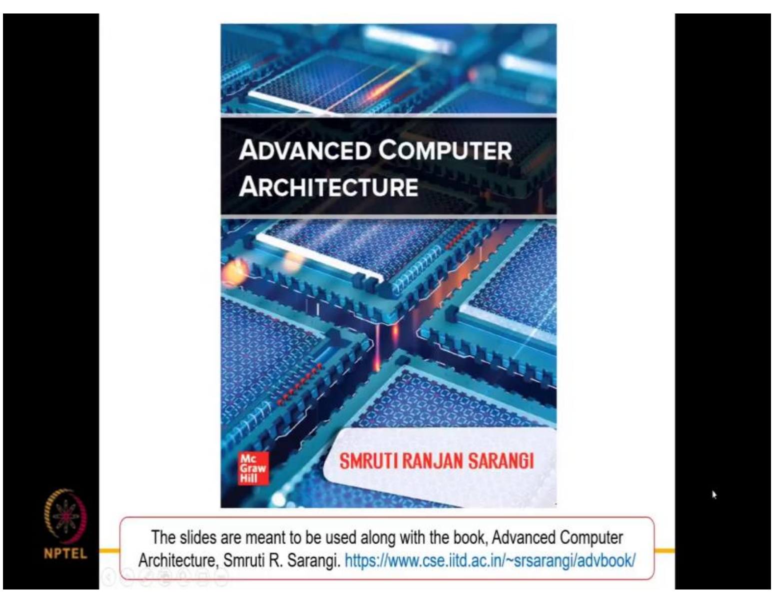
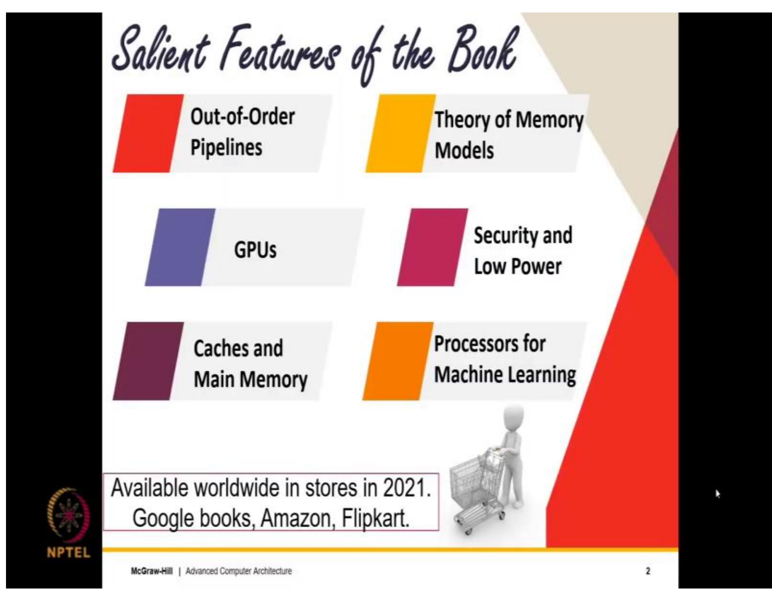
Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 15
Alternative Approaches to Issue and Commit Part - IV

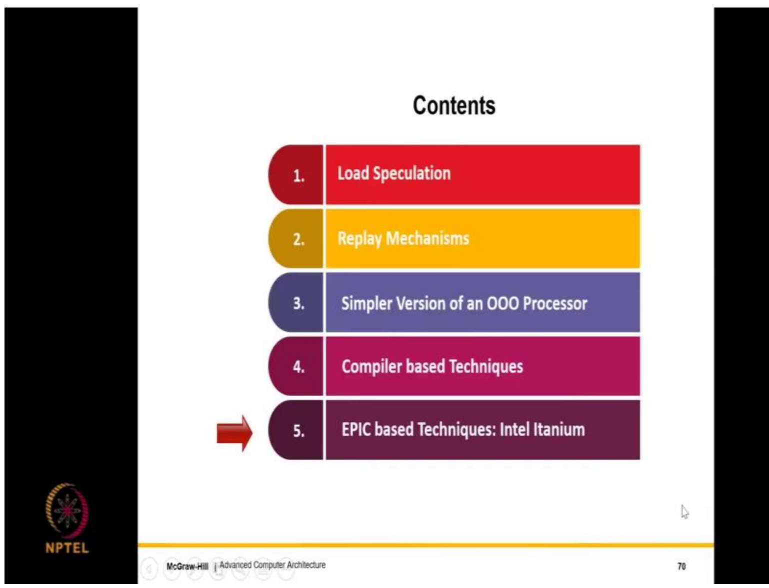
(Refer Slide Time: 00:17)



(Refer Slide Time: 00:23)

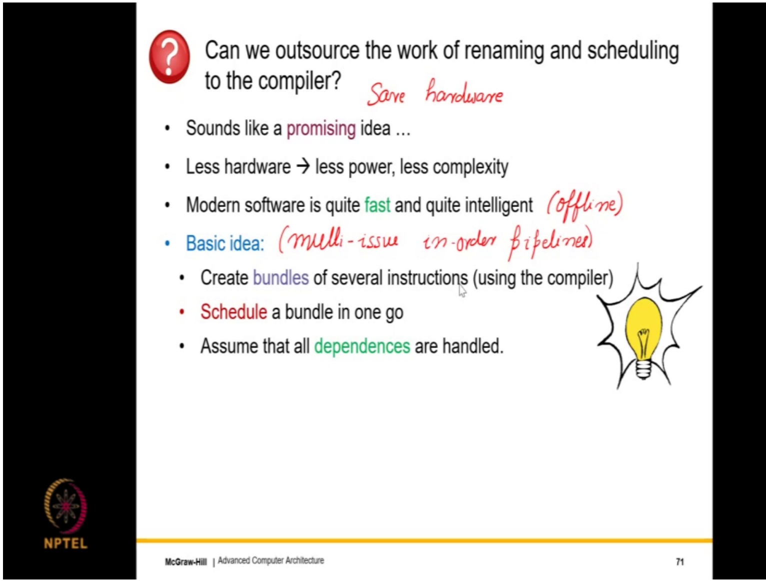


(Refer Slide Time: 00:37)



Welcome to the section on EPIC and VLIW processors. So, in this video, we will look at an important class of process that actually use the compiler for most of their optimizations. so, much of the hardware that we have seen is actually replaced by the compiler.

(Refer Slide Time: 00:58)



So, it start with a provocative question which is that can we outsource the work of renaming and scheduling to the compiler. So, I will use the term scheduling and scheduling together. So, the key question is that, two important functions that we were

doing in hardware namely renaming and scheduling the instructions; can they be somehow offloaded or outsourced to the compiler. So, this will basically save hardware.

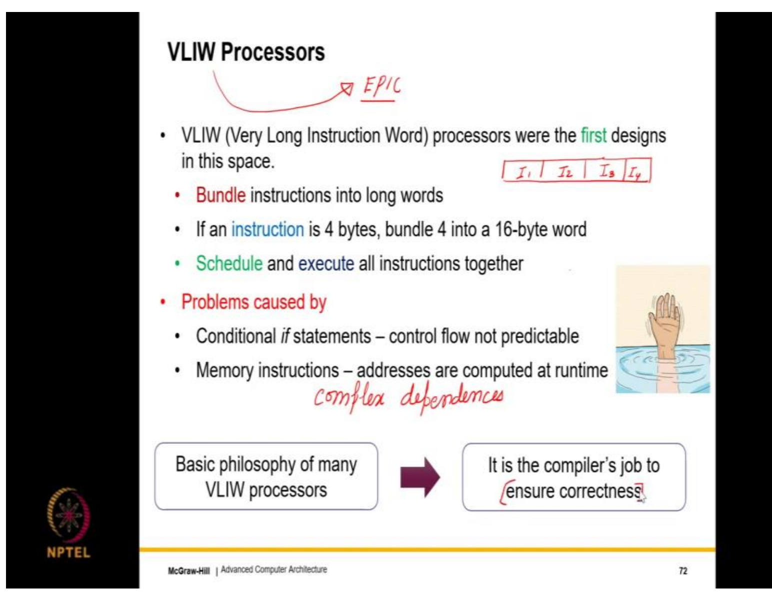
So, the key idea over here is that it is going to save hardware resources and this may be extremely beneficial. So, undoubtedly this sounds like a very promising idea because the moment we have less hardware, we have less power and less complexity. So, less hardware is always good.

Also modern software is quite fast, it is quite intelligent as well and given that we can do a lot of analysis kind of offline before the program has run, we can possibly generate a highly efficient piece of code and also give directions to hardware on how to rename and schedule instructions.

So, what is the basic idea? So, the basic idea this kind of extends from what we have studied in order pipelines multi issue in order pipeline. So, you may want to go back and take a look at that way back in chapter 2, but I would say that we start from here conceptually at least from this point.

So, what we do is we create bundles of several instructions using the compiler. And the expectation is that we do not have dependencies within a bundle and the entire bundle is scheduled in one word one time. And let us assume that for the time being all the dependencies are handled. So, we will take a look at them later.

(Refer Slide Time: 03:09)



So, in this space there are two ideas. So, the first idea which is actually quite old dates way back to the mid-eighties is known as VLIW. Some modern avatar of VLIW is called EPIC. So, let us discuss VLIW first and then we will go to EPIC. So, VLIW stands for very long instruction word these were the first designs in this space.

So, what we do is that we take an instruction or we take a set of consecutive instructions and we add more information such that they can direct the hardware to do different things at a low level so this also is kind of a conceptual successor to micro programming. So, we bundle these instructions into long words. So, let us say if you have a long memory word like this, then we can bundle the instruction.

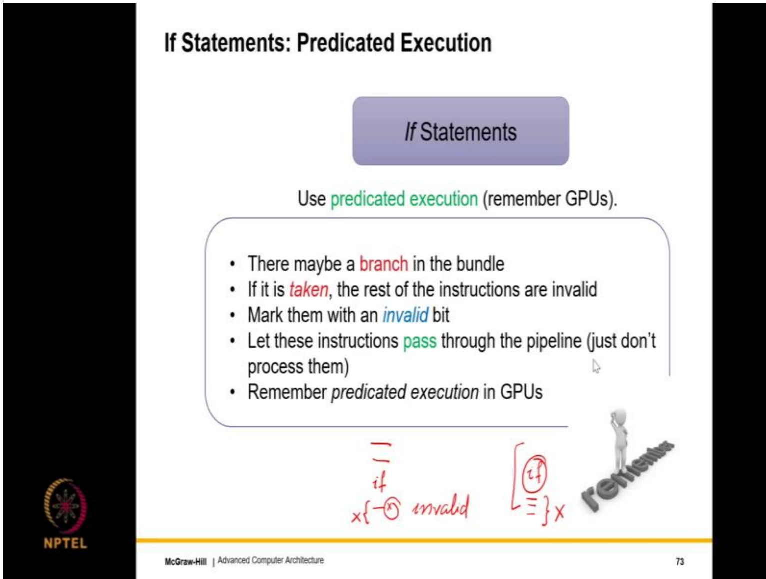
So, let us say this is instruction 1, 2, 3 so on and so forth. So, an instruction is four bytes and we create a bundle of four instructions, then we will essentially have a 16-byte word. So, if we schedule and execute all the instructions together in a word and it is assumed that the instructions are so designed or so arranged that there is no problem whatsoever, but problems will always be there.

So, what we will do here is that, we will look at some other patterns which can cause problems or the problematic patterns. So, one of the patterns is conditional if statements. So, in the case of a conditional if statement the control flow is not predictable. So, we do not know at compile time whether the if part or the else part will actually be executed. So, this will only be clear at run time.

So, the control flow is consequently not predictable and furthermore we can have memory instructions whose addresses are computed at runtime. And as we have seen earlier in this chapter, there can be complex dependencies. Mainly because we are computing the addresses of memory instructions at run time.

So, what is the basic philosophy of many of these VLIW processors? The basic philosophy is that it is the compilers job to actually ensure correctness. So, VLIW processors do not really take upon themselves the job of ensuring correct execution you assume that the programmes are simple the compilers have enough visibility to ensure correctness this being the most important and the most crucial.

(Refer Slide Time: 05:53)

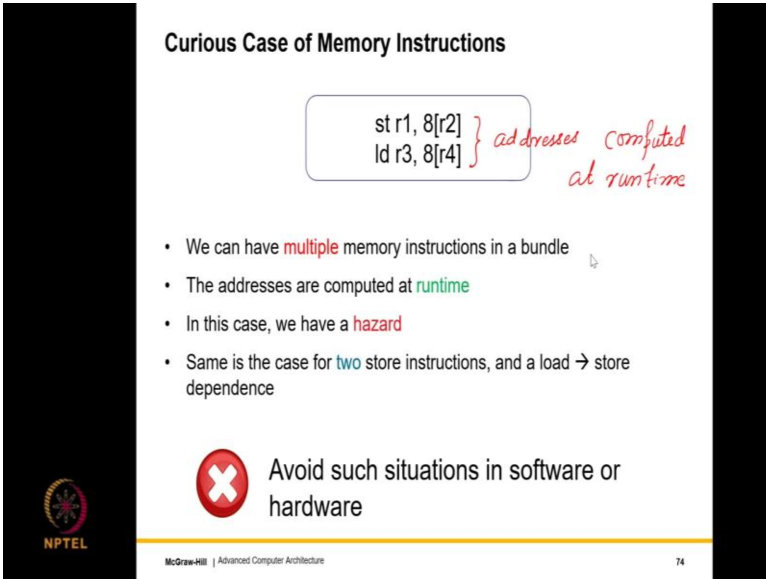


So, let us take a look at some of the ways in which if statements are handled. So, if you remember GPUs. So, we use predicated execution over there. So, we can do something very similar here. So, let us assume that there is a bundle of four instructions one of them is a branch an if statement. So, if it is taken the rest of the instructions in the bundle are invalid which means this instruction is invalid.

So, what we do is that instead of dynamically doing more changes, we allow the rest of the instructions in the bundle to pass through the pipeline as if they were normal instructions. It is just that they are kind of marked with a bit something similar to the Poisson bit that we have seen they are marked with a bit which means that the instruction is invalid.

So, what we do is, we still maintain the semantics of the bundle and if there is an if statement over here which is a taken branch then the rest of the instructions in the bundle are just marked to be invalid and they simply pass through the pipeline, but they do not change the architectural state.

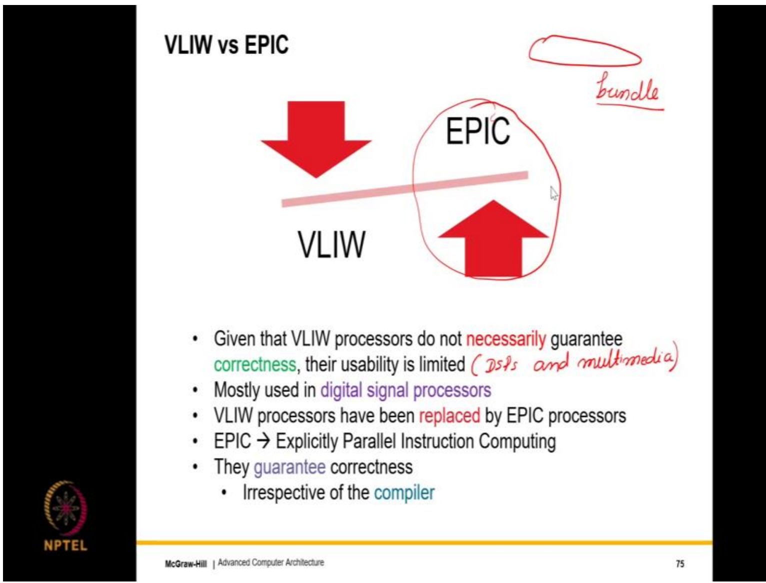
(Refer Slide Time: 07:03)



Memory instructions. Well, we have seen that memory instructions could be quite problematic in the sense that they could have two memory instructions and if we take a look at these memory instructions their addresses are being computed at run time. So, we can have multiple memory instructions in a bundle, addresses needless to say a computer at runtime and we could have hazards and dependencies.

And so, regardless of the memory instructions unless they are two loads because we never have a retreat problem, but otherwise there could be issues. So, we have to avoid certain situations in either software or hardware. So, we will take a look at very elaborate mechanisms to take this issue into account.

(Refer Slide Time: 08:21)

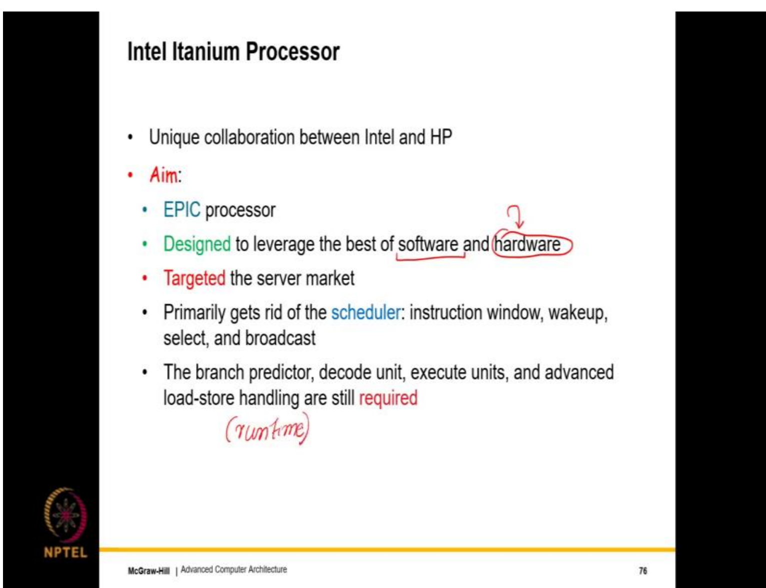


So, these were all about VLIW processors where what we do is that we create a large group of instructions and software, we refer to the large group as a bundle. The entire bundle of instructions is executed at the same time. It is just that VLIW processors do not guarantee correctness and consequently their usability is limited. As of today, they are mostly used in DSPs digital signal processors and multimedia processors.

So, they never saw a generic use so, but let us say if you want to bring VLIW processors into the mainstream which is an effort that was tried in the early part of the 2002 to 2010 decade by Intel and HP. So, they made a very popular processor called Itanium which was also called an EPIC processor. So, EPIC is like the second generation of VLIW it is explicitly parallel instruction computing.

So, in EPIC the idea is that regardless of whatever the programmer or the compiler does the hardware always ensures or guarantees correctness. This increases the load on the hardware, but it also improves the acceptability and the portability of the processor. So, basically fares better in the market mainly because more programmes can run on it and also compilers support can be slightly lacks. So, this is where VLWI ends and the journey of EPIC begins.

(Refer Slide Time: 10:13)

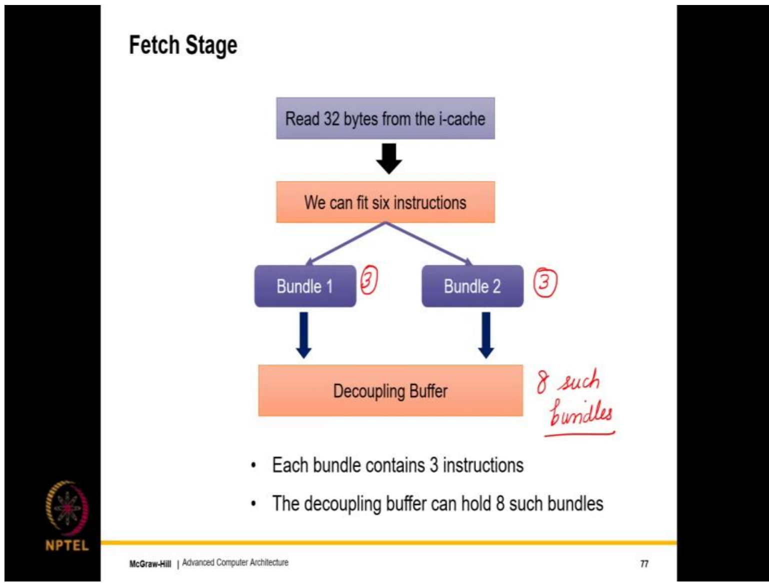


The Intel Itanium processor was a very unique collaboration between Intel and HP. The aim here was to design the first commercially successful and available EPIC processor. So, one thing we need to understand what the that at the time that this processor was being designed roughly in the late 90's and early 2000 compiler technology was already quite matured.

So, that is the reason the best of software was already there and also hardware technology also quite matured, but instead of investing the hardware resources in out of order execution, it was thought that it is a better idea to actually use the hardware for implementing an EPIC like server. The main idea was to target the server market unlike VLIW processors that target specific applications that was not the aim. So, the main aim was primarily server only.

So, what was supposed to be removed was the scheduler notably the instruction window, the wake up select and broadcast mechanisms. But we still needed some units from traditional process such as the branch predictor the decode unit execute unit and also advanced load store handling unit. So, those are still required because branches are something that you need to predict and advanced load store handling is also required because memory addresses are resolved at runtime.

(Refer Slide Time: 12:00)

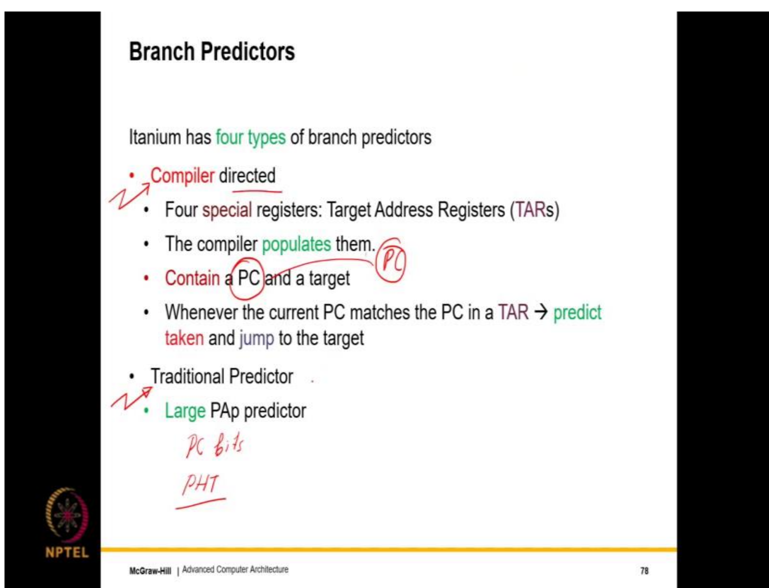


So, we will look at the different stages of the Itanium processor and of course, the best reference are the papers by Sharangpani and Arora on the Itanium processor. So, Itaniun had a high bandwidth fetch rate for those days. So, we could read 32 bytes from the instruction cache in one go. 32 bytes are enough to fit six instructions. So, it is possible to create two bundles of three instructions each.

Given that the fetch rate is very high and the rest of the pipeline may not be able to process instructions at that rate, it is necessary to have a queue between the fetch stage and the rest of the pipeline this was known as a decoupling buffer further the decoupling buffer could hold 8 such bundles. So, it is possible to hold 8 such bundles in the decoupling buffer.

So, one advantage of this is that even if there is a slight mismatch in the rates of fetching and the rates of processing the decoupling buffer can sort of ensure that when we enter the high IPC phase enough instructions are available.

(Refer Slide Time: 13:24)



The branch predictors needless to say are required. Even if we do not have the elaborate broadcast wake up schedule units, branch predictors are required we cannot kind of wish a branch predictors Itanium has four types of branch predictors. So, let us look at them. So, we will look at two types in this slide.

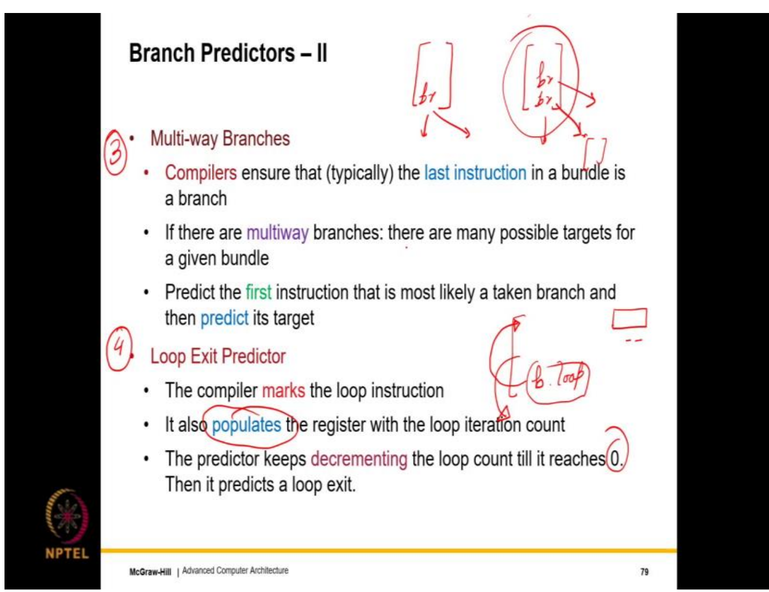
We look at the compiler directed predictors and the traditional predictor. So, the compiler directed predictors are those predictors where the compilers borrow information. It figures out the direction of the branch at the target as well and that is populated software. So, we have four special registers known as the target address registers or the tars the compiler populates them.

So, what does a tar have? It has a PC a target of course, it is assumed that the branch is taken. So, the moment that the programme counter the PC matches the PC stored in the tar reach after the target we predicted taken and jump to the target. So, this would happen let us say in cases where so something like an exception handling code or something where most of the time we will kind of jump out.

In that case it is a much better idea for the compiler to do some analysis and directly give a hint to hardware that a given branch is taken and also it will jump to a given target, here is the target you take it there is no need to actually do additional branch prediction or use a VTV because I am giving you all the information that you need.

So, this is useful this is useful this is clearly the first priority and it is useful in cases where it is possible for the compiler to come up with such information. Otherwise, we will use a traditional predictor. So, the traditional predictor is a large path predictor which we have seen in chapter 3, which basically means that the branch history register PC bits going to that which is a first level and they also go into the second level that has the pattern history tip. So, in both the PC bits are used.

(Refer Slide Time: 15:39)



So, we have two more kinds of specialised branch predictors in itanium, the reason being that fetch was seem to be a massive bottleneck. So, if you think about it Itanium as a very high IPC guaranteeing processor. So, that is the reason you do not want the fetch to be the bottleneck and given the fact that we are doing the best possible compiler analysis that we can including profiling, we can do our best in ensuring that the rest of the pipeline does not face an issue because of the fetch stage.

So, here is the third kind of predicted multi way branch predictor and a fourth which is the loop exit predictor. So, normally what happens is that a compiler would ensure the last instruction in the bundle is a branch. So, if this is a bundle the last instruction in the bundle is a branch. So, step taken or not taken it does not matter the bundle at least is correctly executed, but sometimes it can so, happen there are too many branches we can have two branches.

So, in this case, let us say after this bundle there can be many targets one can be the default where both are not taken, one can be the bundle when the first branch is taken one can be the bundle when the second branch is taken. So, we clearly as you can see we can have multiple targets. If there are multiple targets for a bundle then what we need to predict is that let us say out of the branches that are there which of them is taken that we want to find the earliest branch that is taken and its target.

So, this is a multi-way branch predictor. So, multi way branch predictor is there and Itanium is made it is kind of constructed in a similar manner as the rest of the predictors that we have seen, where the broad idea is that out of all the possible options it kind of gives you the most probable options such that we can start predicting from that part, then we have a loop exit predictor.

So, normally what happens is in many scientific holes we have a loop and the number of iterations of the loop this is known beforehand the compiler knows it. So, in this case what it can do is that, it can have a small loop iteration register or a loop count register it can be populated with the initial value of the loop count and we can always of course, given a piece of code find the branch that leads to the loop that is easy to find ok.

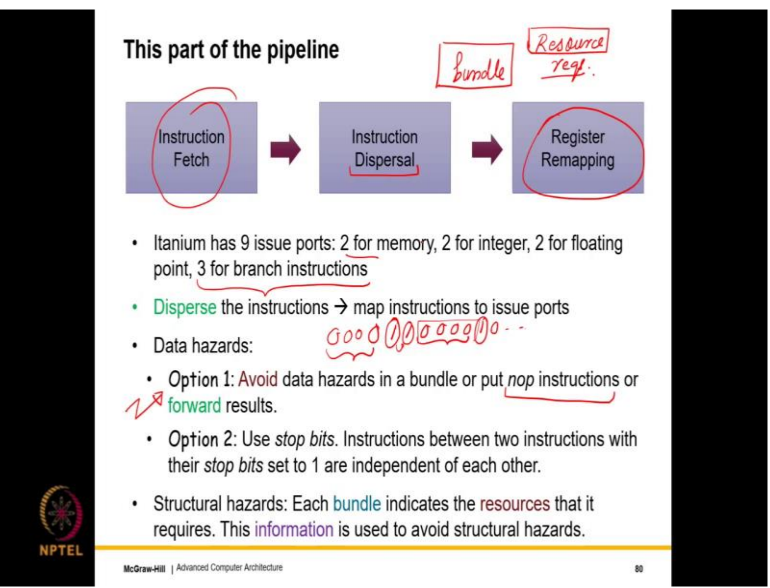
So, this can be in this case this is marked by the compiler even otherwise any branch that normally has a negative offset within a small range, we can think of that as a loop branch. So, the key point over here is that the predictor will keep on decrementing. So, we will have a register for the initial value of the count will be loaded which is the number of iterations of the loop every time we arrive at this loop instruction this will get decremented until it reaches 0.

So, when it reaches 0 what we have seen in our discussion in chapter 3 is that, we normally make a misprediction. But in this case we will not make misprediction instead we will make the correct prediction primarily because the loop count has become 0. So, now we know that at some point wherever we do the check we are actually exiting the loop and that loop exit can be predicted correctly.

So, one of those one odd errors that happen when we either enter a loop or when we are exiting a loop, they will also not happen. So, we are removing those sources of errors as well. So, now, just a quick summary we have four branch predictors compiler directed at

additional pap large pap predictor, a multi way branch predictor to figure out which branch in the bundle is taken the first one and in loop exit predictor dedicated to loops.

(Refer Slide Time: 19:44)



So, now, let us look at this part of the pipeline. So, we have looked at the fetch stage then we have instruction dispersal. So, instruction dispersal is kind of like dispatching instructions to different functional units. So, you can think of it as a combination of dispatch and issue well it is called dispersal over here and EPIC literature. And then register remapping which is something conceptually similar to the renaming that we do of course, with major differences.

So, let us look at the details of the hardware first Itanium has 9 issue ports, 2 for memory; memory instructions that is 2 for integer 2 for floating point and 3 for branch to find whether a branch condition evaluates to true or not. So, you can see the disproportionate amount of resources that have been dedicated just for branches, the reason is that the designers of Itanium took the fetch bottleneck quite serious.

Dispersion of the instructions as we have just discussed is mapping the instructions to the issue ports such that they can be sent to their functional units of course, here we need to be mindful of data hazards. And data hazard in a bundle of three instructions can be there. So, we will try our best not to have them, but the code might be such that we would still have data hazards.

So, there is no going around bit. So, the code might be such that there is nothing that we can do to avoid data hazards. In this case well we have the traditional option to fall back to which is nop instructions or of course, we can forward. So, there is an ample amount of forwarding within itaniums pipeline.

So, that is not something that should worry us, but in some cases like the load use hazard and so, on what we have seen there are of course many more specialised cases in Itanium where forwarding is not really possible. So, in this case we use stop bits. So, what happens is that with every instruction we have a stop bit.

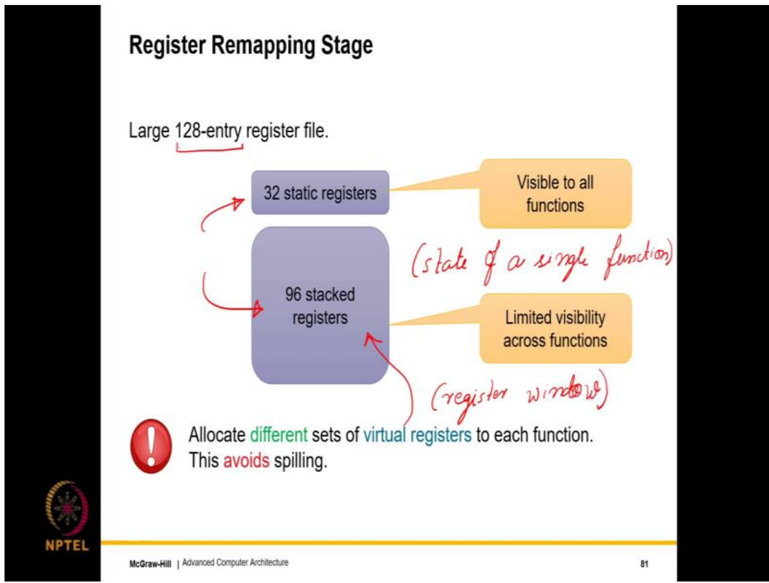
So, we look at the instruction across bundles, the stop bits could be something like this. So, what this is, this pattern is essentially trying to tell us that if let us say all of the instructions between two consecutive ones were independent of each other. So, the hardware can take all of these and schedule them in any way that it wants because there are no there is no dependency between any pair of instructions.

Stop means that it is like it is like a dependent stop ok a stop because of a dependence which essentially means that we it sequentializes execution. So, as long as there are a bunch of zeros we execute them in parallel then we stop execute this instruction, then we execute this again the next port we execute in parallel again we execute this instruction and so, on and so, forth.

So, it is the job of the compiler to insert these stop bits and it is the job of the hardware to interpret the stop bits find all the consecutive zeros or instructions with consecutives zeros as their stop bit and execute them in parallel along with that Itanium also allocates each bundle with the resource requirements. So, for every bundle it is annotated with the resource requirements what it would require?

So, anytime that the scheduling happens, the hardware takes that, figures out the resource requirements and then on the basis of that it does the process of issuing the instructions. So, essentially the resource requirements help us avoid structural hazards.

(Refer Slide Time: 23:43)



Finally, we come to the register remapping stages, the register entry in Itanium is quite special it is quite different. So, first is we have a large number of registers similar to GPUs because GPU registers were truly virtual in the sense we assume that there are they are a very large number of registers, here we are not making that assumption even though we are making something quite similar.

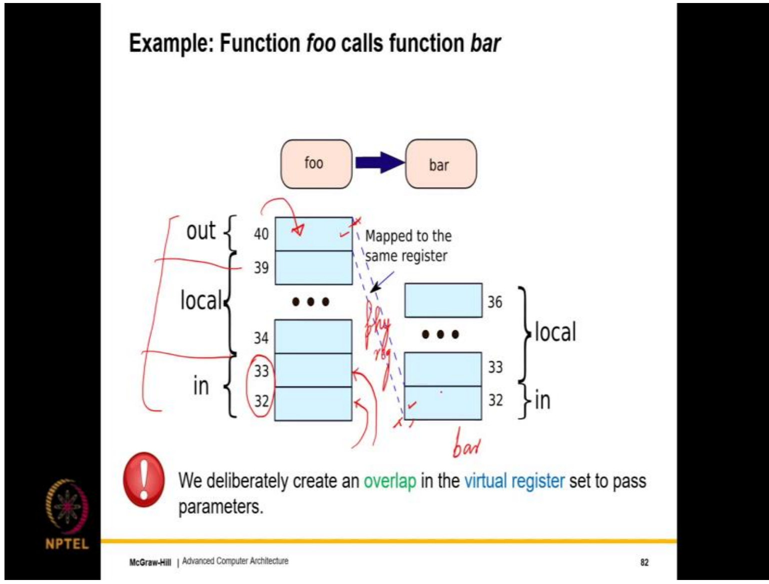
So, the first is that we divide these 128 registers into two parts, 32 static registers and 96 stacked registers. The static registers are visible to all functions. So, all the functions can see and access these static registers whereas, the stacked registers are pretty much meant to have the state of a single function. So, they are not really visible and they are also managed in a special way which we will shortly discuss.

So, here is a very very limited visibility across its very low. So, what we do is that we allocate different sets of these virtual registers in the stacked region to each function and then because we avoid different sets you have a large number of registers. So, we can even if you have multiple functions that arrive at the same time, they can be assigned different registers.

So, because they are assigned different registers we do not have spilling the register spilling into memory the normal way that we have, that is because first we have too many registers and second different functions are given different registers within these set of stacked registers. So, this idea is also known as a register window which means

that every function is given a set of private registers to work with and so SPARC also at a register window. So, we will also do that.

(Refer Slide Time: 25:46)



So, let us take a look at an example. So, the example will make it quite clear. Assume that the function foo is calling the function bar. So, what we do is that we take a set of registers and we mapped them like this that we divide them into three sets the n registers are the ones in which the parameters are passed into the function.

The local registers are the ones that are used to store internal variables and do computations and so on and the out registers are the ones that are that are used to essentially send parameters to functions. So, this function invokes the callee functions. So, since foo calls bar, what we actually do?

So, what you are seeing over here? These are the virtual register numbers. So, what we actually do similar to virtual memory we mapped this register over here to this register over here. So, we just mapped them. So, one advantage of mapping them in this fashion is like this that basically similar to sharing a virtual memory page both of these registers actually become the same.

So, the moment the foo function write something over here this automatically is visible to the bar function which is of course, invoked at some point of time by foo, but bar

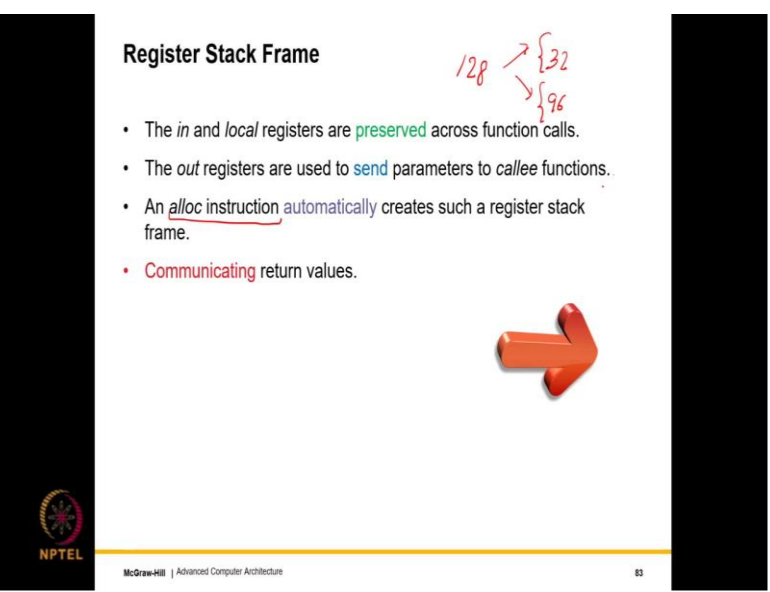
automatically finds its input argument over here in this register. So, what you can see is that the same.

So, these so these are virtual registers, but two different virtual registers are mapped to the same physical register. So, as a result one right is automatically visible to the called function and then the callee over here which is bar it again has its set of local registers and out registers.

So, what is being done over here is that, we are deliberately creating an overlap in the virtual register set of two functions to pass parameters and there is similar to virtual memory there is a manager in hardware and software to manage the set of registers. To ensure that these mappings are created to manage that we have a management mechanism we will talk about that later.

But the key idea is that passing parameters and functions and passing function parameters is quite easy. Basically, we take the set of virtual registers partition them into three types in local and out and out registers are mapped to the in of the callee function and that is how values are passed.

(Refer Slide Time: 28:41)

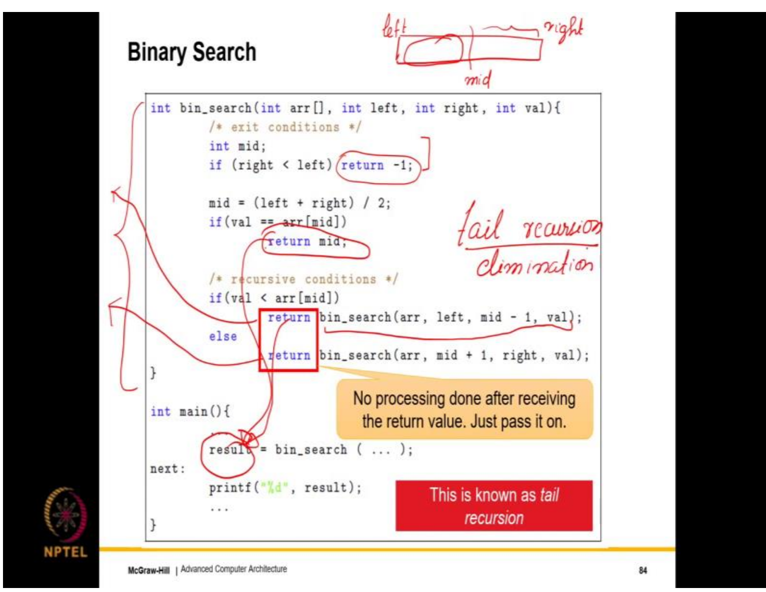


So, we have a register stack frame. So, what happens is that the in and the local registers are preserved across the function calls, as you have just seen the out registers are used to send parameters to callee functions. We have an alloc instruction in itanium, it

automatically creates a register stack frame for a function. So, that is how the stack frames are allotted or allocated.

So, currently let us start talking about overflows because recall that we have only 128 registers, out of 128 we have 32 registers called static. and we have 96 registers called stacked which are precisely being used for this purpose.

(Refer Slide Time: 29:30)




```
int bin_search(int arr[], int left, int right, int val){
        /* exit conditions */
        int mid;
        if (right < left) return -1;

        mid = (left + right) / 2;
        if(val == arr[mid])
          return mid;

        /* recursive conditions */
        if(val < arr[mid])
          return bin_search(arr, left, mid - 1, val);
        else
          return bin_search(arr, mid + 1, right, val);
}

int main(){
        ...
        result = bin_search ( ... );
next:
        printf("%d", result);
        ...
}
```

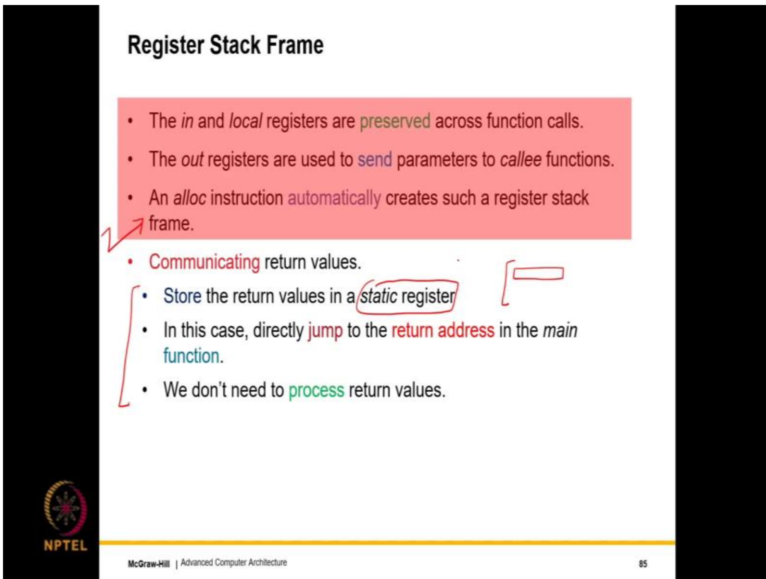
Coming to return values, I would first like to motivate the mechanism by showing the code of a binary search programme. See if you see the binary search programme over here. So, what does it do? What it does is that it takes in an array, it takes a left pointer a right pointer is such searches for the value within it.

So, after doing some sanity checks it finds the midpoint, it compares the value with the midpoint. If let us say the value is less than array mid then we basically run binary search over this half and return the value that it returns. So, if you look at this functions it invokes the binary search routine for the left half whatever that returns it also returns. So, the return value is not further processed.

Similarly, if the value is let us say greater than mid then we focus on the right half and once there is a match whatever the right half returns this function also returns. So, if you think about it in the calling path of course, a lot of computation is happening, but in the

return path whatever the callee functional is returning, it is essentially the same function albeit with smaller arguments the value is just being transparently passed.

(Refer Slide Time: 30:56)



So, let us now understand some critical aspects of the register stack frame. In and local register that clearly are clearly clearly clearly preserved across function calls the out registers are used to basically send parameters to callee functions. So, once the callee functions are used there we do not really care and what we have discussed in the past is an alloc instruction is used to creates register stack frame.

Communicating return values is slightly tricky, but if we let us say take the previous example as a representative, it is a much better idea for let us say whoever produces the value let us say that this instruction over here. So, you have to see where the recursion is terminating. The recursion is terminating at these two statements.

So, since the recursion is terminating over here either at return minus 1 or return mid, then we can think of these as the final return statements and then whatever they are returning it just comes coming back back back back until this point. So, it is much better if you directly have a shortcut from here to let us say over here and we can completely skip the rest of the functions. So, what we will do is that we will store the return values in a static register.

So, recall that this is a register in a static area which is the first 32 registers. So, the return value can be stored at any point over here. In this case, we can directly jump to the return address in the main function which is basically directly jump over here and whatever is the result of the binary search will directly be assigned to the variable result.

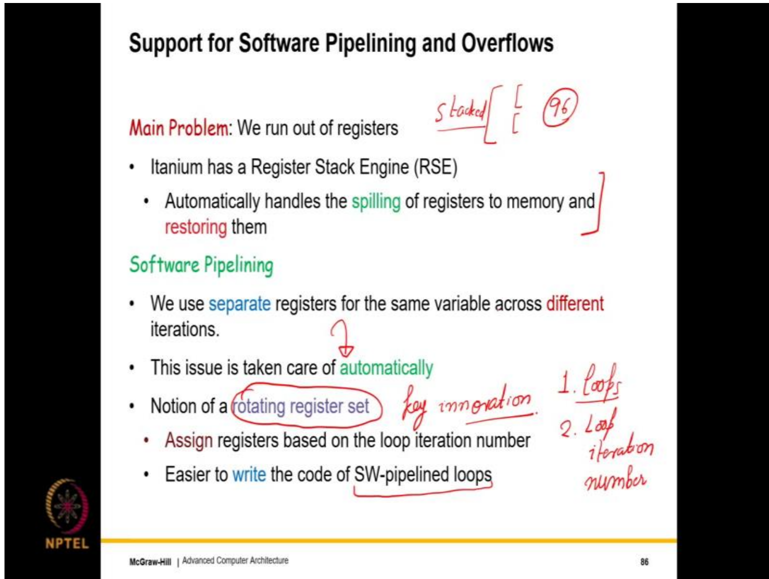
So, that is the reason it is good to have the return value in an area which is not within the register stack frame, but it is within the static region and then well we can quickly resume processing the return value of binary search. So, that is the reason we allocate a separate area.

So, I am not saying that for all function calls the binary search example is representative. In fact, there is a very famous pattern it is called tail recursion. So, this pattern is known as tail recursion and if we directly jump from let us say this point and then set the value of result this will be known as tail recursion elimination.

So, it may appear that this scheme is kind of disproportionately biased towards tail recursion, but there are many similar patterns can have throughout ports and of course, a lot of analysis was done, so it was found out that for return values we can safely keep it in a space which is not in the stack such that it can be processed not just by this function, but by other functions as well and then we can return back quickly.

So, in that case, we do not really have to copy values across registers or restore from memory or do anything of that type. So, that is why a separate storage region was provided for return buckets.

(Refer Slide Time: 34:09)



Let us now look at some more issues. So, let us first take a look at the first problem let us say we run out of registers. So, the reason we will run out of registers is because as I have discussed we have a stacked register storage area and the local state of all the functions at least that are currently active is going in there and since only a 96 registers in this area we might have run out of registers.

So, the fun part over here is that Itanium has a registers stack engine an RSE which automatically handles the spilling of registers to memory and restoring them. So, which is actually great. So, this means that the compiler or the user do not really have to be bothered with this aspect of writing code. So, they can happily write code and then a dedicated engine will just manage the communication of registers to and from memory.

So, this is let us say a major assistance that is being provided to the assembly language programmers and the compiler that this is being done automatically. So, basically handling the function state to a large extent is being done quite easily and quite automatically. Now coming to software pipeline. So in software pipelining if you would go back a few slides you would see that we use separate registers for the same variable across different iterations.

The reason is that we are essentially taking instructions from different iterations executing them together. So, for the loop variable particularly we use separate resistance such that its value is updated correctly. This issue in it in Itanium is taken care of

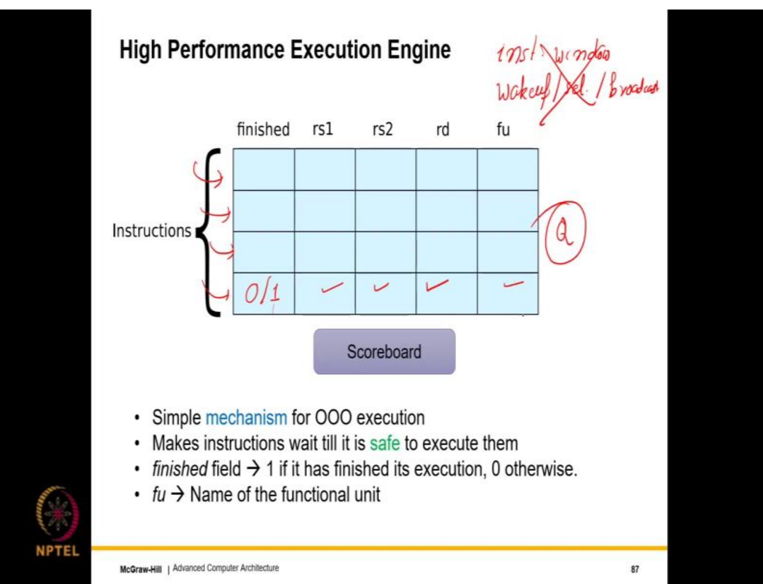
automatically. So, we have a rotating registers set. So, this is the key innovation over here which is the rotating registers set ok.

So, we assign registers based on the loop iteration number. So, as we have discussed earlier in the case of the loop exit predictor, we can very easily predict what is the current iteration number well why that is because the compiler annotates the loops. So, you can also find it ourselves by looking at backward branches, but in this case the compiler annotates.

So, at least a loop iteration number is always known depending upon the loop iteration number we can use the appropriate register from the rotating register set which is think of it is exactly the same thing that we were doing in software pipelining. It is just that at that point we are using more registers compiler were generating all this extra code all of that goes away because the hardware takes care of it.

So, this makes it significantly easier to write software pipeline loops because the key problem which was to assign different registers to different iterations is being taken care of.

(Refer Slide Time: 37:09)



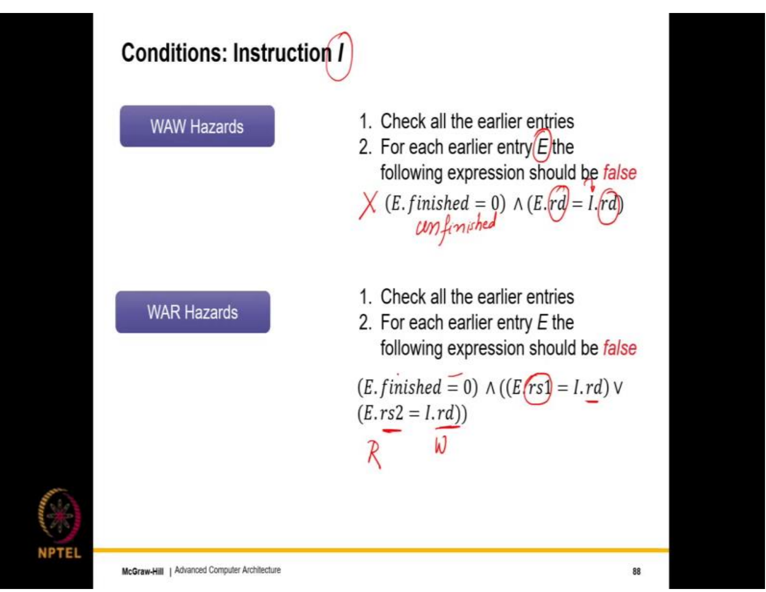
Next comes the high performance execution engine. So, in this case we do not have an instruction window, we do not have a wakeup select broadcast mechanism. So, these are things that we do not have the these are things that we do not have. Instead what we have

is we have a scoreboard a scoreboard is something which was kind of in vogue in the late 70's after that it was not used.

But it is a very fast and it is a very power efficient mechanism of scheduling instructions in this case a scoreboard was found to be a good solution. So, I will tell you quickly what a scoreboard is, it is clearly much simpler than the regular out of order execution mechanism all that it does is the consider it as a Q where instructions are entered like this it makes an instruction wait until it is safe to execute.

So, we store its source registers destination register, the functional unit that is going to use and whether it has completed or not right 0 or a 1 that is all we have stored these five fields for the instruction and this is implemented as a Q.

(Refer Slide Time: 38:35)



So, now, if you look at this what we do is we allow an instruction to go through or we allow an instruction to execute only if certain conditions hold what are the conditions? That we check all the earlier entries for each earlier entry E the following expression should be false. So, first is so, what should be false? (E. finished = 0) which means that it is unfinished not finished and (E. rd) is destination is same as the destination of the current instruction which is 'I'.

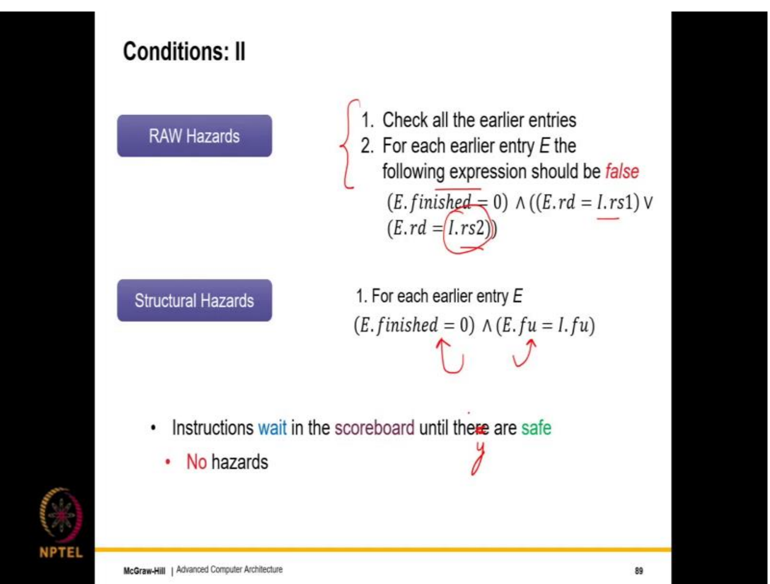
So, 'I' is always the current instruction. So, this means that we can have a write after write hazard because there exists an earlier instruction that is writing to the same register

and that earlier instruction is not finished. So, this condition over here which is (E. finished = 0) and (E. rd = I. rd) this condition has to hold false.

Fair enough we can create a similar set of conditions for the rest of the hazard types. So, like write after read hazard same thing check the earlier entries for each earlier entry E the following expression should be false. So, (E. finished = 0) which means is unfinished and either the source of (rs1) I mean its sources equal to is destination (rs1) = I s destination or (rs2) = I s destination.

So, this is a typical write after a read kind of scenario. So, in this case we just need to create a circuit to verify all of these Boolean equations or Boolean expressions and on the basis of this we need to figure out if there are any earlier entries with which we can have a war hazard or not if there are then we do not schedule and we wait.

(Refer Slide Time: 40:44)

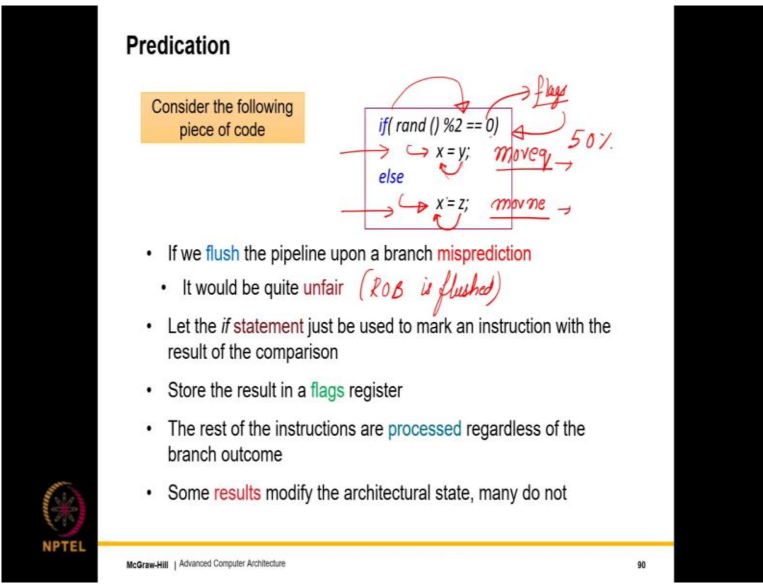


A similar expression for an RAW hazard the rest remaining the same the instruction should be unfinished and in this case, (E. rd = I. rs1) or E. rd or E s destination should be equal to the current instructions second source. So, either its equal to the current instructions first source or second source there is a typical read after write scenario.

And on a similar on similar lines a structural hazard is again when the instruction is unfinished not finished and the functional units are the same. So, instructions wait in the scoreboard until they should be they are safe and there are no hazards. So, they wait in

the scoreboard until there is complete safety there are no hazards. So, there is a parallel score board unit that keeps on checking this and once an instruction is saved it is executed.

(Refer Slide Time: 41:50)



So, now let us take a look at a few more aspects of the Itanium execution pipeline. So, consider the following piece of code we find a flip a random coin, if it is 0 then we set execute this instruction (x = y) otherwise this execute this instruction. So, in this case, the probability of a branch misprediction is very high it is close to 50% because the output is from a random process.

So, if you flush the pipeline after every branch misprediction it would be quite unfair. Why would it be quite unfair? Well the reason is that this is a very high overhead heavy duty operation where the entire rob is flushed the entire state is flushed we will lose a lot of data we will lose a lot of information a lot of work.

So, what we can do is we can let the if statement just be used to mark an instruction with the result of the comparison, in the sense that we can simply mark an instruction and say that look either the comparison succeeded or it did not and we store the result in the flags register as we typically do.

The rest of the instructions which means in this case this instruction and this instruction both are processed regardless of the branch outcome, but it is just that this instruction is

marked with basically the output of the if statement. So, we will take a look at an example. So, what we normally do is that this is known as a conditional or a predicate instruction.

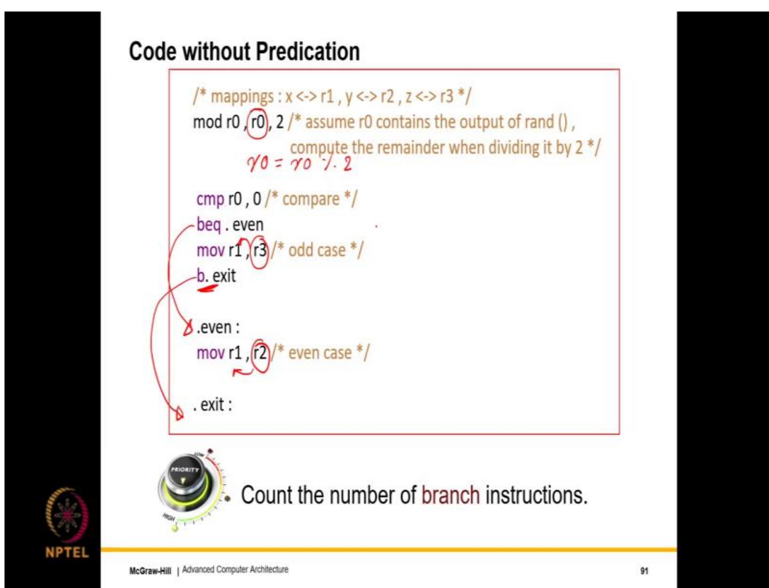
So, in this case its let us say we will use a move instruction to move this value over here, then we can say move equality in a sense if there was an equality in the last comparison, then only this move will happen otherwise it will not happen and on a similar note here we can say mov ne this would basically mean that well we will move the value of that z to x only if the previous comparison over here was not successful.

So, in this case, if you see we are processing both the instructions, but both the instructions have an additional flag or let us have some additional information where we go and check the result of this comparison. In a sense there is a degree of annotation which goes and checks and only if the result has gone in a certain way is the instruction actually executed.

So, some may modify the architectural state some may not, but the key idea is that we will use these conditional instructions such that we can process these instructions. So, one advantage is that just in case there is a branch misprediction. There is no need to flush the pipeline. In fact, what we do is that we just store the result of this in the flags register.

So, that we do not use branches the subsequent instructions which use the two of codes moveq and movne, simply check the flags instruction and in this case since the flags will say that look it resulted in an equality if it actually did the mov will happen in other case if it did not results in an equal equality the mov will happen.

(Refer Slide Time: 45:16)

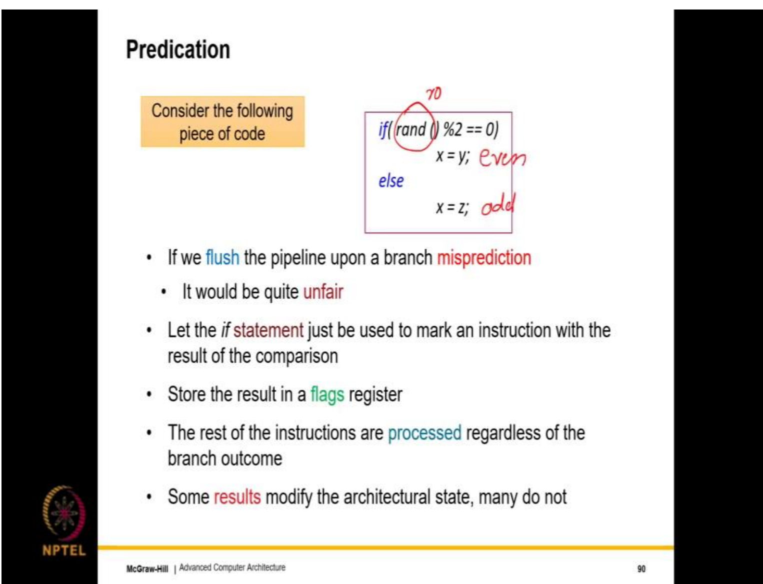


So, I am showing a small piece of code over here which is the code without predication. So, in this case, what we do is that, we take the value input value and r 0 we compute this expression r 0 % 2 which means we take its modular with respect to 2 if its then we compare it with zeros.

Say if it is an even number we jump over here just note the extra instructions on the branches. So, they cause trouble if it is an even number we mov the value in r 2. So, we assume y is stored in r 2 into r 1 and we exit if it is an odd number then we mov the value stored in our 3 into r 1 and then we exit.

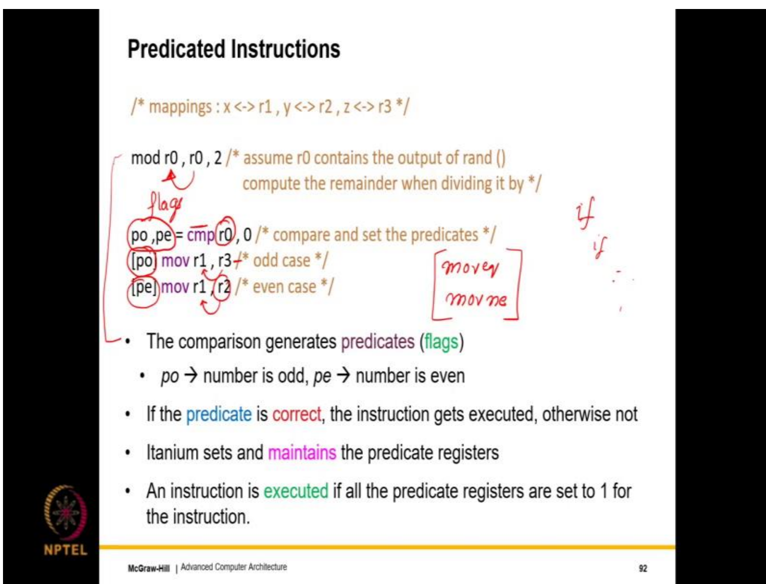
So, note the fact that we have an unconditional branch over here. So, that has a set of problems and we have a conditional branch over here which has major problems in the sense that if we mis predict it, then the entire pipeline has to be flushed. So, the code that you are over seeing over here is exactly the implementation of the code over here.

(Refer Slide Time: 46:32)



So, I will just maybe clear off the ink. So you can see is that the code over here is very similar with the value of rand is stored in the register r 0. So, here we compute the modulo compare and do a bunch of jumps to implement the logic.

(Refer Slide Time: 46:42)



What we can do which is actually much simpler and this is something exactly on the lines of what many processors not just itanium, but even arm processors are also doing it these days is that, well we first compute the modullo. So, we first compute r 0 % 2 and store it over here, then we compare the result with 0.

Then what we do is that we store these two these are called predicates we are also been calling them flags they are stored in the dedicated registers. So, these predicates are stored in 2 registers p o and p e where if it is odd po is set where if it is even pe is set then what we do is that in the odd case or let us say for the odd case which as you can see is this case.

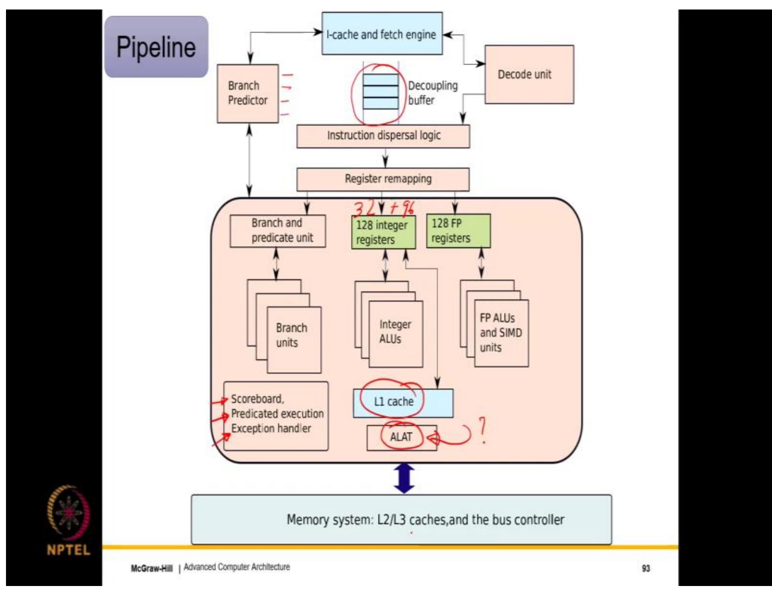
Where we set it if I were to just go back, this is an even case over here this is the odd case over here where x is set to z and so, then we come here and then come here say the odd predicate is set we mov r 3 to r 1 which is essentially the value of z and this is set over here and then we have the even case where the even predicate is sets a note that only one of these predicates can be set so the even case is set then we mov r 2 to r 1.

So, in this case there are no branches there is nothing. So, there is no possibility of flushing the pipeline and as you can see the predicate is correct the instruction will get executed otherwise it will not. So, this is similar to the previous example that we showed with moveq and movne which is similar to the arm terminology that was explained in the first part of the book not this book, but the previous one on the on undergraduate computer architecture.

But the key point over here is that the predicate registers capture the outcome of the prediction. If let us say the given predicate register if its value is 1 as set by the cmp then only this instruction will execute otherwise it will just flow and pass through the pipeline without changing the architectural state.

We can also have nested branches and if and if within it and so, on there could be multiple there could be multiple predicate registers. So, an instruction will get executed and it will change the architectural state only if and only if all the predicate registers associated with it are 1 or it would take an and of them and all of them are one then only it will get executed.

(Refer Slide Time: 49:28)



So, here is a summary of the pipeline as far as we have seen up till now. So, in this case, we have an I have a I cache and fetch engine, we have a decode unit we also have a decoupling buffer whose main job is to separate the fetch unit from it and we have a branch predictor we have seen that we have four kinds of branch predictors compiler directed, a large pap branch predictor, a multi way branch predictor and loop exit predictor.

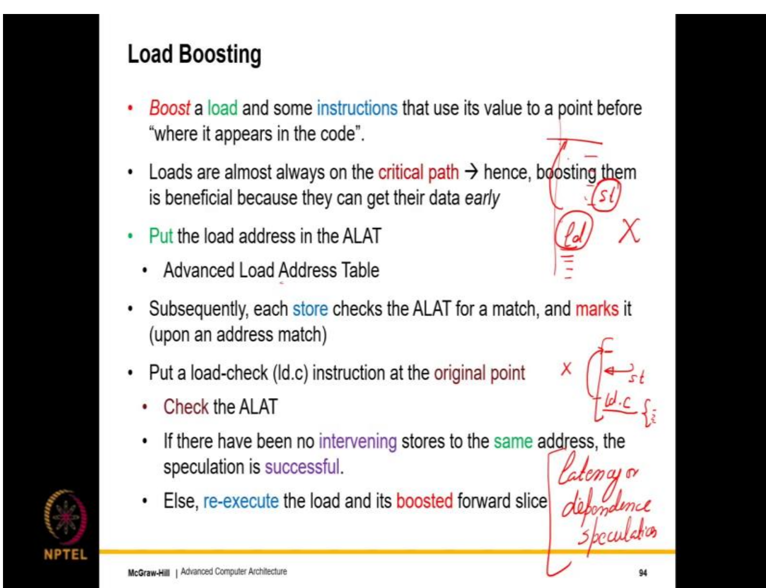
Then we are the instruction dispersal and register remapping logic. So, dispersal essentially assigns it to the issue port. So, we have a fair amount of support and also resources for the branch and predicate units. So, the branch units compute the result of branches and predicates as well. We have 128 integer registers divided into (32 + 96) like static and stacked something similar with FP registers floating point registers.

So, the Itanium processor also has assembly units and then along with integer ALU and FP ALU we connect the integer registers to an L 1 cache. So, we will discuss the advanced load address table mechanism. So, this is something that you have not discussed up till now, but we have discussed the rest we have discussed the scoreboard which schedules and orchestrates the execution we have discussed predicated execution.

So, what will not discuss the exception handler? So, exception handler happens if there is an interrupt from the keyboard or something and finally, the processor over here the

EPIC processor interfaces with the memory system via the L 2, L 3 caches and the bus control.

(Refer Slide Time: 51:21)



So, the only major part only major chunk or the only major component that is remaining is the lodestone mechanism. So, this is quite intricate. So, what we do is that, if we let us take the value of a load we boost it up in the program in the sense if the program the load appears over here we move it up.

And along with that we also maybe move up some instructions that may use its value, there is a good reason is that loads are almost always on the critical path. Hence, we boost them up and we kind of get the data early and then you know if we are ready with the output of the load many more instructions which are dependent on the load they can begin to execute and so, we will basically have a higher degree of ILP.

So, that is the reason a lot of loads whose let us say value either you can predict or we can predict that there are no dependencies they boost them up. Once they are boosted up in the sense that they are in the point of the program where they should not be, there could be several problems in the sense if they are boosted over here there could be stores in this region that write to the same address and in that case there will be an error.

So, what are we doing we have regular program over here we have a load over here the load is being boosted up over here, why is such that the value comes the load comes in

early and all the instructions are dependent on the load, they can get their value. So, they can start execution earlier. The only spoiler could be if there are other store instructions that appear over here in this case the this will be incorrect.

So, what we do is that any load that is boosted we put it in the advanced load address table or the ALAT subsequently whenever there is a store it checks for the ALAT for an address match, if it finds it will kind of mark that entry as dirty. In the original position of the load which is much later instead of the load we will have a load check instruction which will check the ALAT and find if the lord is still valid or not why will it not be valid?

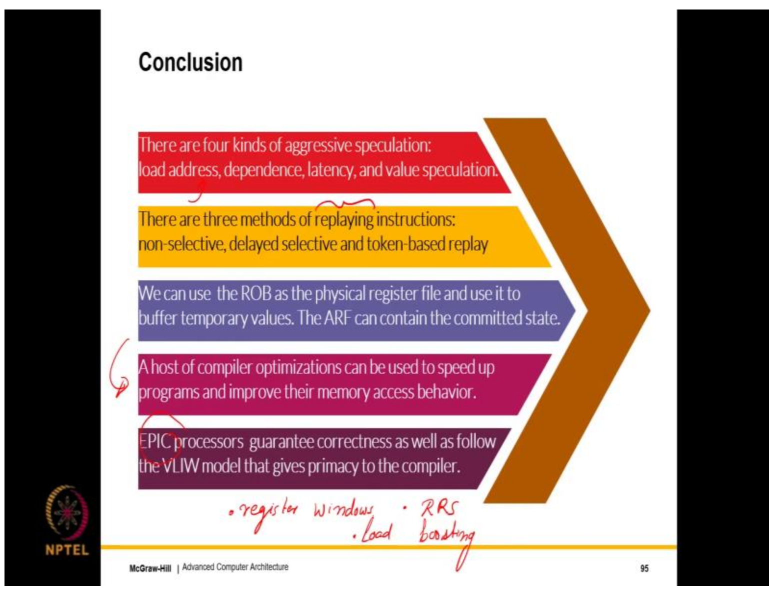
It will not be valid if there have been some intervening store to the same address which will signal an error basically. In this case, if there is no error we are fine then in a boosting of the load as in a successful operation otherwise what we do is all the instructions on the load and its forward slice that we took up at this point where its original point we again re execute the load and its forward slice.

So, this is the basic idea of boosting up a load which means try out do it early something similar this is something conceptually quite similar to let us say latency speculation or dependent speculation in LSQ, but of course, in that case we had a dedicated hardware we had a LSQ and all of that.

In this case we do not, but at least idea is quite similar conceptually at least if you see that the compiler is kind of boosting up the load, but it is not completely forgetting it, it is putting it in ALAT and any subsequent store to the same address that that will also check the ALAT and basically indicate that there has been an error.

But the error will be fixed in the sense if you restore the original point we will again check for the load, it is still valid if it has not been invalidated by a store then we will fix the error by reissuing the instructions.

(Refer Slide Time: 55:12)



So, this brings us to the conclusion of this chapter. So, I will quickly go through the five most important points. There are four kinds of aggressive speculation we have discussed all four in great detail. First is we can speculate on a load address no doubt, dependency in a load store cube, load latencies in the sense we can predict it will hit an L 1 cache and then do something about it and finally, value prediction.

So, these are the four major mechanisms. If there is a mistake we need not flush the entire ROB there are three methods of replay that we looked at. Non selective replay kind of shooting machines with an AK 47. Delayed selected slightly more selective reduces the amount of wasted work and token base replay where we have an exact idea of the forwards slices. So, this minimizes the (Refer Time: 56:14) statement.

After increasing the complexity, we reduced it. So, we first looked at hardware solutions which is the ARF based design where we use the ROB the physical register to buffer temporary values. So, we found that its simpler, but of course, in this case values are much larger and we are communicating values with whereas, the PRF based design whenever communicated values. Values are quite wide as opposed to a seven-bit physical register a value can be 64 bits which is quite wide.

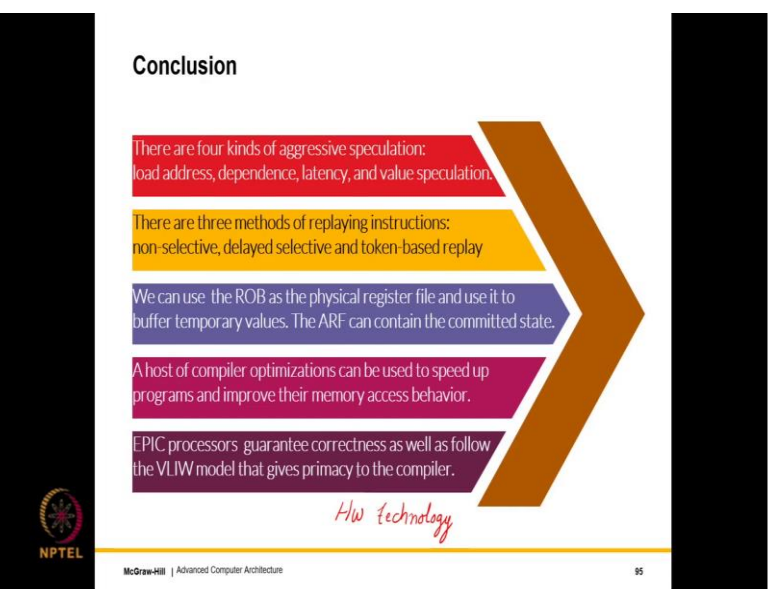
We further simplified we went down to the compiler level we looked at a large number of optimizations starting from common sub expression elimination on strip folding strength reduction to software pipelining. These ideas were conceptualized and an EPIC

processor. So, we discussed Itanium where we kind of saw everything come together. So, we saw a piece of hardware that supports quite advanced concepts it supports register windows for example tail recursion elimination which comes as a part of that.

It supports the idea of rotating register sets for automatically supporting software pipelining. It supports load boosting and a bunch of other things a bunch of other you know really cool features that it has. So, EPIC processors like Itanium did have their hay day they survived for quite some time and there was a lot of buzz around them.

But gradually they fell out of favor and one reason could be well I can only conjecture at this point because I do not have the information. By 2005 or so, the hardware technology was quite mature and it did not really require ones compiler support.

(Refer Slide Time: 58:03)



So, the hardware technology even with decent open source GCC like compiler support was able to give very good performance in that to it a very good let us say performance for watt or performance per dollar kind of price point. So, the industry really did not see the value of moving to a technology or a platform which was totally new.

So, you can see that you may even think about this that any industry has a fair amount of inertia whenever any revolutionary technology comes such as itanium. Even Itanium to a large extent was made compatible with existing ISAs and so, on. It is not that like that

effort was not done, but as I said many times many good and revolutionary ideas get stuck up in this inertia where the industry as a whole does not want to move.

So, something similar happened, but this is clearly not the end of processors like this. So, we will see many more such efforts in the processor design space, we just need to look out for great new ideas that will keep coming. So, this finishes chapter 5. So, our entire discussion on processor design is over now. So, we will next move to GPUs graphics process.