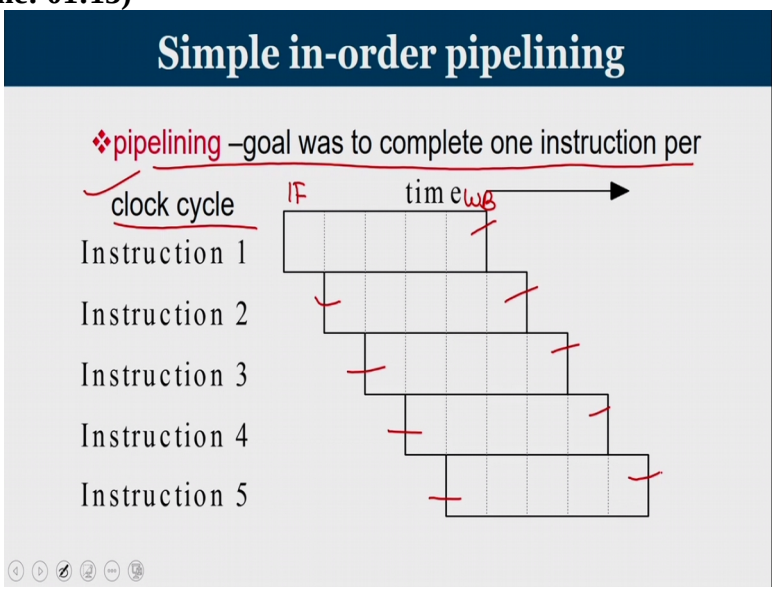
Advanced Computer Architecture
Dr. John Jose, Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati, Assam

Lecture 11
Advanced Pipelining and Superscalar Processors

Welcome to lecture number 11.In this lecture, our focus of studies is on advanced pipelining concepts as well as super scalar processors. Over the last 10 lectures, we have seen how an instruction pipeline works. And in order to improve the performance of instruction pipeline, we have introducing few techniques that will enhance its overall execution speed up handling with hazards handling with branches, and then how multiple execution works together multi cycle pipeline everything we have handled.

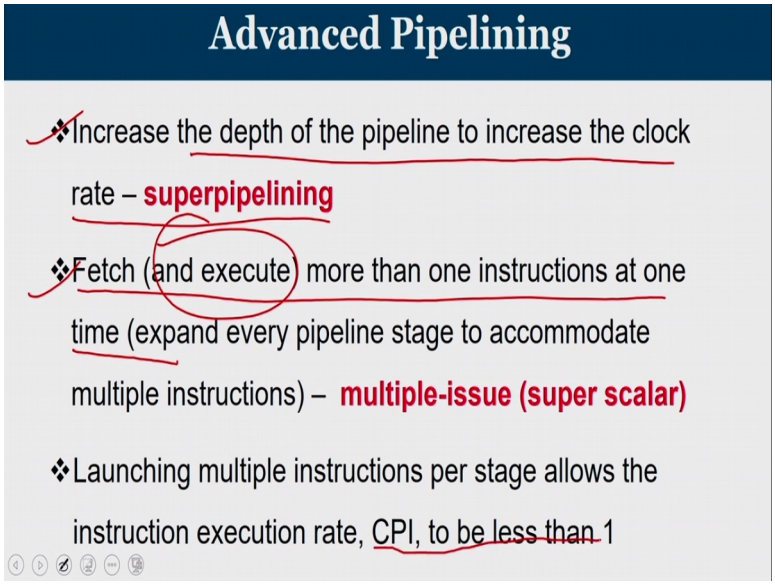
We will now focus on more attention into few of the advanced concepts in pipeline leading to super scalar processes.

(Refer Slide Time: 01:13)



We have seen what a simple in order pipeline is the goal of pipeline was to complete 1 instruction per clock cycle that was idea of simple pipeline, this is a 5 stage pipeline where we have seen in which an instruction is fetched every cycle and it is move to the WB stage at every cycle and we can see that every new cycle one more instruction is fetch and in every cycle one instruction is getting over under ideal circumstances.

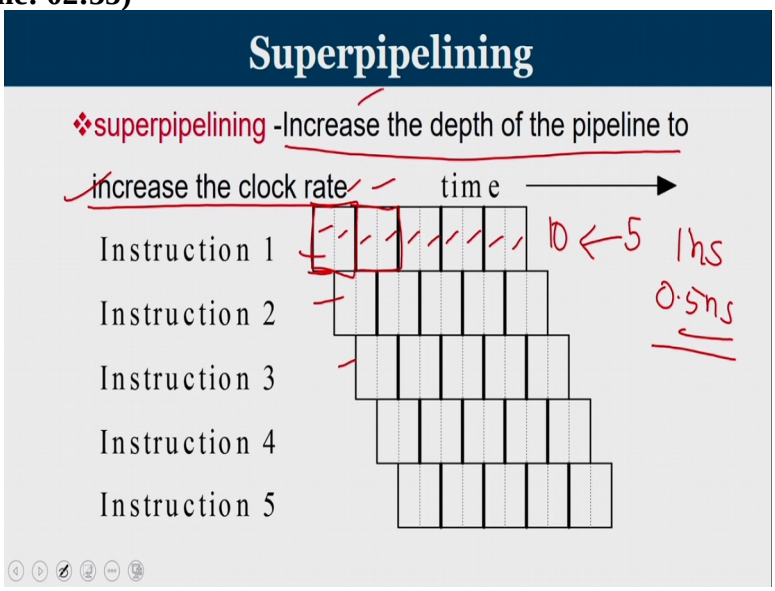
(Refer Slide Time: 01:40)



Now, if you want to increase the depth of pipeline, then we have to increase the clock rate and that concept is known as superpipelining and the next stage is known as super scalar when you fetch more than 1 instruction at 1 time, thereby we are executing more than more than 1 instruction in a time and that is known as super scalar instruction. So, we will have a quick look of what is superpipelining and what is super scalar pipeline.

Launching multiple instructions per stage allow instruction to execute with a CPI cycles per instruction to be less than 1. So on an average I need only less than 1 clock cycle to execute an instruction and that concept is known as super scalar pipeline. It is a bit tricky affair, we will try to see how super scalar technique is implemented.

(Refer Slide Time: 02:35)



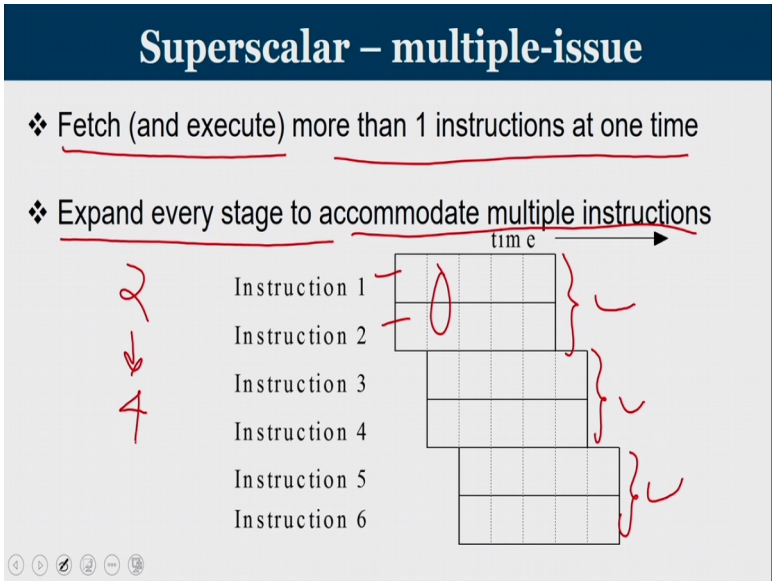
First let me introduce you to superpipelining where we are increasing the depth of pipeline by increasing the clock rate. So, imagine the previous scenario which is been shown by a bold arrow this was the time that is needed for completing instruction fetch and this is for instruction decode that was your cycle. Now imagine if instruction fetch should have completed in little less time and then we are going to divide the pipeline into let us say 10 stages.

Now we can see rather than having 5 stages now I have a 10 stage pipeline and each of the stage is going to take even lesser amount of time. That means in less than half of the clock I can complete one of the operation. That means I could initiate more number of the instruction in the same given unit amount of time and this is called the concept of superpipeline. For example, let us say our initial pipeline was working in 1 nanosecond clock.

Now if you are going to work in 0.5, nanosecond of clock then means instruction fetch happens at every 0.5 nanosecond leading to an every 0.5 nanosecond, one more instruction is getting over this is called increasing the depth of the pipeline. Now how can you achieve super pipeline, it is not that simple and straightforward. Whatever activities that we were doing.

I have to find out that activity has to be subdivided into 2 equals sub activities, and that need to be done across all stages in the pipeline. So instruction fetch essentially will be cut into 2, you just imagine like that, it is not possible to perfectly cut instruction pipeline into 2 like that, if I could subdivide all the operation in our initial pipeline to smaller sub operations and then trigger the clock at that rate and that will lead to super pipelining.

(Refer Time: 04:22)

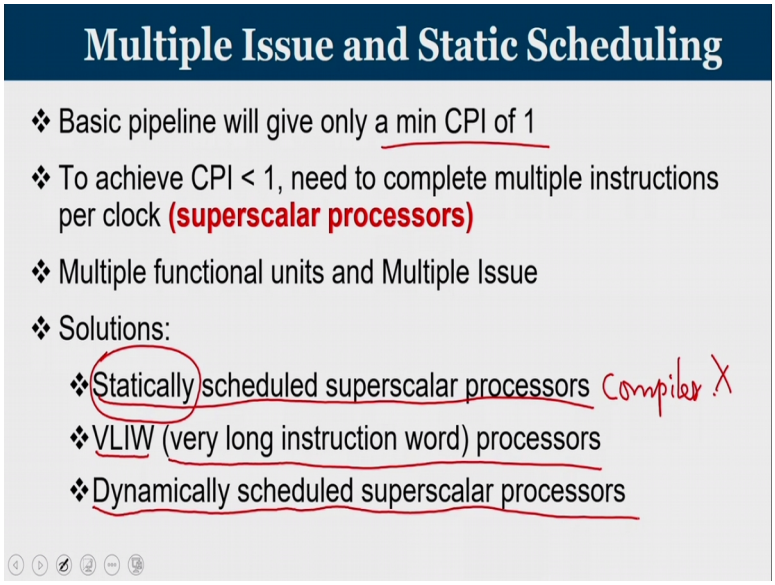


Now, let us see what is super scalar. Now super scalar pipeline is also known as multiple issue pipeline, where you are fetching and executing more than 1 instruction at a time. This means you have to expand every stage to accommodate multiple instruction consider this scenario, you have 2 instruction that are fetched exactly together meaning how to decode of them together eventually I will complete 2 instruction in 1 clock cycle.

So in every clock cycle, I am able to complete 2 2 instruction This is called double issue when the number of instruction fetch this eventually to decode 2 instruction I can make it to 4 this is called 4 wide issue. So there will be 4 instruction that are going to get completed in unit time. Essentially the number of instructions that are handled per cycle is larger than 1 and that will make you to operate on super scalar.

So, the word super scalar basically means if the CPI value cycles per instruction is less than 1 or IPC value instruction that you can do per cycle IPC value if it is larger than 1 that is called a scalar processor.

(Refer Slide Time: 05:29)



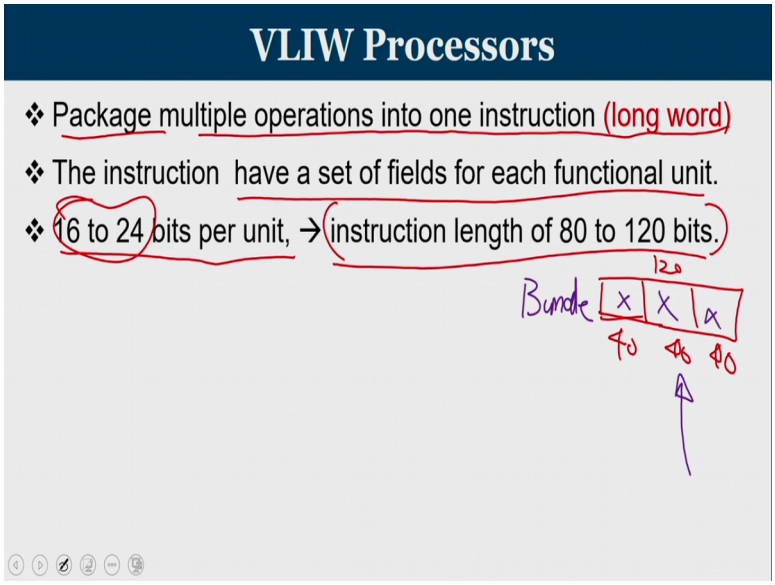
Your basic pipeline we give only a minimum CPI of 1 to achieve CPI less than 1 we need to complete multiple instructions per clock cycle and that is what is super scalar for facilitating super scalar processing you need to have multiple functional units and a issue unit that is capable of issuing multiple thing and basically there are 3 approaches to handle this, one is known as statically scheduled super scalar processors.

You are going to execute multiple instructions together, but which all thing to execute that is been statistically determined by compiler. So compiler has a big say in statically scheduled super scalar processors. Essentially you have to run multiple instructions together you have to fetch multiple instructions together everything will happen multiple now which all things will be grouped together which are the ones that are going to be fetched together.

It is determined statically so compiler has a big say, but it is not that a popular approach because of the overhead involved in compiler and it is extremely difficult to find out the combinations that have to be run together. The second one little wider used version is known as VLIW it is known as very long instruction word processors, which we will learn in detail later.

And the last one most common one is known as dynamically scheduled super scalar processors. Let us have our focus our attention to VLIW processors.

(Refer Slide Time: 06:55)



In VLIW processors, we are packaging multiple operations into 1 instruction and that is known as a long instruction word, let me tell you, our normal instructions were like this, It has an opcode portion and then an operand 1 and operand 2 this is our conventional instruction. Now, when I talk about VLIW processor, you have a very long instruction word very long instruction word out of it.

So, let us say divided into 3 different format this may be a load instruction and this may be an ALU instruction and this may be a floating point instruction. So, this method either it is a load or a store operation followed by whether it is an R1 R2 whatever and here it is the address portion. So each instruction each integer each very long instruction word should have a load instruction ALU instruction and 14 floating point instruction.

Now what happens is all of them goes together into the execution unit. So, this instruction load and ALU instruction and the floating point instruction work exactly parallel for this 2 happen, there should not be any dependency between them. And it is the role of the compiler to bundle to group instructions like this. This whole concert is known as very long instruction word. So from a sequence of instruction I have to find instructions of 3 combination needs or a group of 3 instructions together which is known as instruction bundle as well.

So the instruction have a set of fields for each functional unit, that is what we have seen now 16 to 24 bits is used per unit and instruction length can vary up to 80 to 120. So let us say this is my instruction, which is 120 bits. Now, I divide them into 3 set each of 40 bits. So, my first 40 bit of my very long instruction will be part of a load store operation, the next 40 bit has to

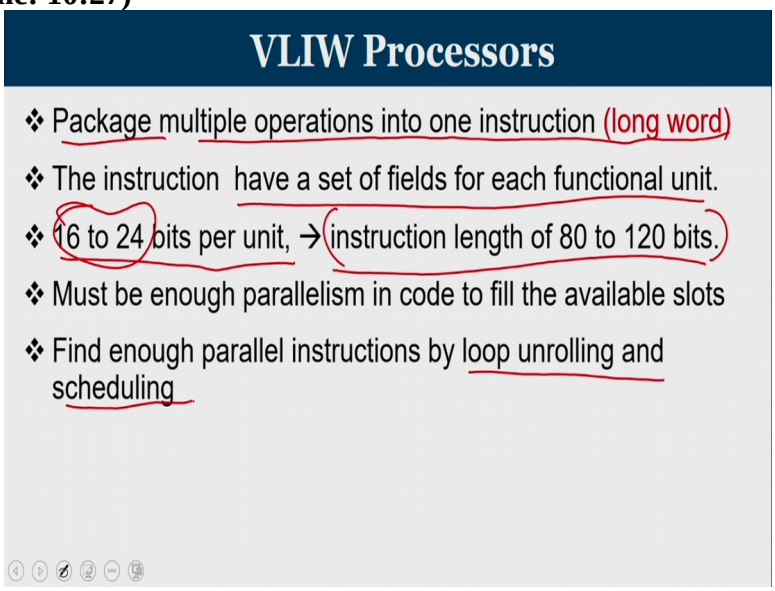
be interpreted by the ALU unit and the last 40 bit has to be interpreted by the 14 floating point unit.

So, instruction length is very huge, which consists of multiple smaller instructions grouped together. Now, you should have enough parallelism in the code. Now to understand that, so, think of a case with whatever instruction that you get all the instruction has to be grouped into this. So any bundle this is known as a bundle. So, any bundle need to have 3 entities one should be exactly a load store instruction, other one exactly should be an ALU instruction and the third one exactly should be a floating point instruction.

Now if you look at the concept of the bundle, for every program, it is a duty of the compiler to find out bundles. What if I do not have any load store instruction for the next hundred in hundred instruction. So, it is very difficult to fill up all the slots. So that will lead to sometimes certain slots being unfilled or it is going to be empty slots. So when I do not have a floating point operation.

One by third of this particular load instruction word that I mentioned goes unused, this is a challenge.

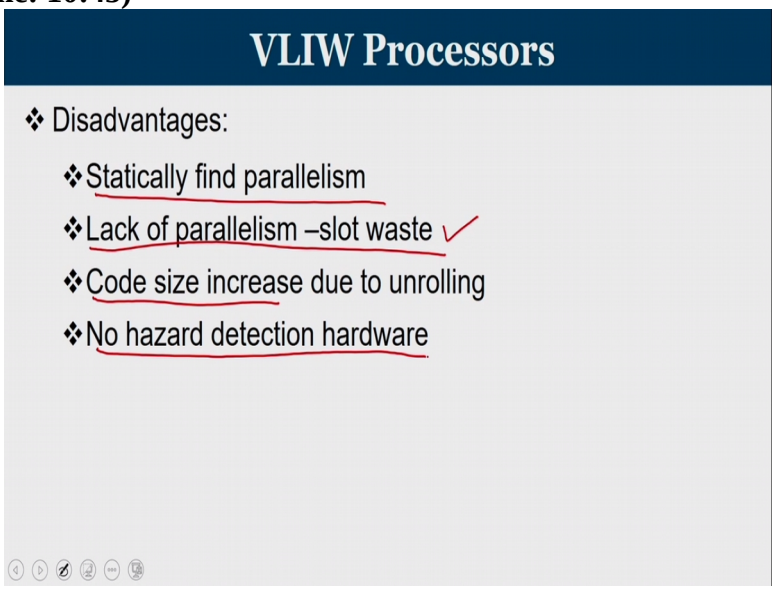
(Refer Slide Time: 10:27)



So finding out enough parallelism for the VLIW word that has been already mentioned is also a challange. So, there should be enough parallelism in the code to fill up the available slot and find enough parallel instruction by loop unrolling and scheduling and generally, VLIW

bundles are formed whenever we do not have enough instruction within a single block then you perform loop unrolling and do the scheduling such that I can view in VLIW.

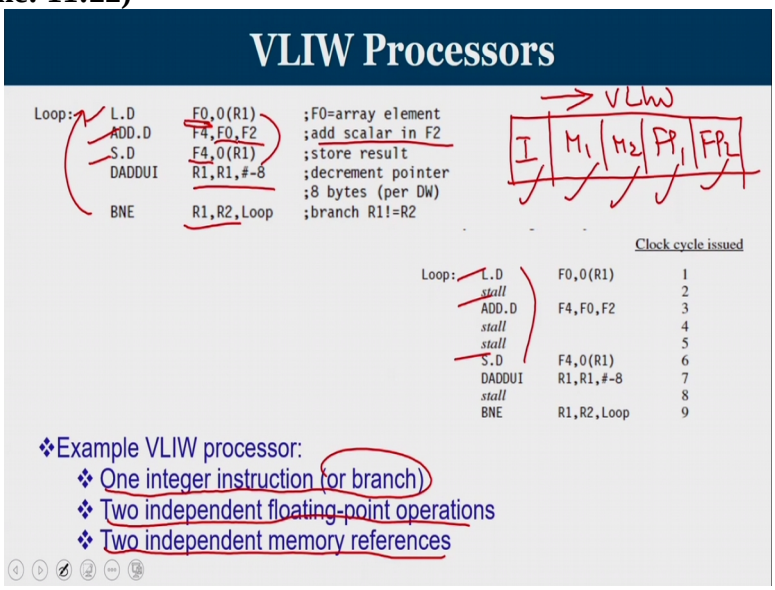
(Refer Slide Time: 10:45)



So, what are the disadvantages VLIW processors we have to find out parallelism statically and all those instructions, which are nondependent can be run in parallel. And sometimes when there is no parallelism available between instruction, then the VLIW slots are going to be wasted. And since you are performing unrolling and scheduling, the loop unrolling is going to increase the code size and you cannot detect hardware in parallel.

Because these are the instruction which are to be run together. So all hazard detection had to be done at the compiler level.

(Refer Slide Time: 11:22)



So, consider this VLIW processor where we have a load instruction, value was loaded to F0 and then you are going to ADD F2 to F0 it is called a scalar adding add a scalar value that is present in F2 to F0 and you are going to store the resultant in F4. Now whatever is the result that is there in F4 you are storing into the same location. So 0 of R1 is the location. So ideally, what we are we doing from the memory, take a value into a register known as F0. Now ADD F2 on to it, and then you store back in the same location.

That is what the operation is all about. And then since you have to repeat in a loop, you are going to decrement the value of R1,

$$R1 = R1 - 8$$

and then you check whether the value of R1 is equal to a loop or not. And then the loop is repeated. Now consider the case that this is how a general program looks like you have a load and then followed by an ADD and then store. So, all these 3 instructions are dependent. So I cannot run them parallelly. So there is not enough parallelism that is available inside this basic block.

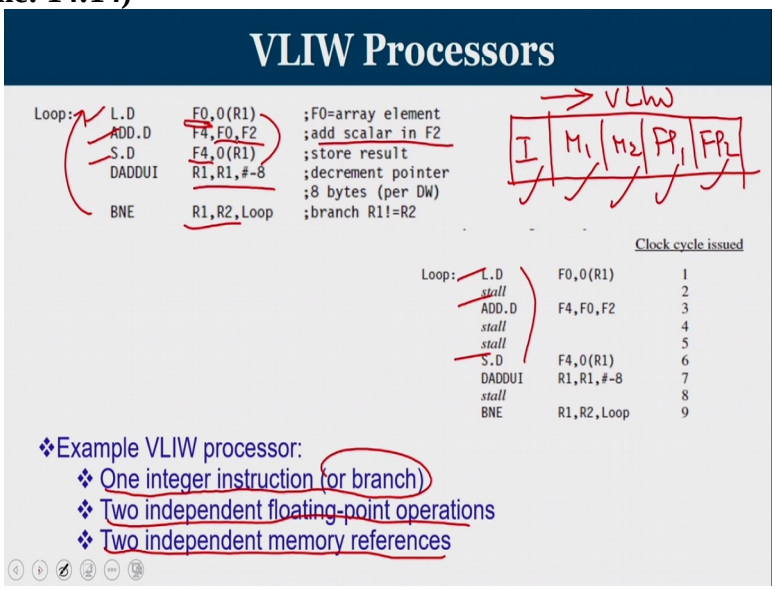
Now consider you are going to work on a VLIW processor, where each instruction bundle has an integer of unit which can take care of branch as well 2 independent floating-point operations and 2 independent memory references. So, you are in very long instruction word VLIW has 1 is an integer unit then you have memory operations memory 1 and memory 2 and then you have floating point operation FP1 and FP2. So, an VLIW instruction a VLIW instruction here have 5 components.

So it is capable of running 2 memory operations 2 independent memory operations parallely to independent floating point operations parallely and an integer or a branch operation together all these 5 can work together. So, what is the role of compiler that you have from among the available program, you find out file these components there should be a integer or a branch instruction and then there should be 2 memory instructions and to be 2 floating point instructions.

All these are put together to form the long instruction word, but in the code that is been shown there is 1 load, 1 ALU operation, then there is 1 load 1 floating point unit operation, another 1 store and then you have a branch operation and then you have an ALU operations that are total of 5 instructions given in the loop and the loop is being repeated. So, it is very

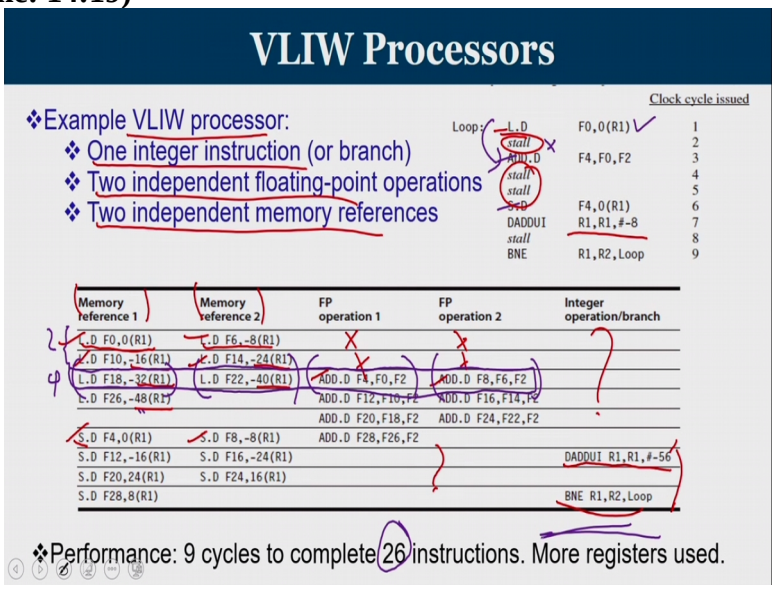
difficult to make use of 2 multiplication unit, 2 floating point multiplication unit work together. Let us now see how it has been achieved.

(Refer Slide Time: 14:14)



We have already mentioned it is possible only with the help of loops.

(Refer Slide Time: 14:19)



So, the same example, we are going to have a VLIW processor with 1 instruction of integer type 2 independent floating point instruction and 2 independent memory reference instruction, this is the program we are going to perform loop and rolling. So this is of VLIW pattern see, you have 2 memory reference. So in 1 row is your 1 long instruction word. So each of the row there will be 2 memory references 2 floating point operation and 1 integer or branch operation.

So, what is the first instruction. first instruction is load. So, when I look into the first instruction, I can find out this is the first instruction load F0 0 of R1. Now, this load and its corresponding ADD has to be separated by 1 cycle. So this is the ADD that will go so this is cycle number 1 cycle number 2 3 4 5 like that, these are the cycles. So, there is a gap of 1 cycle which is already shown here between the load and the corresponding ADD.

The ones the ADD is over, it takes 2 cycles to perform the store. So this is the first store. So these ADD and store are dependent, these load and ADD are also dependent that is what a dependency that I would like to show here. Now, since 2 memory the references are there, if I unroll a loop for one more time, then you will get the next set of load, load F6 -8 of R1. And these 2 nodes are operating on 2 different memory locations.

I can say that they are 2 independent memory references. So I could do both the loads together. And both the ADDs also together both the floating point ADD also together because I can perform 2 floating point operations exactly same time. And that result in 2 stores coming together when you unroll again, this is the next set of values which is acted upon -16 of R1 and -24 of R1, -32 of R1, -40 of R1 and -48 of R1.

So, literally there are 7 iterations and proportionately we can see that there are many ADD operations that is also coming and if you look at each individual pair of load and the corresponding ADD a delay of 1 stall is maintained. And between ADD and store, another delay of 2 is been maintained. Since you unrolled this operation of R1

$$R1 = R1 - 8,$$

we need to do only once

$$R1 = R1 - 56$$

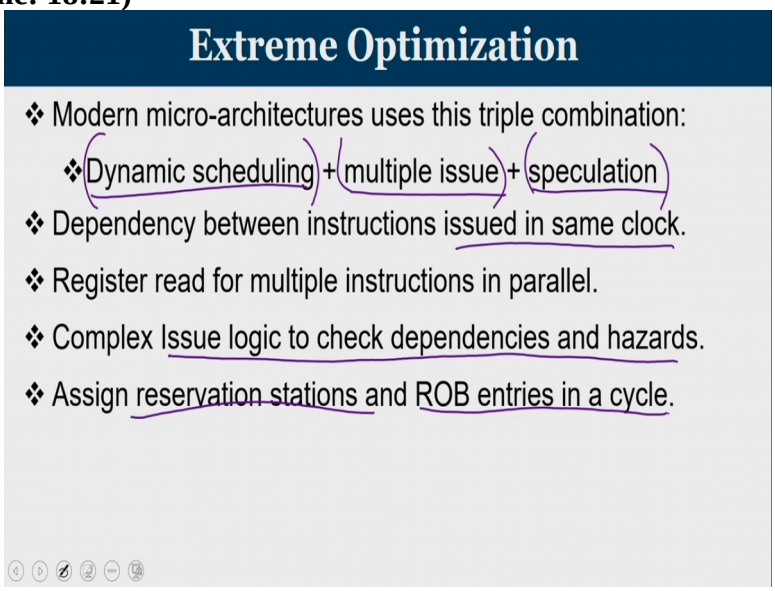
because all other pointing to R1 are automatically taken care of by the loop and rolling and then you have the branch statement. So there is a stall between them.

So, we can see that some of the slots are actually going idle, because there is not enough parallelism available even though I perform loop unrolling, many slots are going to be idle, but you can see that there are certain slots in which both the memory references are being used together. For example, consider this case in whatever instruction that is issued in cycle number 3, I could perform 2 memory operations.

As well as this is memory operation 1, this is memory operation 2 and 2 floating point that operation. So essentially, in cycle number 3 I am performing 4 operations, whereas in cycle number 1 and 2, I am performing 2 operations each, that is here I am performing 4 operation, but when I move to cycle number 4, I have only one memory operation whereas I have 2 floating point ADD operation.

Like that, wherever possible, we are trying to execute parallelly and this whole concept is known as very long instruction word processor. So, essentially, we took only nine cycles, but if you look at you could complete 26 instructions, one of the drawback is we require more number of registers of course that is a by-product of Loop unrolling.

(Refer Slide Time: 18:21)



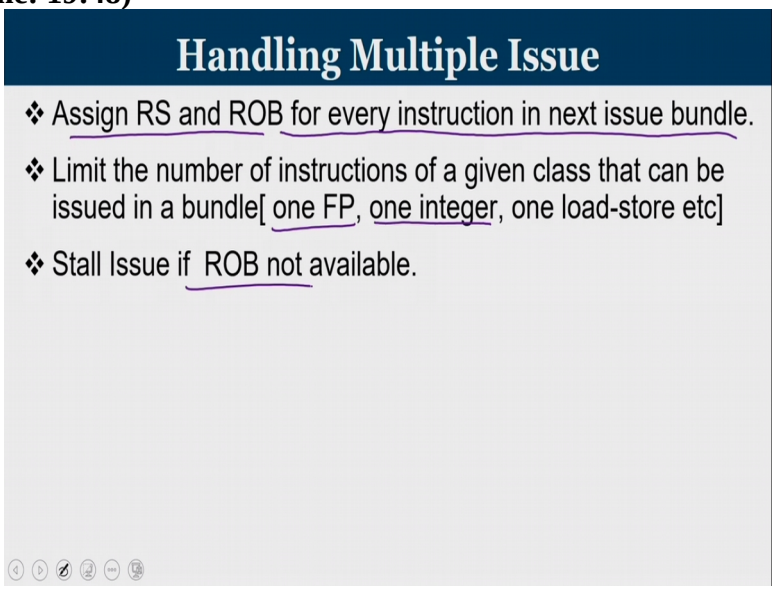
Now what is known as extreme optimization under all these contexts, we have learned lot of techniques, so can we put all of them together. So modern micro architectures use triple combination use dynamic scheduling, multiple issue that we have just learned and speculation. So use branch prediction, dynamic scheduling like normal Tomasulo's algorithm and you make it super scalar.

This is what you see in the latest processors that we are using in our systems. So, what are the challenges when you do this extreme optimization, you need to detect dependency between instruction that are issued in the same clock cycle, you are going to issue multiple instructions together exactly in the same bundle, what about dependency check between them. So, if there are instructions that are dependent, they cannot be issued in the same cycle they should be part of adjacent or different bundles.

So dependency detection is a challenging task and that has to be addressed. And then we need to perform register read for multiple instructions in parallel and complex issue logic. Because this dependency checking and hazard detection will make the issue logic little complex. And then before issue we have learned in Tomasulo's algorithm, we need to look reservation station.

And since it is speculation, we need to find out reorder buffer entries in a cycle. So when you issue multiple instructions, you need to check availability of reservation station for multiple instructions and availability of reorder buffer entries for multiple instructions.

(Refer Slide Time: 19:48)

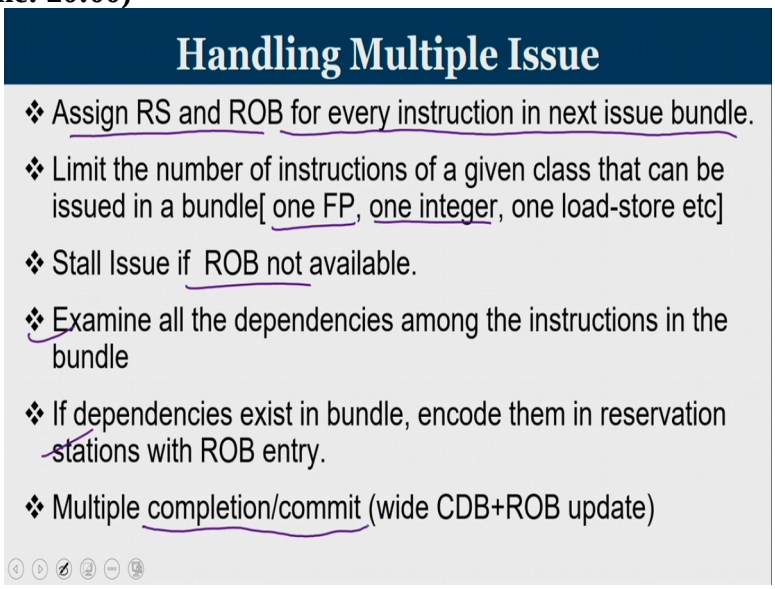


And handling multiple issues always a challenge assigning reservation station and reorder buffer. For every instruction that we are going to issue in the next bundle or limit the number of instructions of a given class that can be issued in a bundle, like what you are telling, you are telling that we can have only 1 floating point instruction permitted, 1 individual instruction permitted 1 load store instruction permitted.

And if reorder buffer is not available, of course, you have to stall. So I was just trying to give a flavour of what are the challenges that an architect need to take here or one of the challenges an architect face while going for the extreme optimization handling of this dependencies handling of the hazards, checking off the reservation station availability, checking of reorder buffer, all these are going to be a complex task.

So all these issues are addressed in what modern day processes we are working on.

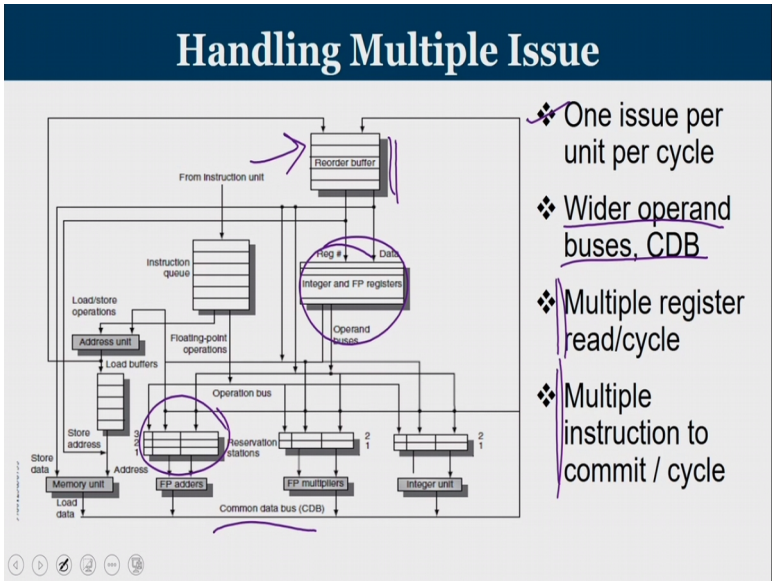
(Refer Slide Time: 20:00)



So and so we have to examine all dependencies among instruction in the bundle. And if dependencies exists in a bundle, you have to encode them into reservation station with appropriate reorder buffer entry. And then once you go for super scalar technique, you know that there are multiple instructions that are going to complete at the same time. So multiple complete or commits, that means multiple functional unit can produce your result exactly at the same time.

So if multiple functional units are going to produce your result, then if there is only one bus, if the unit number 1 and unit number 2, are going to put the result together, then it leads to chaos. So the CDB the common data bus should be wide enough to accommodate results of multiple functional unit together. So when you have multiple effects support multiple decode support, multiple execute support, you need to have provision to accommodate multiple instruction values on the common data bus.

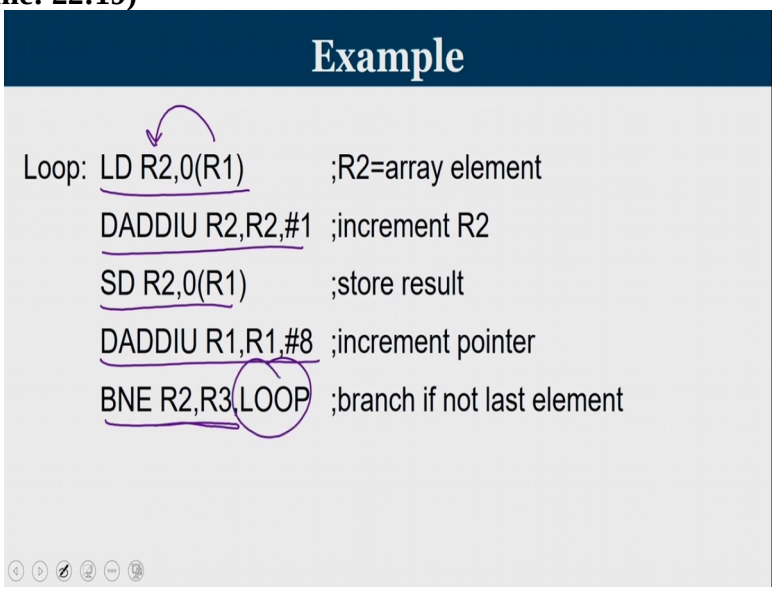
(Refer Slide Time: 21:40)



Essentially, that means we need to have a wider CDB. When you come to handling multiple issue, we need to issue only one instruction, 1 issue per instruction per unit per cycle and wider operand buses are required because you need to fetch multiple register values together. And that means multiple register read and then multiple instruction committed is also been required.

So essentially, your CDB should be bigger, there should be multiple reading that should be done on your integer and floating point registers. And multiple commit also has to be done. And the reading of reorder buffer as well as the reservation station also, there should be provision to multiple blocking.

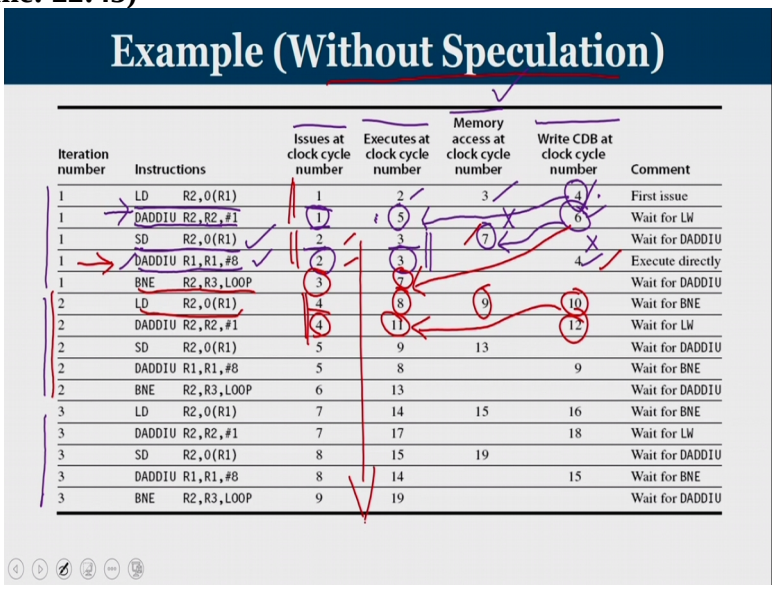
(Refer Slide Time: 22:19)



So consider this example what we have here, so you have a load operation, you are going to load an element into R2, and then I am going to increment the value of R2. And then I am

going to store the value. So it is basically take the value that is present in the array incremented by 1 and store it back and then you go to the very next location and then check if R2 equal to R3 and that is a loop termination condition.

(Refer Slide Time: 22:45)



Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9	19			Wait for DADDIU

Now, let us see how to run this one without speculation. So this is the time cycle description of the task you are going to unroll this loop 3 times. So these are the instruction, that are part of first iteration, second iteration and third iteration. Now this source when the instruction is issued, when the instruction execute, when memory access takes place in the case of load and store, and when are you going to write into CDB.

So when it is there is no speculation, then that is CDB that is required whenever you write the result into CDB. That is the time in which the result is obtained. You take the first instruction, let us say I am going to issue in cycle number 1, the execution happens cycle number 2, since it is a load instruction. So what happens in the execution in the execution stage, you are going to compute the effective address.

So

$$0 + R1$$

is computed that happens in cycle number 2, and in cycle number 3, you go to the memory location, which is given by the address

$$0 + R1$$

So the end of cycle number 3, the operand is ready and cycle number 4 you write into CDB, eventually, they are going to write into R2. Now, since I am using multiple issue, the second instruction also I am going to issue in cycle number 1.

But I cannot execute the second instruction because second instruction is operating on R2 and that is a value that is available only at the clock cycle number 4. So, second instruction even though I am going to issue it in clock cycle number 1, it will start execution only at clock cycle number 5, because of the dependency factor. So, this will happen only after this clock cycle.

Since the first load instruction right into CDB at clock cycle number 4, the first ADD Operation DA ADD ADD DADDIU that is direct ADD unsigned integer that will happen only at clock cycle number 5. Since it is an ADD operation, it is not going to make use of memory access. And once you perform the ADD on 5, you can write the result on 6. And what about store. Store get issued on 2.

So here the assumption is I am using a dual issue at every cycle I can issue 2 instruction store also can be issued at 2 when you issue at 2 you can execute a 3 to computation of address happens at 3. And when are you going to perform the memory operation memory operation happens only after the result is available. So storing happens at 7 and there is no CDP writing as far as store is concerned.

And then you are going to perform since there is no speculation I am going to run the loop and prior to running the loop, my R1 values are adjusted so that also issued at 2 the issued at 2 I can do it at 3 both of them happens at 3. one is a memory operation, other one is a integer operation. So, we have 2 separate functional units both happen together.

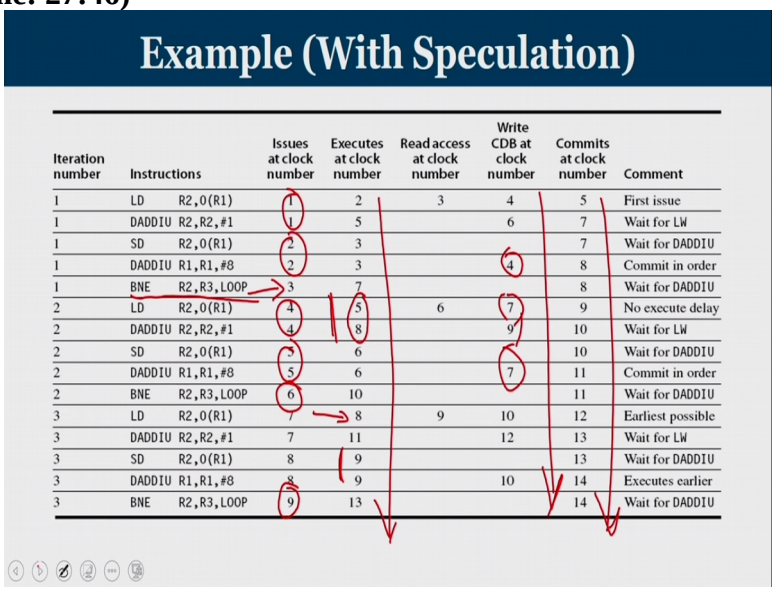
So when you do something at 3, I can write it on 4 see, now you see this instruction, this particular instruction, which was issued at 2, it got completed at 4 but instruction store you should at 2 it is completing the store operation at 7. So clearly you can see there is an out of order execution. And once this is over now I am going to have a loop and remember one case.

Since all the loop statements are executed together in one slot means even though it is a dual issue processor, where in every cycle 2 issues are permitted along with the branch, no other instruction is issued. Because when it comes to branch happening or not happening, clearing or going back status, doing wrong prediction, whatever it has to be done. So generally, you are remember the fact that along with their branch. No other instruction is issued.

So if I do a branch at 3 it is operating on the value of R2, so branch can happen only after the value of R2 is ready. And when will the value of R2 is ready, at this point. So the value 6 and branch can happen at 7. And now I am going this is without speculation when it is without speculation, every instruction that is coming after a branch these one will happen only after the outcome of the branch happen.

So you know the outcome of branch that clock cycle number 7, let us say going to the next titration, these are issued at 4. One will happen at 8, 9, 10 that is computation of address accessing memory and writing into CDB. Whereas the second one 4, 11 why this is 11 because its 10 to 11 dependency and then 12 we are going to write back even the same way if you continue. I request you to go through the remaining entries and make yourself comfortable and thorough with what happens and what are the cycle in which these things are happening.

(Refer Slide Time: 27:46)



Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#8	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

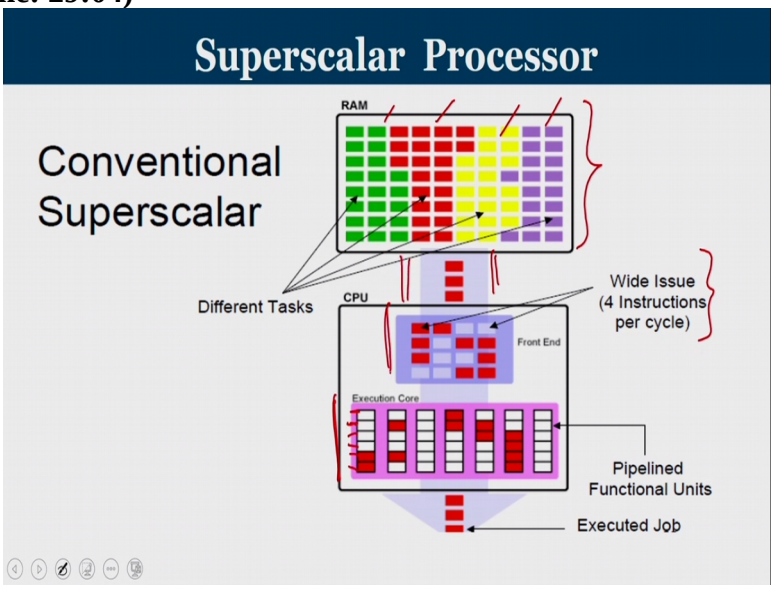
This is an example without speculation. Now, what will happen if you am going for a with speculation, if it is with speculation, I am not waiting for the outcome of a branch you can see there are entries after the branch also which are executed we are going to write into CDB

before the branch happens but you commit only in order So, when you go for speculation there is one more field that comes commit always is in order.

So, here also we have dual issue you can see the 2 instructions are issued at any given point of time here also along with the branch no other instruction is issued. So at cycle number 3 first branch is issued cycle number 6 second branch is issued and cycle number 9 the third branch is issued and if you look at the execution number, you can see they are not in any sequential order.

Some which are issued later will execute earlier than the other one, we can see that it is not an order and the time at which they are going to complete it also would not be in order you can see certain instructions are completing also out of order, but the commit always happens in order. So, this is an example of the code that loop code with and without speculation. Now, we move on to the concept of super scalar processor from a logical understanding viewpoint.

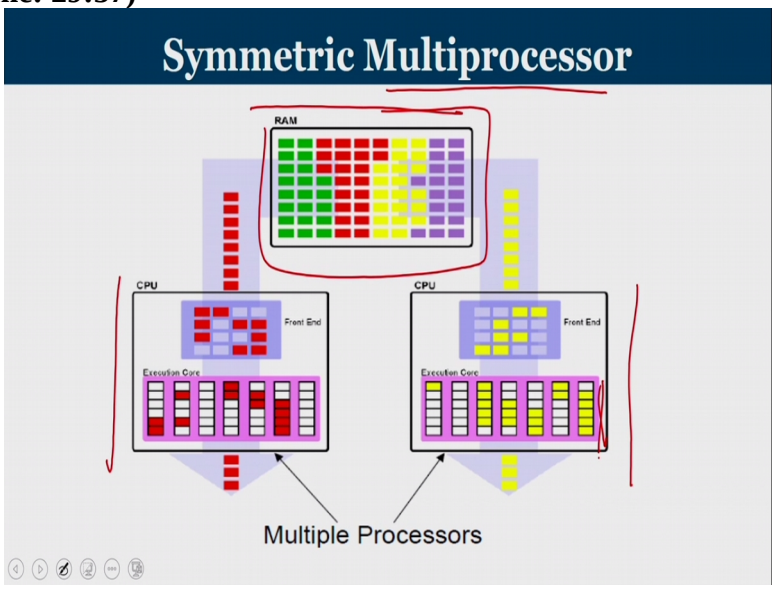
(Refer Slide Time: 29:04)



So your conventional super scalar processor what is given in the diagram is you have a RAM you can assume that you are main memory consists of 4 different programs shown by 4 different colors that is a green, red, yellow and violet and each of the smaller boxes represent instructions. Now in super scalar what we do is you are bringing multiple instructions together and this is an execution code it is a pipeline execution unit.

Meaning and instruction which is here it moves to this unit like that it takes multiple cycles for the instruction to complete the execution so many execution units are there are all connected. So you can imagine I am using your 4 wide issue for 4 instructions are issued in a clock cycle. Conventional super scalar for the idea is multiple instructions are fetch multiple instructions are executed. And that what happens

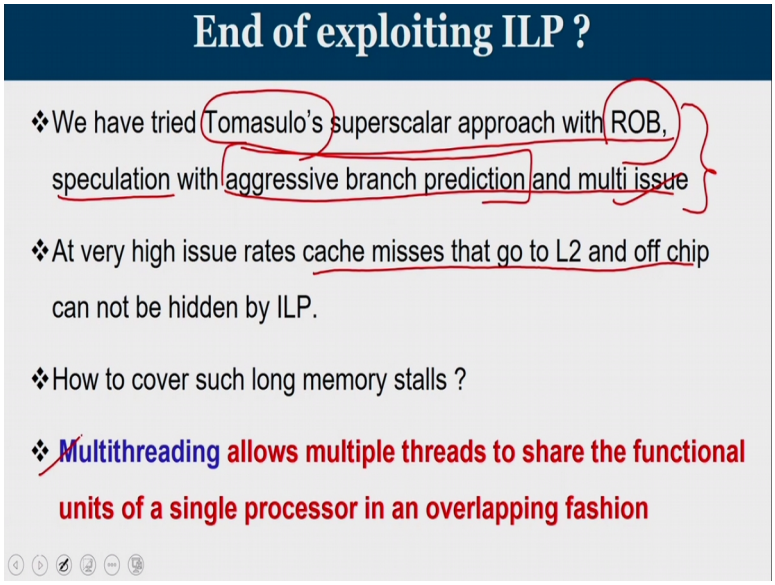
(Refer Slide Time: 29:57)



Now we are looking for symmetric multiprocessors, let us say I am going to talk about dual processor they are sharing the same memory. So the memory unit is common and I have CPU 1, I have CPU 2. CPU 1 is taking care of the red program, CPU 2 is taking care of the yellow program, both are internally super scalar processor so you can see that I am fetching multiple instruction I am executing multiple instruction.

But it is symmetric, so 2 super scalar processors working together on a common memory that is called symmetric multiprocessor.

(Refer Slide Time: 30:30)

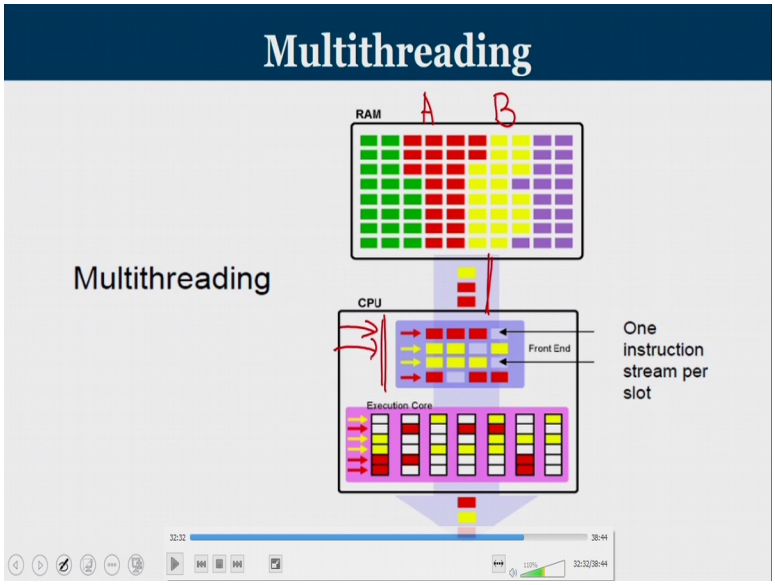


With this are we going to come to the end of ILP know we have tried our Tomasulo's superscalar approach with reorder buffer speculation and aggressive branch prediction and multiple issue that is a whole story all about. Tomasulo's algorithm is taking care of dynamic scheduling speculations is handled by reorder buffer and then you are using branch predictions to take care of control hazards.

And now you increase the number of functional units to facilitating multiple issue. At very high issue rate, you can have cache misses because so many instructions which are fetched and some of the misses will go to L2 and some of them can be off chip as well, we will learn about memory more in subsequent lectures. How to cover such long memory stalls will deal later.

So, we are going to deal with multithreading whenever one particular processor or one particular set of programs is encountered with misses can the processes shift to another program and run.

(Refer Slide Time: 31:35)



So, multithreading allows multiple threads to share the same functional unit of a single processor in an overlapping fashion. So, this is what is known as multithreading. So, so far what we are dealing were conventional super scalar and we are talking about 1 program at a time. Now look at the case you have your red program that is program A and yellow program B, both are brought together. So you bring multiple instructions of A in this clock cycle next clock cycle you bring multiple instructions of B.

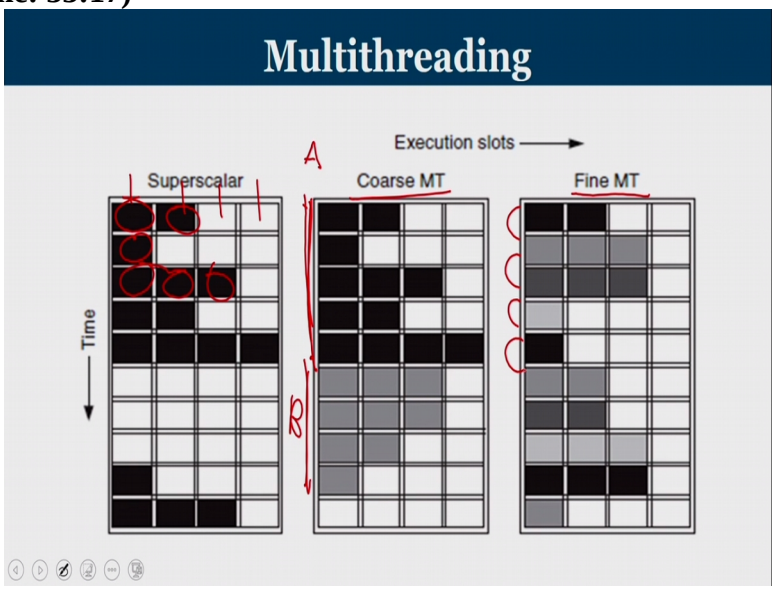
So you decode multiple instructions of A. So this is the decode unit of the issue unit if you look at horizontally, either it will be white, that is the empty slot or it will be red, red means you have instructions of the red program that is currently being issued. So either it is a red in a row or it is an yellow in a row. And if you look at the execution unit also it contains instructions belonging to multiple programs. But given 1 clock cycle, either it will be red or it will be yellow and that is known as multithreading.

So I will rephrase multithreading once again, imagine you have multiple programs that is stored in your main memory and your processor has a lot of functional units. Sometimes all the functional units may not be needed as far as 1 program is concerned. So in 1 clock cycle, I bring that a 4 instruction from 1 program A issue it then they are going to execute. In the next clock cycle rather than fetching from A, I am going to fetch from B.

So any dependency between instructions of A because of this context, because of the switching between instructions that also will take care of you are normal stalling. So this switching that happens and that is what is known as multithreading. So if you look at the

hardware, certain clock cycle hardware has been used by program A certain other clock cycle, this hardware has been used by program B.

(Refer Slide Time: 33:17)

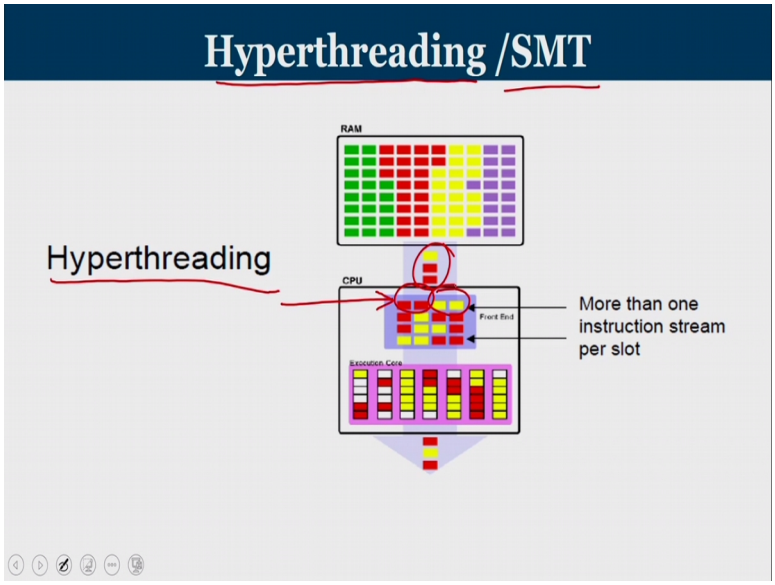


So this concept of multithreading is very important. Let us say there are 4 functional unit that you have. So in each cycle, some of the functional units are being used wherever you have white that is basically empty. Now, multithreading means for some time I am running this program A which is shown by dark red colour. And some time I am running program B which is on a lighter grey shade.

So is the switching between to, let us say I am going to run for 5 clock cycles and then I shift to B, so A is operated on 5 clock cycles then I operate on B for 5 clock cycles. So if the switching from A to B is after few cycles, then I call it as Coarse multithreading and find multithreading means that every cycle, there is a switch and not going to operate on program A only for next few clock cycles.

So the switching between the task is very frequent, we call it as fine grained multithreading the switching is not that frequent, we call it a coarse grained multithreading.

(Refer Slide Time: 34:21)

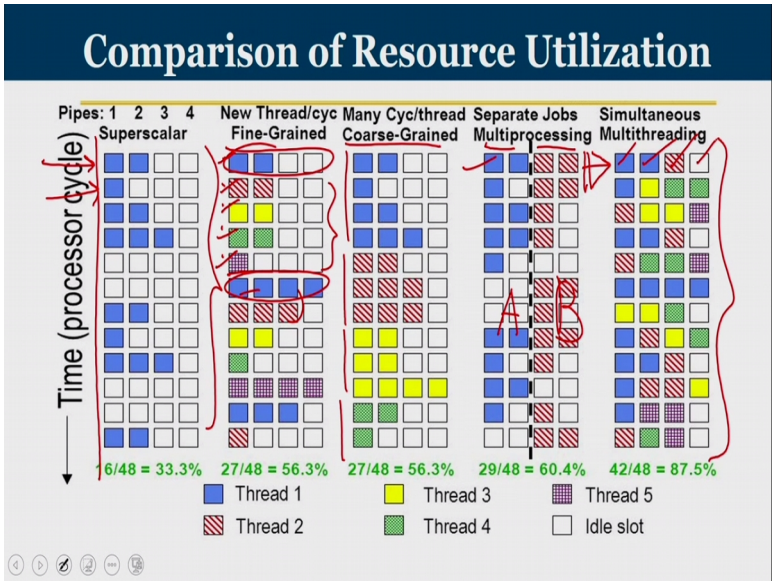


We are moving into the next level of super scalar processing. So this is known as symmetric simultaneous multithreading or hyper threading. In this case also we are bringing multiple instruction. But if you look at I am going to issue 2 instructions of red and 2 instructions of a yellow. That means in a given clock cycle itself, multiple instructions I am going to handle this is known as hyper threading.

Let us say what is the difference between hyper threading and multithreading in a multithreading what we do is you bring 4 instructions of program A, you decode 4 instructions, and then you are going to give. Let us say there are 10 execution units, I cannot satisfy all this ten, because my instruction A or instructions from A may not be reading all these functional units.

Now can I do little better can I bring 2 instructions from A can I bring 2 instructions from B mix it together issue it together such that if I look at my functional units at any given point of time, it consists of instructions from A as well as instructions from B, this hybrid mixing is known as hyper threading. And it is one of the most popular technology that has been used in almost all the latest processors that we are using in our laptops, or Desktop PCs. And that is all about hyper threading.

(Refer Slide Time: 35:44)



So this is a comparison chart where in the left said we have a super scalar processor, super scalar means multiple instructions are running parallel. So wherever you see this blue color, these are all instructions that is running parallel. And given a clock cycle 2 blue instruction here there is only one blue instruction. So these instructions will go through all these functional units. At the end it is going to come a lot of slots are wasted.

Now if you look into what is fine grained, the multithreading and I am not handling with only blue program and handling with many different types of programs. So go to blue, I go red I go to yellow then green then violet then only come back to blue. So at every cycle I switch between many of them in sometimes I am able to make use of all the 4 functional units like this.

But in some time only 3 still I have slot wastage. This is called multithreading. Now the same multithreading rather than switching every cycle let us say continue with blue for 3 or 4 cycles, then a switch to red for 3 or 4 clock cycles, then to yellow then to green, that is called coarse grained multithreading. And multiprocessing means I am running blue and I am going to run you are red in 2 different processors.

So this is processor A and the other one is processor B both A and B are going to access the same memory and that is known as symmetric multiprocessor and the last one is simultaneous multithreading at any given point of time, different functional units will have instructions from different instructions stream and that is known as simultaneous multithreading or hyper threading.

So, this slide give you an overall picture about what is the difference between conventional super scalar versus fine grained multithreaded processor versus coarse grained multithreaded processor was a symmetric multiprocessor and hyper threaded processor. And with that, we come to the end of today's lecture wherein we learned about what are the extreme optimization cases, with speculation and then how you do multiple issue the concept of super pipelining and super scalar processors we learned that and the concept of VLIW, very long instruction word were in compiler is going to take a sequence of instruction, pull it together wherever there is no dependency and issue it as a bundle.

We learned about super scalar processor in general and symmetric multiprocessor then, about different types of multithreading, fine grained and coarse grained and we conclude with what is hyper threading. Thank you