order, this will lead to a hazard, which is known as, which is essentially write after write hazard, so this is a write after write hazard, you are writing in a same register. So, how this type of hazard can be overcome in tomasulo's approach, so this is you are the reading from the same memory and you are from a particular memory on writing into a register, you are reading from memory, writing into same register.

So, in this case that means, it is involving a memory, but it is writing into a same register. In that case, if this is executed fast and then, this will lead to write after write hazards, how can it be overcome. This can be overcome in this way, so whenever this instruction is executed I mean, it encountered then, the effective address is calculated. So, effective address of this load, of a particular load is calculated and that is available, as you know in the A field, address field of the tomasulo's that the data structure, which is available for the reservation stations.

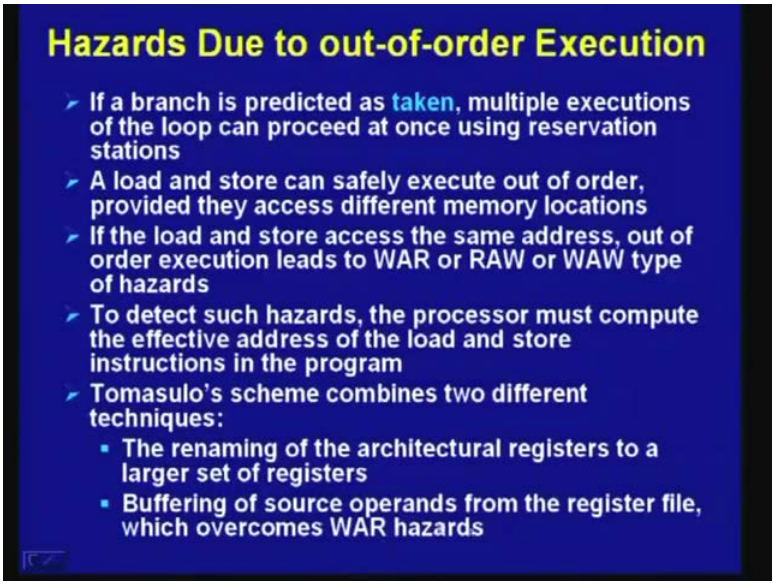
Now, whenever the then, after knowing the effective address of particular load, it checks whether there is any other load that is present earlier, which is active that means, which is not yet been carried out. Then, what you have to do, this particular load has to be delayed that means, it will not be loaded into the load buffer. So, there is a load buffer and store buffer, so the instruction following a load using the, having the same effective address is not loaded.

So, this is how it has been taken care of and similarly, there is a store data, again it is involving same register. So, you are storing the data into a memory from this register and if this is executed fast and then, you are loading it and then, you are loading it, storing it into a memory location, so this will lead to again hazard. So, here also that means, if the load is succeeded by store and load instructions, which are appearing having the same effective at this then, that particular load has to be delayed.

So, similar situations will happen in case of store instructions that means, you have to look at the calculate the effective address, which is precisely stored in the A field. And that effective address has to be compared with other effective addresses, preceding

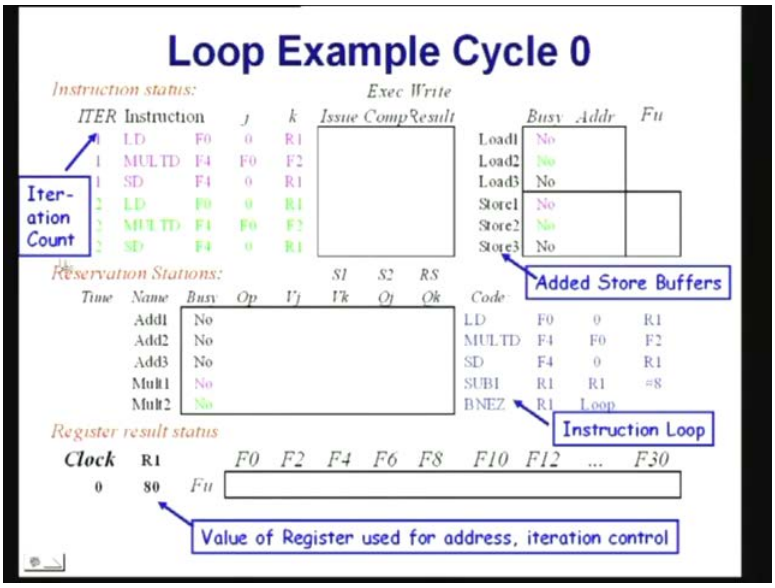
though that load and store instructions, which have not yet been carried out. But, active that present in the reservation stations, so this is how, you can take care of the load and store.

(Refer Slide Time: 21:00)



So, if you load and store access the same address, out of order execution leads to WAR or RAW or WAW types of hazards, particularly WAR or WAW types of hazards that you already seen. To detect such hazards, the processor must compute the effective address of the load and store instructions in the program and tomasulo's schemes combine to different techniques. The renaming of the architectural register to a large set of registers, buffering of sources operands from the register file, but this is essentially corresponding to registers, but for memory b we have to do it this way.

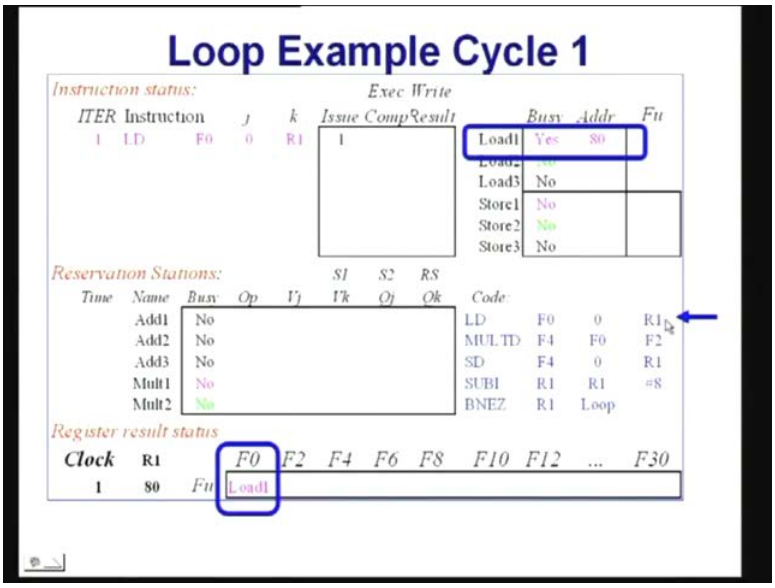
(Refer Slide Time: 21:43)



Now, let us consider the loop example that I have already mentioned and you can see here, here you have got two iterations present to name instructions two loops present in this instruction window and this is the code to be executed, which is present here. And initially, this register R 1 is having the value 80 that is, the memory location that is being available I mean, from where it has to be loaded is 80. Now, with this starting point, let us see, how the execution will proceed.

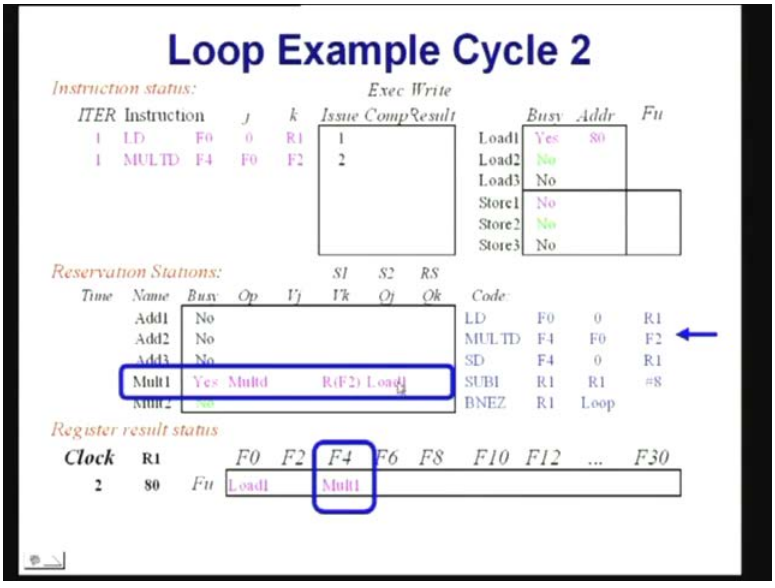
So, these are I mean, various things which is store added store buffers, load and store buffers are shown here, instruction loop is given here, iteration count is given here. This is the first iteration, this is the second iteration and value of registers used for address in the iteration control is given here.

(Refer Slide Time: 23:02)



Now, let us go to clock cycle 1, so it will issue the first load instruction and you can see the load unit is busy and the address is, the effective address is 80, that is available. And the register, in which it to be loaded from I mean, from the load instruction is given here and this is pointing, this load is being issued, this instructions is being issued.

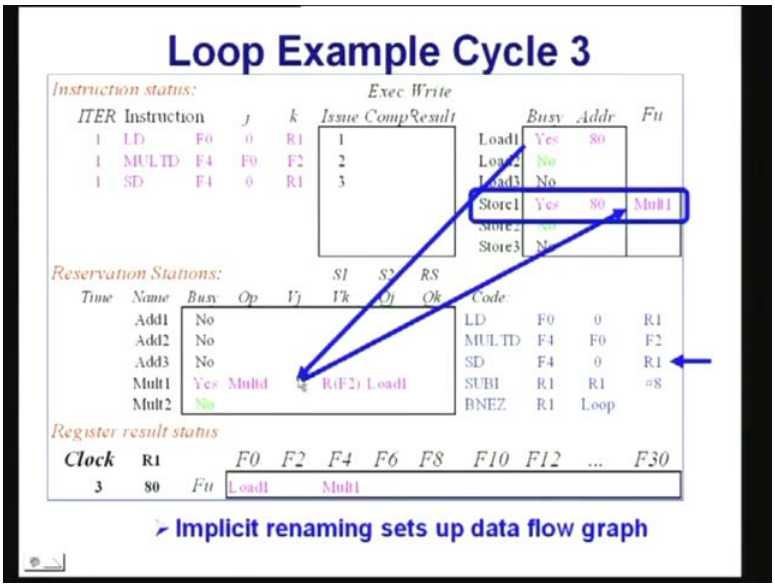
(Refer Slide Time: 23:32)



Now, we go to the second clock and the second instruction is the multiplication double, that is being issued and necessary data structure is updated. And this shows that, this particular instruction has been issued and here, the busy then, operation then, these are

the I mean, values from where it will get. And here, you can see, the value will be taken from read F 2, R F 2 and that will be that means, the register, it will be from the register. R F 2 means, it will be coming from register F 2 and here, the one operands will come from load 1, this is F 0 will come from load 1, so this is been provided here.

(Refer Slide Time: 24:28)

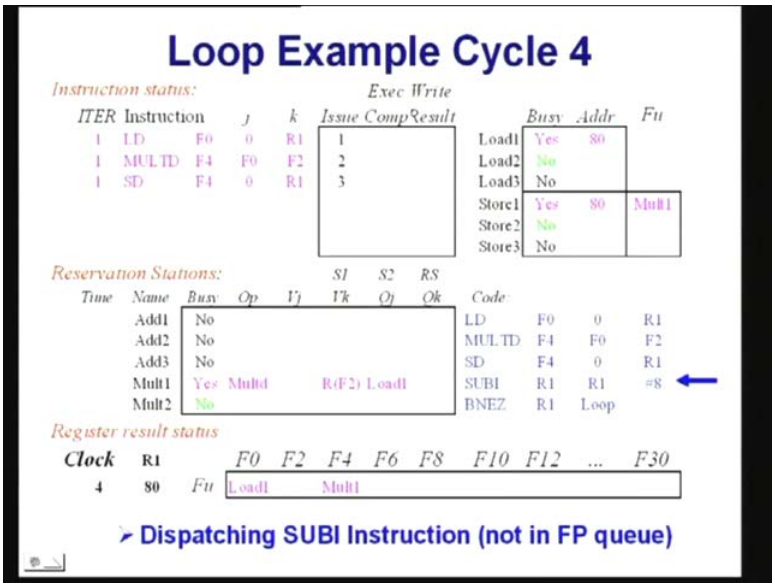


And now, we go to the third cycle and here, this store instruction is issued of the first iteration and again the corresponding data structure is updated. So, this busy that means, that store unit is (Refer Time: 24:47) busy at this correspondence address is 80 and effective address is 80 and that functional unit involved is I mean, from where it will come, so F 4 will come is multiplier 1 that is, in the multiplier 1, so that is been showed here.

And here, there is no other change then, it will go to the outside, so here implicit renaming sets up dataflow graph that means, implicit renaming means, here you can see, that loading I mean, implicit renaming means, he will not really perform any renaming of these registers that means, whenever you go to the second iteration, but it will be done implicitly, because the information will be taken, data will be taken from the reservation stations, that we shall see.

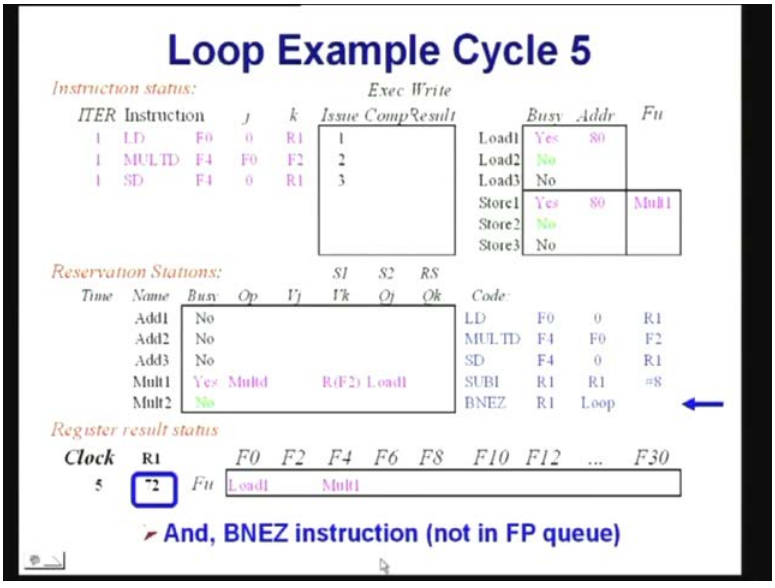
So, you can see here load, it will come from load and again it will be provided here that means, it F 0 will come here and it will be provided here also for this store. So, you can see that means, both the things are done here.

(Refer Slide Time: 26:21)



Now, it will a dispatching this SUBI instruction, but it will not go to the floating point queue. So, here it shows the floating point operations, that queue, but that subtraction immediate instruction, it will not go to the floating point queue, but it will be executed, so this will lead to change the value of R 1.

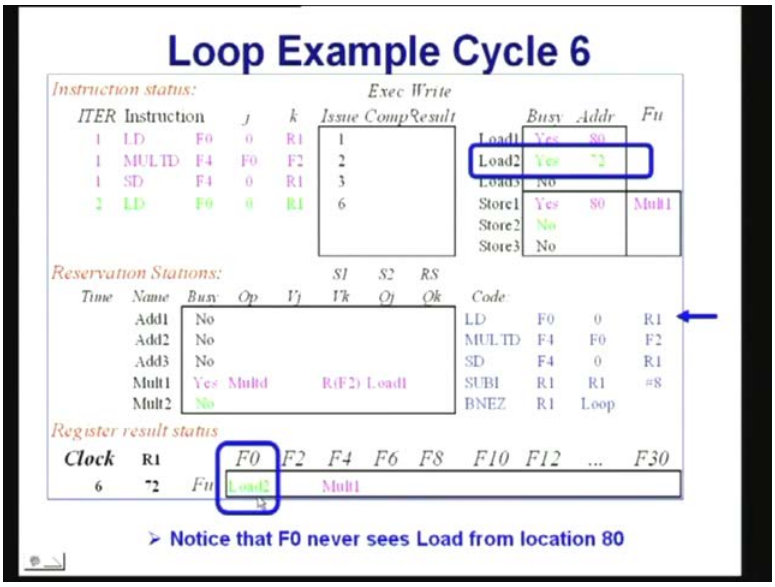
(Refer Slide Time: 26:48)



So, that is what is happen that, now that register R 1 has been changed, the value has been changed, it is now 72, 8 has been subtracted from it. And so, this instruction is

executed then, this instruction will executed and it will go to the second iteration. So, this instruction is getting executed.

(Refer Slide Time: 27:14)



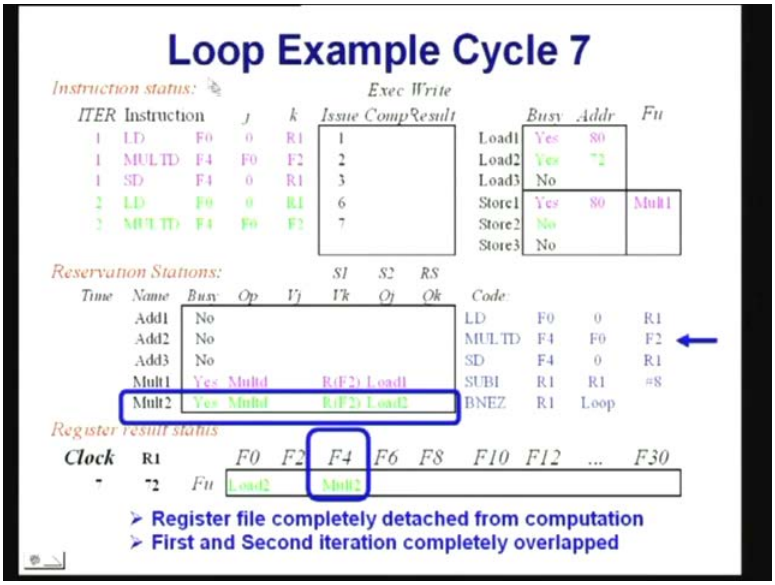
Now, it has gone to the second iteration, so you can see, there was two cycles missing here, 4 and 5, which is missing here, before the second iteration starts. So, before second iterations starts, the two cycles are not present here, because the calculation of the effective address for the second iteration was necessary and then, computation of the condition. So, this condition is being performed and then, it will issue the load instruction now, in the sixth block cycle.

And accordingly, this load unit will now becoming busy, but here the effective address is 72, not 80 and as it is shown here. So, although you can see, to load instruction have been issued and, but they have not completed their operation. Now, you can see, how it is happening here, earlier here it was written load 1 that means, this floating point register was supposed to be loaded from the operation of load 1. Now, it has been changed to load 2 that means, the before loading was done, the load 2 is written here that means, the F 0 never sees the value that is loaded by this load 1 instruction.

Because, this operation is not carried out, it is the information is stored are available in the reservation stations, but it was not written into the registers. Before it was written into the registers, the another load operation has taken place, which will load into the same register, that is why this has been modified. So, this result writing, before it

proceeded to result writing, that load instruction has not completed writing result. So, before that, this has happened, so this will not see the load operation and it will proceed to the next cycle.

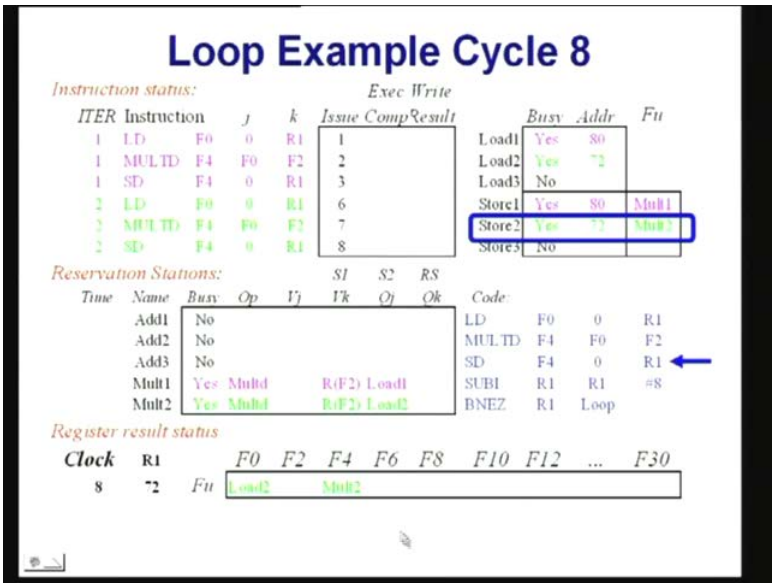
(Refer Slide Time: 29:28)



In the 7 th cycle, again the second multiplication operation is issued, because you have got two multiplier, so the second multiplication operation is issued and necessary updating of the data structure is available here. So, here it will lead one operand will come from register F 2, that value is written here, is available in V k and after load is performed, this value will be available here. So, load 1 and load 2 will provide the I mean, this shows that, this operant will come from load 1 and load 2 and value of course, will come here in V j.

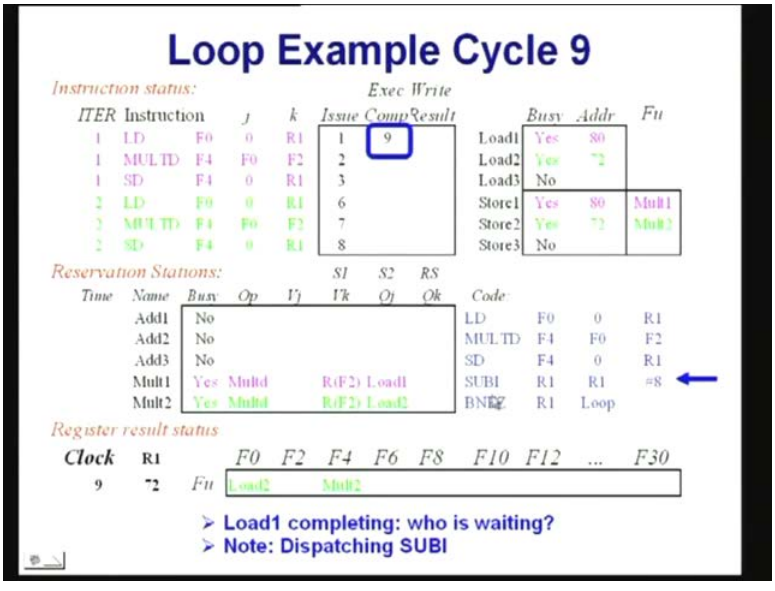
So, this instruction has been issued, now it will go to register file completely detached from computation, because that computation will proceed without involving the registers. Now, we have come to, so you can see here that, first iteration and second iteration, both the iterations are now getting over left. They are getting over left that means, the instructions of the both the iteration have been issued although that completion has not yet taken place.

(Refer Slide Time: 30:54)



And now, this store instruction will be also issued, because it is, there is a store unit available and store instruction is now issued. And accordingly, the corresponding address is modified here, it will come from 72, effective address is 72. So, you can see, there is no hazards so far, present in this.

(Refer Slide Time: 31:18)

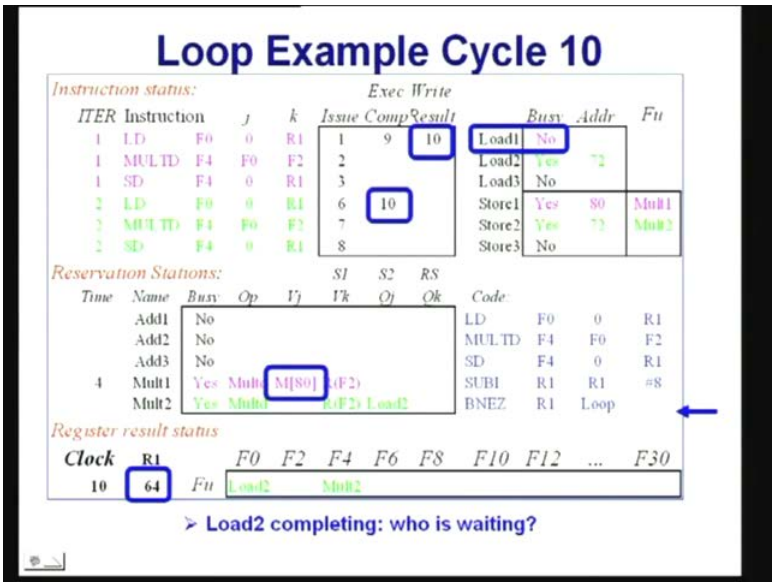


Now, it will perform this computation, that load operation it will continue, it will perform the, it will complete the load operation and because we know that, this load instruction, first load will take 8 cycles, that was our assumption. Because, this was a

cache miss, so it has taken 8 cycles, it was issued here and it has taken 8 cycles. So, in a 9 th cycle, the load operation is completed and now, the value will be available in the I mean, in the corresponding reservation unit and it will appear here, we will see.

Load is completing and obviously, this multiplier 1 is waiting for the data to be available here. Now, it is dispatching this instruction, you can see, it will again now dispatched this one. That mean, it will corresponding to the, for this is instruction I mean, for the second iteration, this will continue.

(Refer Slide Time: 32:33)

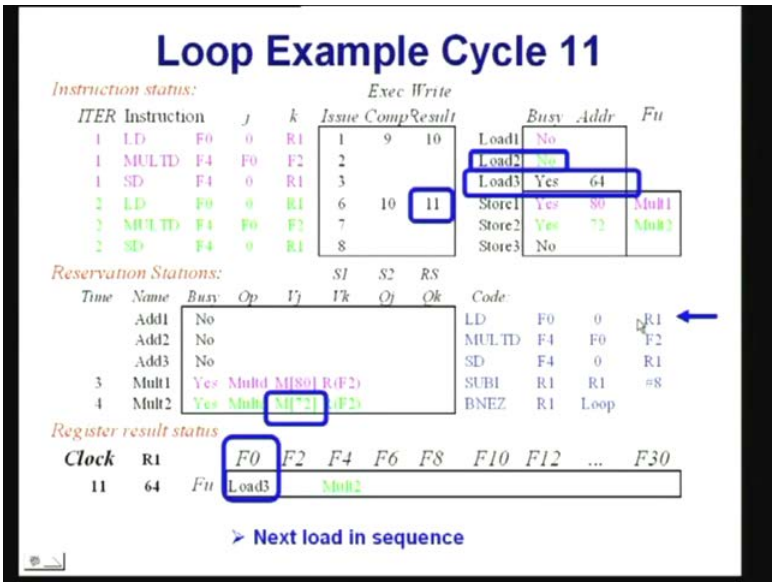


And now, here you can say, as I was telling that, load has now completed and corresponding value is transferred directly to this registers v j, v j is a register which is available as a part of reservation stations. Now, both the values are available and obviously, in the next cycle, this multiplication instruction will start computation. So here, 4 is written here indicating that, the computation has started and it will require 4 cycles to perform the computation.

And in the mean time, as you can see, it is now prepared for the third iteration, because R 1 has been modified corresponding to the third iteration. So, the first iteration was corresponding to I mean, the effective address was 80, for the second iteration effective address was 72 and for the third iteration, it is 64. So, it is already (Refer Time: 33:32) for that and load 2 is completing now, because it was started in instruction cycles 6, so it

will take 4 cycles. So, in the 10 th, 6 plus 4, 10 cycles, this load will completed and it will dispatching this instruction.

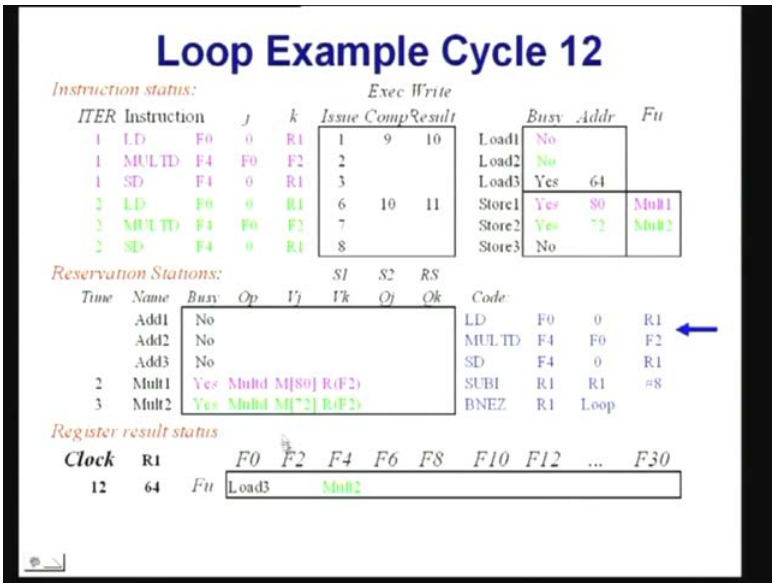
(Refer Slide Time: 33:55)



And you can see, this load is completing, so result is been written here, so when result writing will take place in the 11 th cycle and here, the operand is the available that means, result is available means, it is providing to this multiplier 2. This load will provide multiplier 2 and it will directly go to the reservation station buffers and it will be available here.

Now, here you can see in the meantime, that F 0 has been updated to load 3 that means, again this result, which is coming from the second load, will not be written into the register. Because, already other instruction can be issued, which will provide the operand to the register F 0. So, this will continue in this way, so next load in sequence is coming, for third iteration has started now, so third iteration has started next load in sequence.

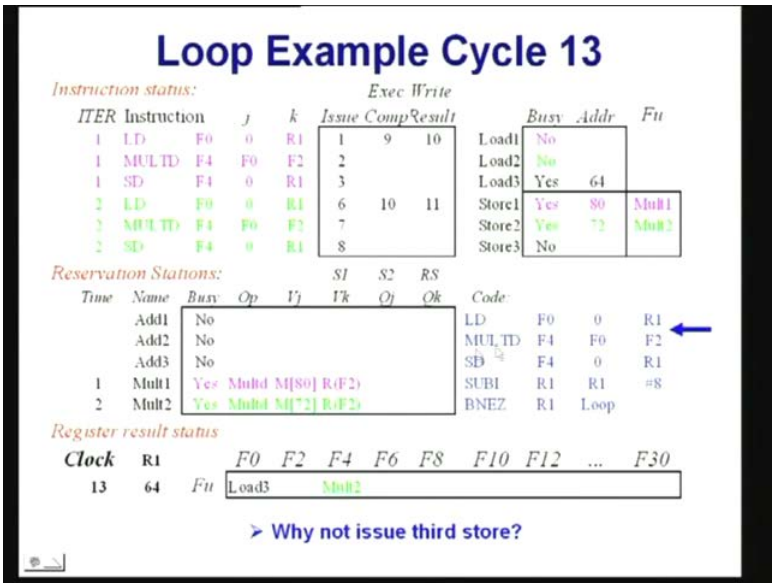
(Refer Slide Time: 35:02)



Now, question arises, will it be able to issue, you see although instruction can be issued, but it is not shown here, because this instruction window, we have not updated, it remains the same. Although this instruction has been issued, this third iteration instruction has been issued, now you can see, this multiplier, this multiplication operation, can it be issued. The reason for that I mean, it cannot be issued, the reason for that is, both the multiplier are now busy.

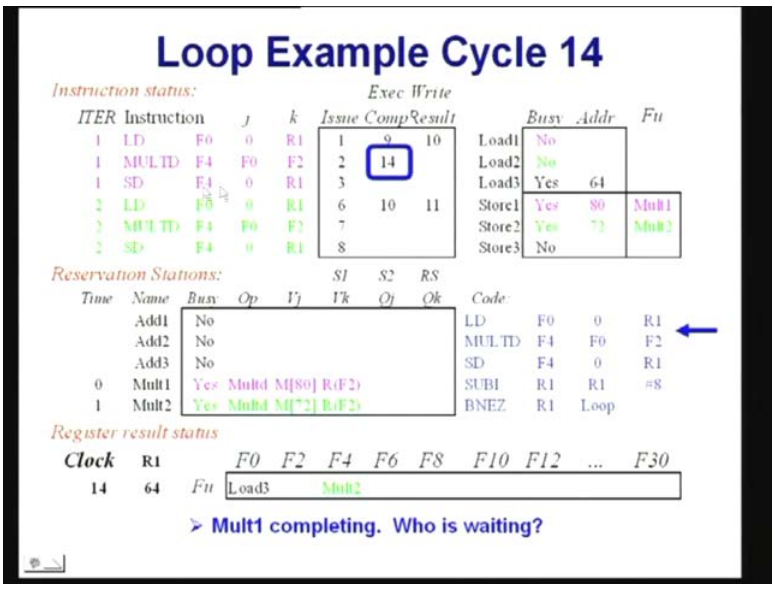
Since both the multipliers are busy, you have got two multipliers, so this instruction cannot be issued, until the multiplier the one of the two multipliers becomes free. So, as a consequence, although the third iteration has been reached, but this multiplication operations cannot be issued, because of structural hazard.

(Refer Slide Time: 36:07)



Now, here the third store, so these third store cannot also be issued, why it cannot be issued, the reason for that is, you have seen the tamosulo's approach performs in order issue that means, see multiplication cannot be issued, no other subsequent instruction can be issued. And as a consequence, since multiplication operation has not been issued here, cannot be issued, this third store cannot be issued also.

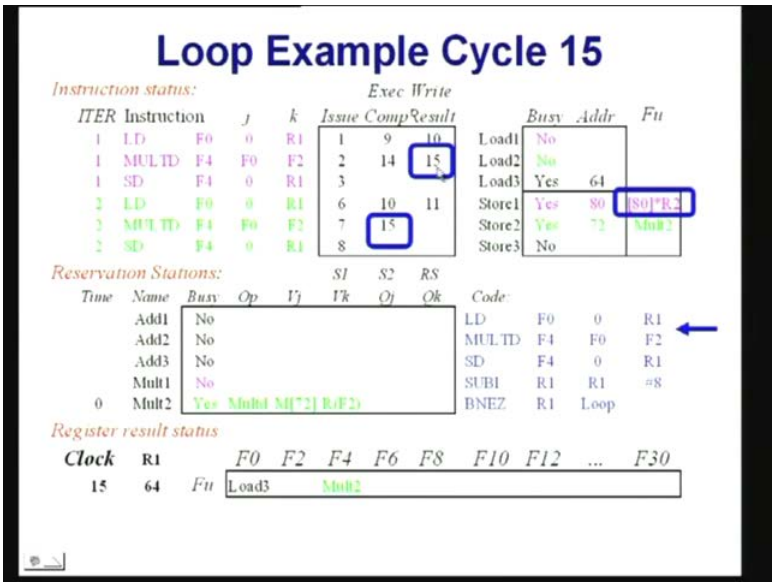
(Refer Slide Time: 36:55)



Now, this multiplication operation is getting completed, it will be performing this, multiplication is complete. So, and multiplication operation, that result will go to F 4

and, F 4 and this store data, this instruction is waiting for the data, so in the next cycle, it will do that.

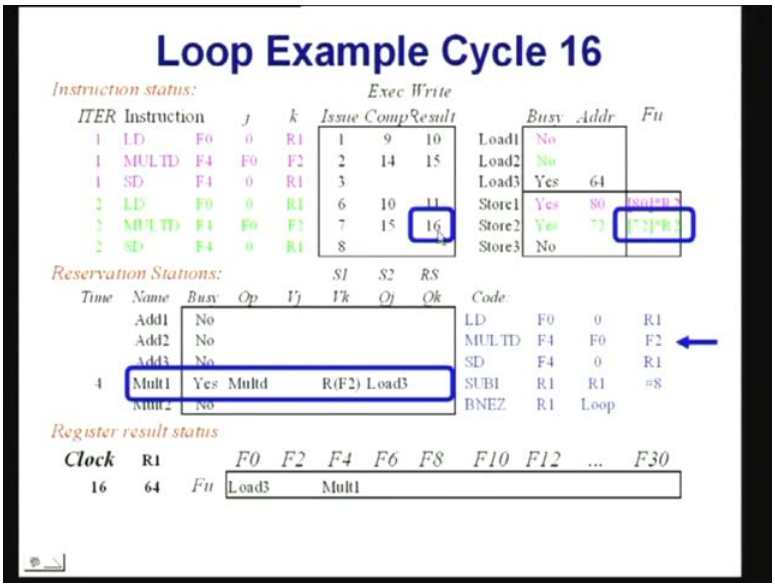
(Refer Slide Time: 37:23)



So, you can see, it has started this, it has provided the result, that result will be used I mean, by the store data instruction to store the result, but that will start in the next cycle. After it is completed and in a meantime, the second multiplier operation will be also completed. You can see, just in one cycle difference, both are completing, because here it was started issued in 7 cycles, but result were available, see in this particular case, it has taken 10 plus 4, here it is 11 plus 4.

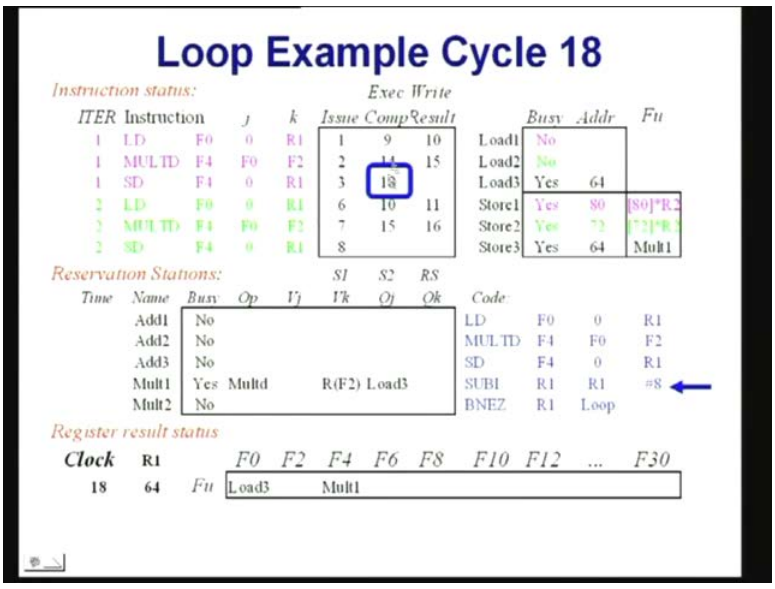
So, multiplication operation is requiring 4 cycles, so that is why, here it is completing in 15 cycle and this will completing in 16 cycle, as you shall see.

(Refer Slide Time: 38:29)



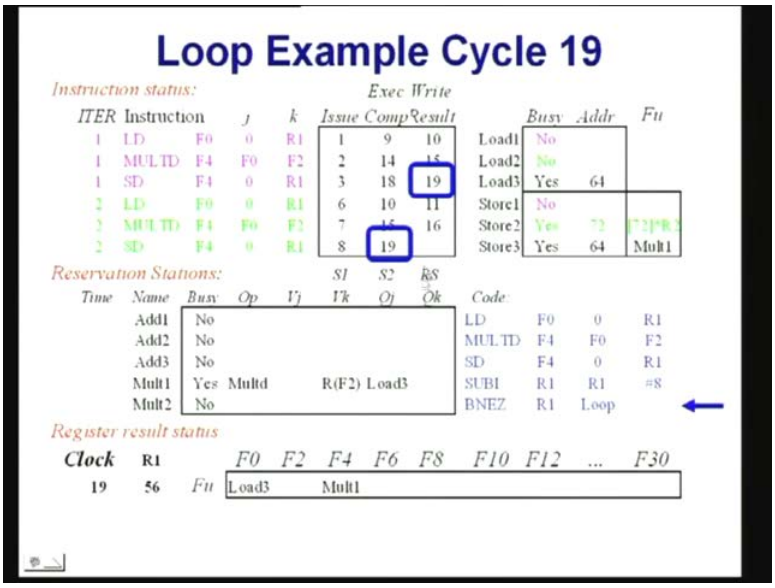
So, it will completing in 16 cycle and now, the stored data will be completed.

(Refer Slide Time: 38:38)



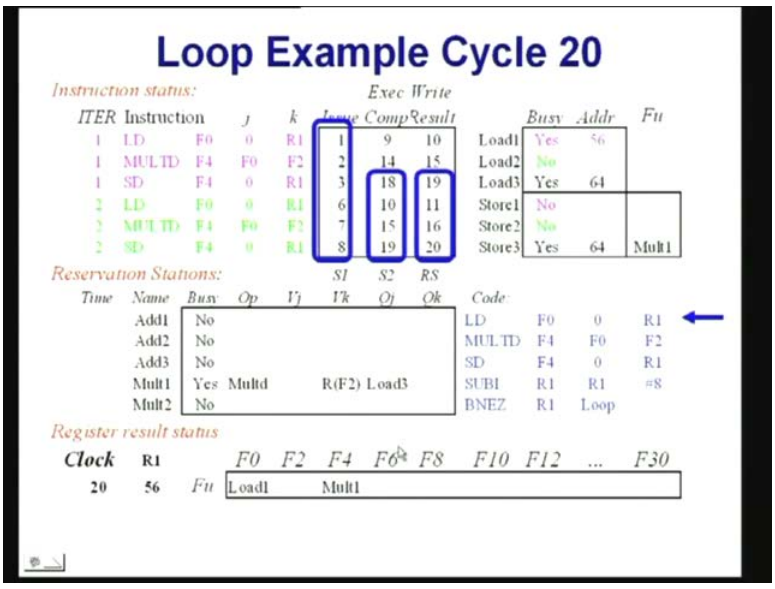
Stored data will perform completion, so it will required 4 cycles.

(Refer Slide Time: 38:47)



14 plus 4, 18 and it is completing, we writing the result and in the 19 th cycle, both this one is completing and I mean, writing results and this one will completing and this store is completing.

(Refer Slide Time: 39:08)

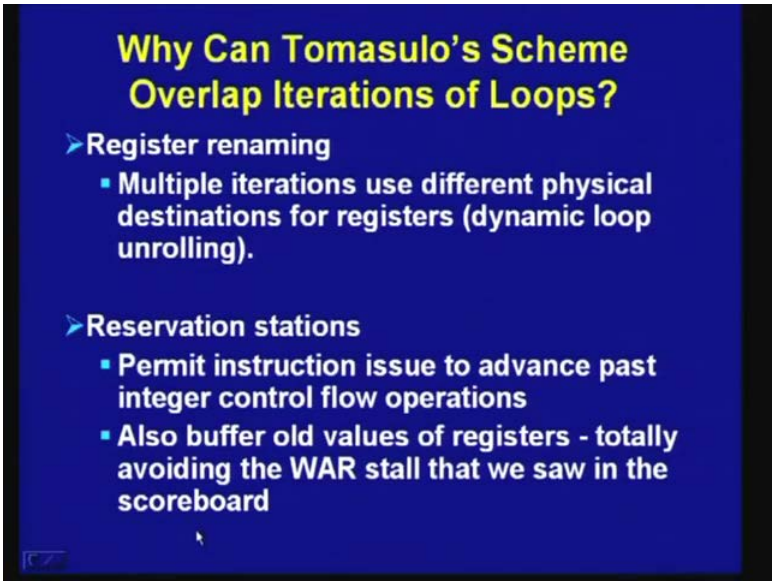


And now, you see this loop example, across the loop boundary, computation has been performed and let us look at this, here we find that, in order issue, no instruction is issued out of order, 1 2 3 6 7 8, so in order issue has been performed. However, out of order completion, execution has been done, out of order execution has been done and as a

consequence, you can see, although I mean, it is completing in 18 cycle, but this is completing in 10 cycle.

So, out of order completion, again this is completing in 15 cycle and this is completing in 19 cycles. So, you can see here, out of order completion is taking place, not in the same order, so this one is completing earlier than this, this one is completing earlier than these. So, out of order completion is taking place and out of order execution is taking place and out of order completion is also taking place in this part. So, as you can see, this is how it is happening.

(Refer Slide Time: 40:26)



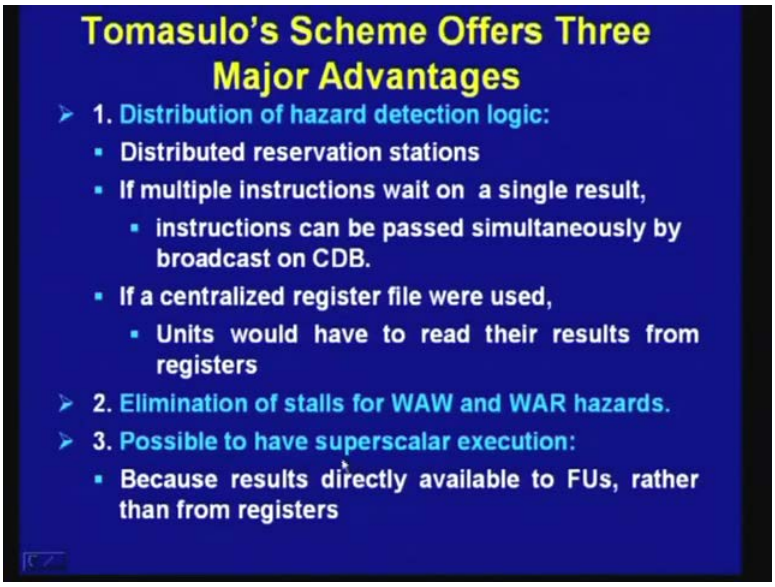
Now, the question arises, why can tomasulo's scheme overlap iterations of loops, how is it possible. Number 1 reason is, multiple iterations use different physical destination for registers, so ii does register renaming. Because of register renaming, different physical registers are used and essentially, you may considering dynamic loop unrolling. Earlier we have discussed in detail loop unrolling, which is carried out with help of compiler and there we have seen explicitly, you have to use registers available in the processer for the purpose of loop unrolling.

But here, as you can see, even without having architectural registers, by that I mean, registers available in the processer. It can do you loop unrolling dynamically using the registers buffers available in the reservation stations, so that is the reason, why it can do. Second is the reservation station permit instruction issue to advance past integer control

flow operations, that we already seen. Also buffer old values of registers, totally avoiding the WAR stall, that we saw in the scoreboard.

So, I mean, starting from score boarding is that, how the write after read or stall is avoided, that we have discussed in detail, that is also performed. And as a consequence, we are able to overlap iterations of loops in the tomasulo's scheme.

(Refer Slide Time: 42:18)

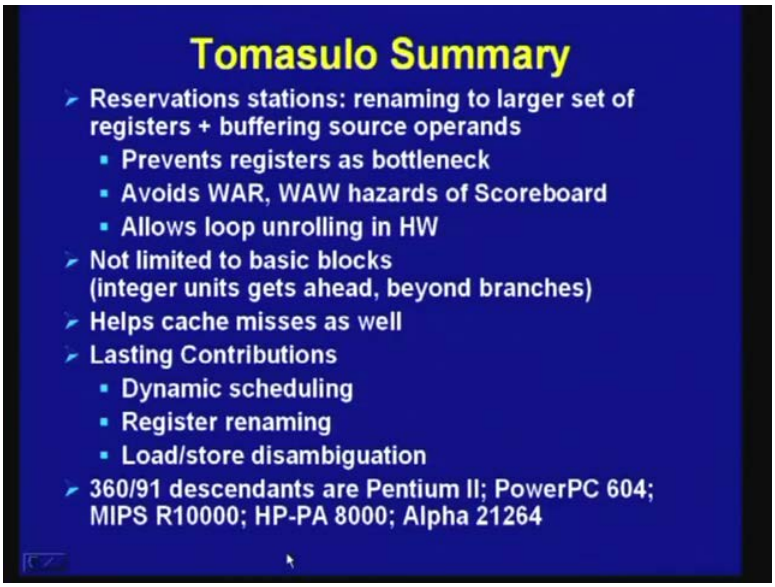


These are the three major advantages provided by tomasulo's scheme, first of all distribution of hazard detection logic. We have seen, there are several reservation stations, which will detect the hazards and if multiple instruction wait on a single result, instruction can be passed, simultaneously by broadcast of common database. And if you centralized register file were used, units would have to read their results from registers. So, these thing is avoided, because we are not using centralized register file, but we are using the registers files available in the reservation stations.

And elimination of stalls of WAW and WAR hazards, I have already elaborated on these, how these stalls are overcome by using that dynamic register renaming and by using that effective address calculation for memory miss, so possible to have superscalar execution. Now, these are lead to another possibility, so far we have considered that, number of instruction issued is only one. But, if you have enough instructions available, which can be executed into parallel, you can go from superscalar architecture.

In other words, tomasulo's basic scheme can be extended for superscalar architecture, however in that case, it may be necessary to have more than one common database. And you may have to duplicate the reservation stations corresponding to more than one common database, but it is possible to add superscalar execution.

(Refer Slide Time: 44:26)



So, this is the summary, reservations stations renaming to larger set of registers plus buffering source operands. This prevents registers as bottleneck that means, this physical availability of the registers in the processers, that restriction is over come. And I have already repeated many times, it avoids WAR, WAW hazards of scoreboards using tomasulo's algorithm.

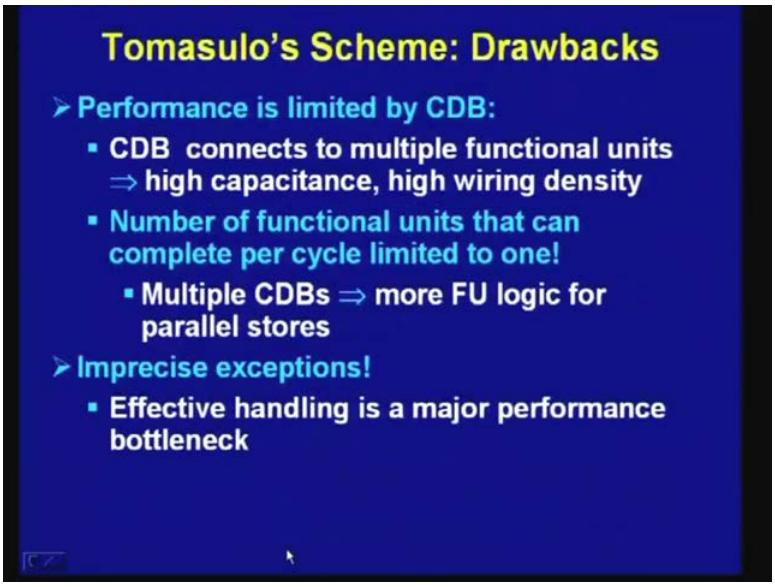
And it allows loop unrolling in the hardware by dynamic register renaming is possible in this scheme and this is also quite clear, it is not limited to the basic blocks, because you can go beyond branches, integers units gets ahead and it go it goes beyond branches. And it helps caches misses, I have already explain this and these are the three lasting contributions of tamosulo's approach. Number 1 is dynamic instruction scheduling, register renaming and load and store disambiguation.

Load and store disambiguation means that I mean, that WAW types hazards and WAR type of hazards arising out of load and store, these are being overcome I mean, disambiguation of load and store this thing was done in tomasulo's algorithms. And as a consequence, many descendants of 360/91 that means, as we know the tomasulo's

approach are originally conceived or developed for 360/91 to improve the performance of floating point operations.

But, subsequently, they have been incorporated in all the I mean, a good number of modern processes like pentium 2, PowerPc 604, MIPS R 10000, HP-PA 8000, there alpha 21264.

(Refer Slide Time: 46:28)

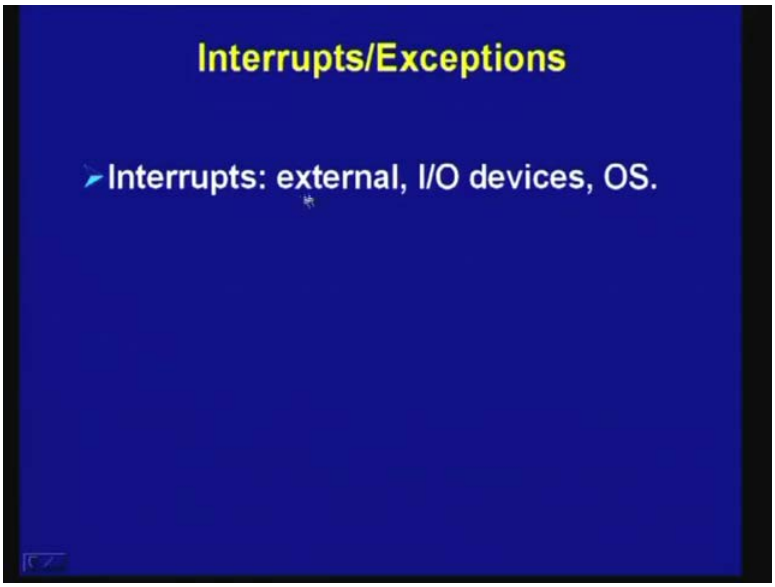


Of course, there are some drawbacks that is, in the common database connects to multiple functional units, because of high capacitance, it will be slower. So, number of functional units that can be computed per cycle is limited to 1, because of common database. So, common database is a major concern in this tomasulo's approach, because all the functional units are feeding to a common database. And as more and more functional units are feeding to it, the capacitance increases and only one of them can output the result through that.

Suppose, it can go to multiple functional units, result can go to multiple functional units, but only one functional unit in it can produce results on the common database. So, this is a severe restriction and so, the alternative approaches to have multiple common database, so more functional units logic for parallel stores. So, this can be done, but it will definitely increase the complexity, so that is one, that is also a drawback to tomasulo's scheme.

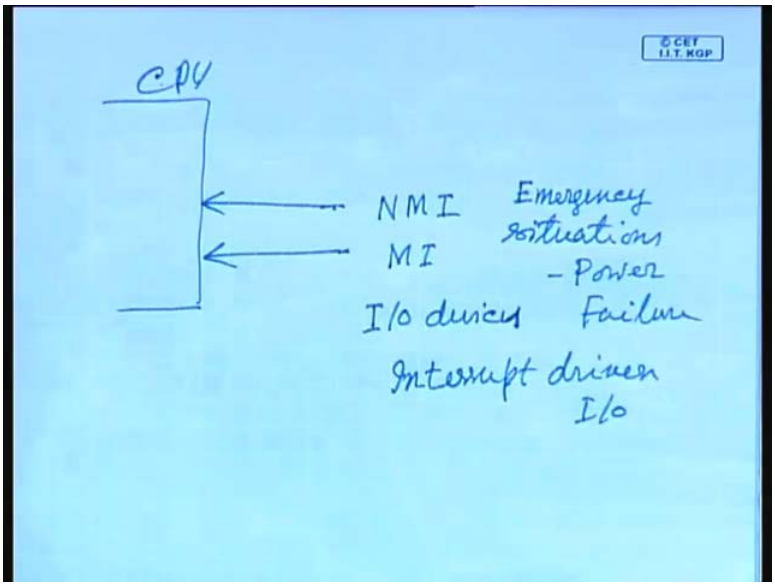
So, the hardware complexity is relatively high, because you required reservation stations having large number of registers. And if you go for multiple common database, it will again increase the complexity of hardware. And another important aspect is imprecise exceptions, so effective handling is a major performance bottleneck.

(Refer Slide Time: 48:17)



Let me briefly discuss about these interrupts and exceptions, which can arise in a processor. Now, as you know, interrupts are essentially generated externally, as you know each processers is provided with...

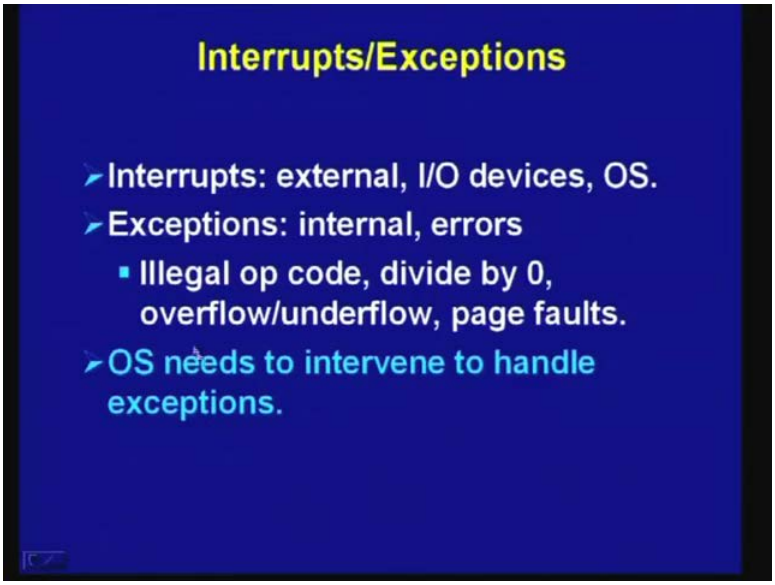
(Refer Slide Time: 48:36)



Suppose, you have brought a CPU, usually keys provided with two interrupts, two interrupts inputs, these interrupts inputs are known as, one is usually none miscible interrupts and another is miscible interrupts. They are available in verity of names, but whatever it may be, one is non miscible interrupts and another is miscible interrupts. Non miscible interrupts is commonly used for some kind of emergency situations like power failure and other things, so it is restricted to that.

But however, this miscible interrupts input is commonly used for interrupts coming from IO devices and that has lead to that interrupt driven IO operations. That means, whenever IO device is ready for providing data to a processor, it will generate an interrupt then, the processor will read that data. So, that is the basic concept, interrupts driven IO and the interrupts may also come from the operating system.

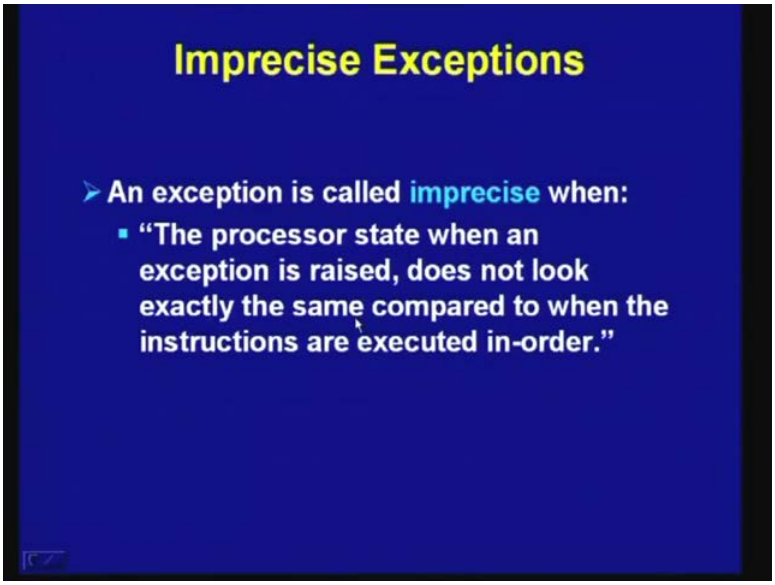
(Refer Slide Time: 50:18)



And another type of interrupts are exceptions, exceptions are internal, which are generated with the, as the instructions are executed. So, for example, an illegal op code is executed, so processor tries to execute an op code and that code is not matching, we need a valid operation codes present in the processers. So, it will generate an exception, similarly then, divide by 0, overflow, underflow, page faults. So, these are various situation that can happen dynamically, as instructions are executed and this will lead to exceptions.

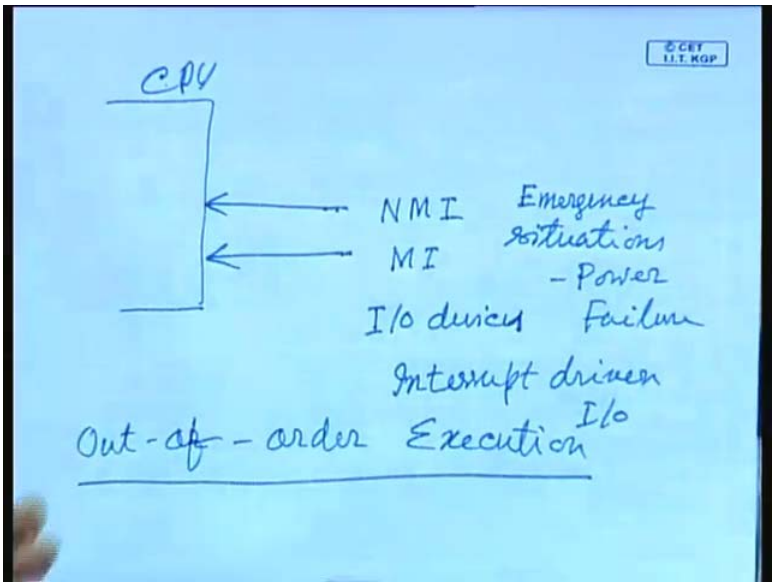
So, whenever these happens, OS need to intervenes to handle exceptions, so whenever these happens, usually control is transferred to the operating system. CPU cannot handle these things, that is why, this control is to transferred to the operation systems.

(Refer Slide Time: 51:31)



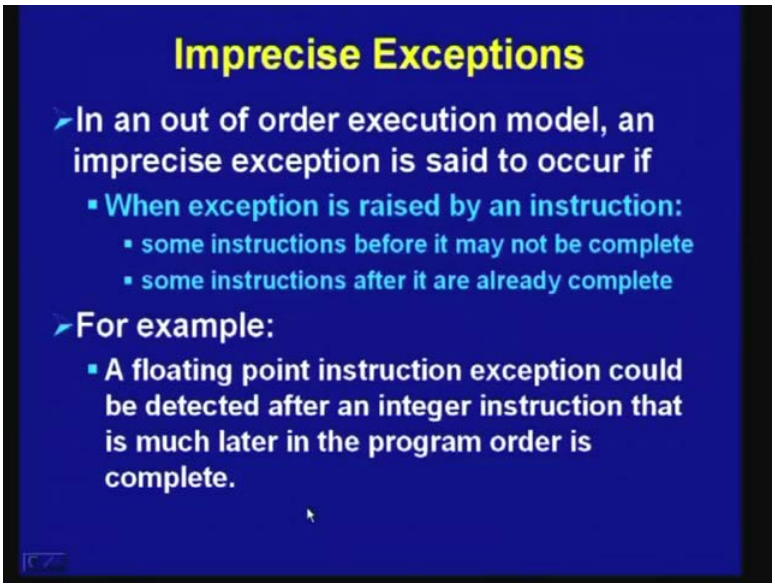
But, there are imprecise exceptions are to be taken care of, what you mean by imprecise exceptions, an exception is called imprecise when the processes state, when an exception is raised, does not look exactly the same compared to when the instructions are executed in order.

(Refer Slide Time: 51:52)



That means, here this is arising out of order execution, we have seen since you are performing out of order execution, what can happen. The exceptions that can be generated by the these instructions should be exactly same, as if those instructions as been executed in order, so these as to be taken care of.

(Refer Slide Time: 52:29)



And this is done with the help of, in an out of order execution model, an imprecise exception is said to be occur, if when an exception is raised by an instruction, some instructions before it may not be complete, some instruction after it are already complete. For example, a floating point instructions exceptions could be detected after an integer instruction, that is much later in the program order is complete. So, this type of imprecise exceptions are to be taken care of.

So, we have discussed tomasulo's approach in the contexts of whenever you have got branch prediction. So, we along with branch prediction, tomasulo's approach can help you to go beyond loop iteration I mean, you can execute instructions beyond loop boundaries I mean, multiple iteration can be executed one after the others. If the prediction is taken, that you have demonstrated with the help of an example and however, there are certain things like imprecise exceptions are to be taken care of.

The various other things like different type of hazards are taken care of automatically, that you have discussed in detail. In my next lecture, I shall discussed about another very important concept and that is, not your branch prediction, but speculative execution. So,

we shall see, what is an speculative execution and how, tomasulo's scheme can be extended for speculative execution, that we shall discussed in my next lecture.

Thank you.