

Computer Architecture
Prof. Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

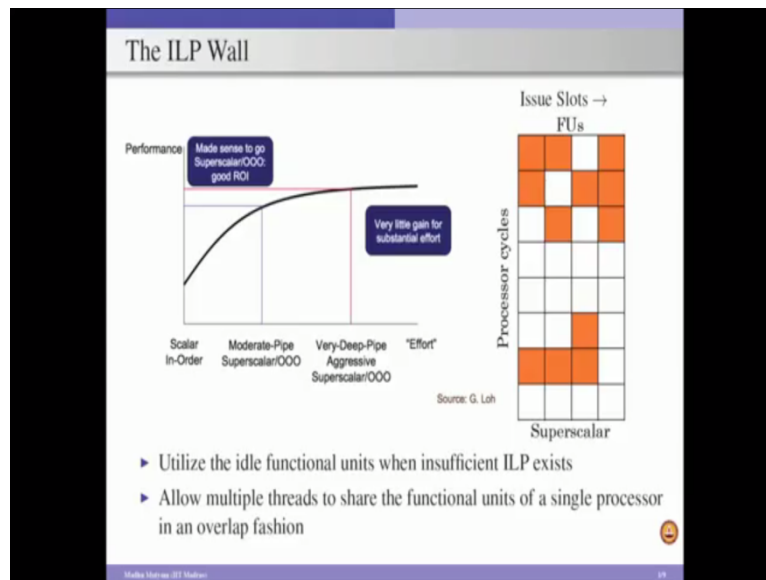
Module – 08
Lecture – 27
Multithreading

So, far in the computer architecture course we have discussed instruction pipelining and super scalar processor design. So, in order to improve the overall performance of the system the processor micro architecture has advanced from in order scalar pipeline design to the out of order super scalar processor design. And also we have gone deeper pipelines in our processor design. Once we have deeper pipelines, so, we can have more number of in flight instructions in different stages. So, as a result we can improve overall throughput of the system.

At the same time in the super scalar processor design we consider parallel pipelines, so, as a result we can actually issue multiple instructions in every cycle. So, once we consider this multi issue super scalar processor with deeper pipeline so our total number of in flight instructions will be significantly higher. So, that by using the out of order execution mechanisms supported by super scalar processor, we can identify independent instructions and execute these independent instructions on various functional units available in the system.

So, because of deeper pipelining and multi issue we can improve the performance of the system significantly compared to an in-order the scalar pipeline. So, again in order to further increase the processor performance, we can go for even much more deeper pipelining and also we can go for aggressive super scalar mechanisms. So, when I say aggressive super scalar mechanisms, we can actually consider speculative execution, we can consider advanced branch prediction mechanisms and we can increase the issue width of super scalar processors and so on. But when we are going for deeper pipelining concept, as well as aggressive super scalar mechanisms in our systems, the complexity of the system is going to increase significantly. And in reality the amount of performance whatever we get because of aggressive super scalar mechanisms as well as deeper pipelines is not as that high as compared to moderate pipeline super scalar processor design. So, this is called as ILP wall.

(Refer Slide Time: 02:55)



So, the ILP wall says the returns whatever we get is not much higher compared to the amount of effort we put in. So, in this graph the x axis shows the effort we put and the y axis shows the performance we get in return. So, we move from scalar in-order processor to a moderate pipeline super scalar processor, which supports out of order execution and we increase the number of pipeline stages in our super scalar processor and also we add this aggressive super scalar techniques such as speculative execution and so on, but we can see here after this stage the amount of performance improvement compared to the previous design points is not significantly increasing.

In another words this performance curve is almost flattened here and as a result we may not get significant benefits if we go for complex super scalar processor design and so on. And this is mainly because of couple of factors associated with our applications. One thing is the applications may not have significant instructions level parallelism. So, that is a reason why it is called as ILP wall. Because if applications are not having enough instruction level parallelism, even if you use aggressive super scalar mechanisms we cannot fill all the issue slots with useful instructions to compute.

So, as a result once we do not have useful instructions to issue on to functional units. So, functional unit cycles will be wasted, as a result overall performance will not be improved significantly. so, that we can show here in this particular diagram. Here, this is 4-wide super scalar processor, which issues 4 instructions at every processors cycle.

So, we have 4 functional units and maximum ILP they can exploit with this 4 wide super scalar processor is 4. So, that means that we can issue 4 instructions every cycle to our functional units. And also we assume that all the functional units are properly pipelined. So, even when we have properly pipelined functional units and our system can support 4 instructions executing simultaneously every cycle. Even then if you see in this diagram there are several empty boxes and each of these empty boxes indicates that the functional unit in that particular cycle is sitting idle.

And this is mainly because the applications may not have enough instructional level parallelism. And also when we are executing some instructions, which may require long latency events, For example, if there is a load instruction that misses in the L2 cache or L3 cash, so, in order to service a load request which misses in the L2 cache, we have to go to L3 cache and which may take 10 to 20 cycles. So, as a result unless we supply the data from the L3 cache for this load instruction, we cannot execute any instruction which is actually requiring the data supplied by this load instruction.

So, as a result, we are not able to, as a result we may not be able to identify independent instructions that can be scheduled on this functional units in the appropriate processor cycle time. So, because of that, we will get lot of the cycles where functional units are not efficiently utilized. So, as we have more empty boxes, that indicates the more number of functional unit cycles are wasted and that in turn results in performance degradation. So, in summary applications may not have significant instructions level parallelism to utilize by using our aggressive, very deep super scalar processor.

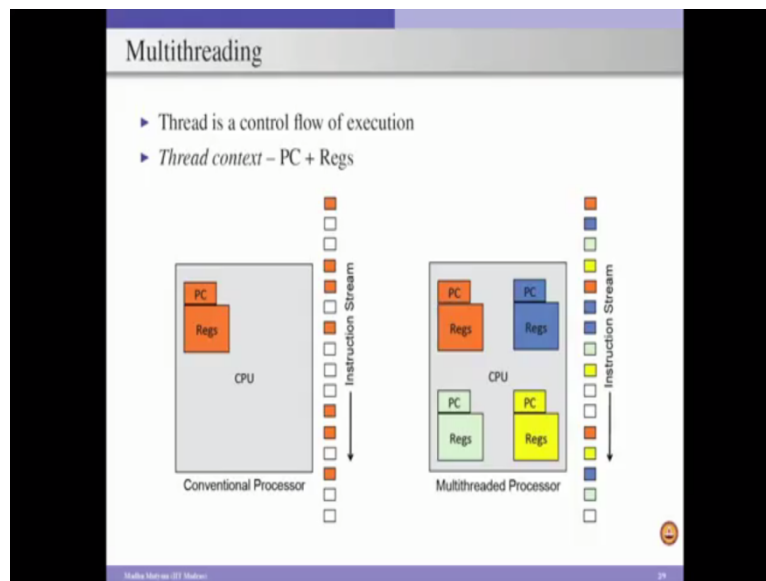
And also the super scalar processor design cannot tolerate these high latency events, because once the application is not having enough instruction level parallelism we cannot hide the latency incurred by these load requests, which miss in the L2 cache or L3 cache and so on. And also another thing is, this super scalar processor always work with a single thread. At any point of time it can fetch the instructions from single thread and these fetched instructions will be decoded and these decoded instructions will be dispatch to the functional units and they will be executed.

So, as a result if the single thread is not having enough instruction level parallelism, we cannot get significant performance improvement with the super scalar processor. So, our objective is to improve the performance of the system significantly by using some other

mechanism. So, in other words utilize these idle functional units which are sitting idle because of lack of ILP in our programs. And in order to utilize these functional units, what we have to do is we allow multiple threads to share these functional units, so that they can execute their instructions on these idle functional units.

As a result instruction execution will be overlapped across multiple threads and as a result resource utilization will be improved. And the overall throughput of system can be improved. So, in order to improve the overall performance of the system by making efficient use of our functional units, we have to go for the other method that is called as the “Multithreading”. And this multithreading is part of thread level parallelization. So, far we have discussed instruction level parallelization as part of super scalar processor. Now, we move on to a different architecture concept that is multi threading, where we actually exploit the thread level parallelization, but of course, the complete thread level parallelization can be exploited as a part of multi-core architecture that we are going to discuss in the next module.

(Refer Slide Time: 09:16)



So, in the multithreading systems we deal with threads. A thread is nothing but a control flow of execution. So, given a program a compiler can divide that program into multiple threads or programmer himself can write multi threaded program. And once we have this multiple threads and each of these threads is called as software threads. And program can consists of hundreds of software threads, but in order to execute these software threads on the underlying hardware, we need to have the support for this multithreading.

So, as a result we need to have a system that takes care of the execution of these threads on its resources. So, for that actually we have to consider the thread context. The context of each thread can be identified based on the program counter as well as the architectural registers. So, whenever there is a switch between one thread to another thread, we have to save all the contents of architecture registers as well as the program counter. So, that we can start executing the next thread and once the next thread completes its execution, then we can again come back to the previously switched out thread, by restoring these values.

The pc content and the architectural registers are back to these registers, so that we can resume the suspended thread execution. So, as a result our hardware needs to have support for executing multiple threads simultaneously. So, for that effectively we need to have multiple program counters, we need to have multiple architectural registers, (the files) and associated rename register logic and so on. So, effectively if a system is supporting multiple threading concept that means like if the system can execute multiple threads on it then it has to have support of multiple program counters as well as the multiple architectural register file. And once we have this pc and architectural register files replicated for each of the threads.

Then we can efficiently execute multiple threads on the system by switching in and switching out of the threads. So, in a conventional processor we typically have one program counter and one set of architectural registers. And this entire thing is called as the thread context. And this conventional processor is actually dealing with one thread at any point of time. And once we have a single thread context for this hardware, then at any point of time it can execute only one thread or instructions from a single thread.

And if you see the instructions stream, so, it always takes the instructions from single thread and execute. Because we are dealing with single thread and single thread may not have enough instruction level parallelization. So, as a result because of the dependencies among the instructions we may have the pipeline stalls or bubbles in the pipeline execution. So, as a result you can clearly see here this white boxes indicate that the functional unit is not utilized in that particular processor cycle.

So, there are many empty slots mainly because our thread is not having enough instruction level parallelization and also because we are executing instructions from single thread. So, as a result we are not able to utilize these underutilized functional units. So, in order to make use

of these underutilized functional units, what we have to do is we have to go for a system that is supporting multiple threading. So, when we consider a system supporting multithreading, now we can see here we have single processor, but this processor has now support for 4 hardware threads or thread contexts.

So, each hardware thread has associated pc and the architectural register file. Of course, we know that the processor is not just collection of the pc and the architectural register file, there are several other resources associated with CPU or the processor. Now, when we have multi-threaded processor all the other resources of the processor will be used by all these the software threads which are executing on these hardware threads.

So, effectively the other than this pc and register architectural register file all other component of the processor can be shared by the software threads, which are running on these hardware threads. We can also call the thread context as hardware thread. So, now in this particular design, we have a single processor which has support for 4 hardware threads. So that, when a programmer writes a multi-threaded application with 100 software threads and if he wants to execute this 100 threaded, multi-threaded program on this particular processor. So, he can execute only 4 threads at any point of time because this system is supporting only 4 hardware threads.

So, effectively, we have to map 4 software threads onto the 4 hardware threads and at any point of time it can execute only 4 software threads. Again depending on the type of multi threaded processor designs we can make only one hardware thread active at any point of time or we can make all hardware threads active simultaneously. So, that we are going to discuss in the next foil.

So, now once we have the multi-threaded processor which has multiple thread contexts then we can clearly see here the instructions stream is actually taking instructions from different software threads that are executed on different hardware threads. So, for example, in the first cycle we are taking an instruction from a software thread running on these the hardware thread or effectively by using this pc which is pointing to the one instruction in the software thread, that is associated with this.

And we fetched that instruction and execute that instruction. And in the next cycle we can move on to the second software thread which is mapped onto this hardware thread and by using this pc we will fetch the instruction from this software thread and execute that

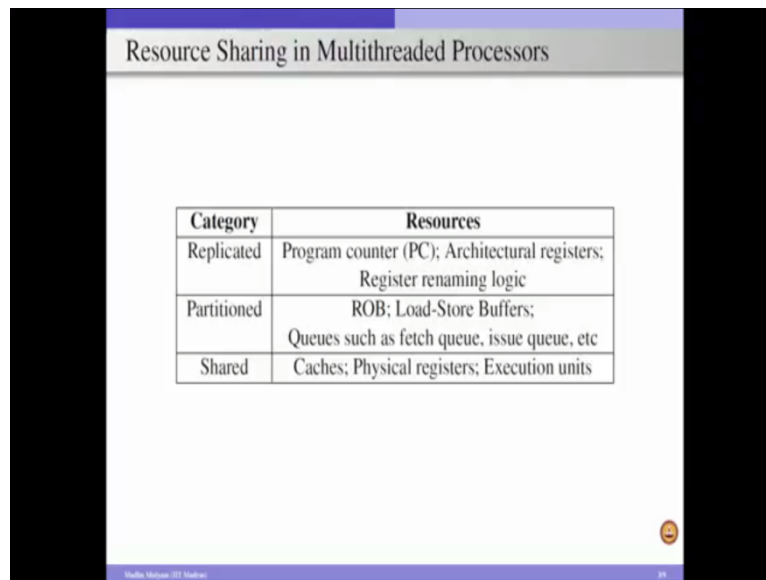
instruction. And in the third cycle we can fetch an instruction from a software thread mapped on to this hardware thread and so on. So, as a result we can minimize the wastage of functional units by exploiting thread level parallelization. So, it is like one thread is blocked for long latency or even there is L1 cache miss something which take even if it is taking or 2 to 3 cycles.

So, depended instructions cannot be issued to the functional units. So, as a result we can mask this cache misses and so on or the miss penalties by scheduling instructions from different threads. So, as a result by exploiting thread level parallelism we can make use of all the functional units efficiently and we can improve overall performance. Now we can clearly see here for example, this instruction which is from this software thread mapped on to this hardware context, if it incurs L1 miss which is going to take 6 to 7 cycles to get the data from L2.

Meanwhile while this instruction is sending its load request to L2 cache and L2 cache is going to supply the data meanwhile we can actually schedule instruction from other threads onto these functional units. So, effectively the latency incurred by this load miss for this software thread is overlapped with execution of threads from the other hardware context. So, as a result we can hide the latency because of this cache misses and that is going to utilize all the functional units efficiently and we can improve overall performances of the system.

So, as a result once we have the multithreading support in our system, we can exploit the thread level parallelization and we can improve overall performance of the system.

(Refer Slide Time: 18:20)



Category	Resources
Replicated	Program counter (PC); Architectural registers; Register renaming logic
Partitioned	ROB; Load-Store Buffers; Queues such as fetch queue, issue queue, etc
Shared	Caches; Physical registers; Execution units

So, once we have a multi threaded processor, now we can see like what resources will be shared, what resources need to be replicated and what resources can be partitioned in our processor. A processor consists of collection of resources and starting from the program counter, the architecture register file, register remaining logic, reorder buffer, load-store buffers, fetch queue, decode queue, dispatch queue, issue queue, the retire queue and similarly, cache memory - L1, L2, L3 caches, renamed registers or physical registers and functional units.

So, several resources are available with the processor and if the processor is supporting multithreading. Now, some of these resources can be shared by all the threads associated with processor, but some resources need to be replicated. Some resources cannot be shared by multiple threads such as program counter, architectural registers and rename register logic. Because we know that the control flow of a thread will be uniquely determined by the program counter content as well as the architectural register contents.

So, as a result we cannot share these resources across multiple threads. We have to replicate these resources if we want to support multiple threads in our processor. For example, if you consider 4-threaded multithreading processor then, we need to have 4 PCs, 4 architectural register files, we need to have 4 register rename logic units. And similarly, if you are considering 2 way multithreading then we need to replicate these units 2 times.

Whereas in the case of reorder buffer, load-store buffers and several other queues, we can partition these things across multiple threads. And this partitioning can be done statically or dynamically. We can equally divide our rob entries for all the threads of our system or we can divide the load store buffers equally across all the threads and so on. When I say thread, here it is hardware thread associated with our processor. And similarly, we can divide queues across all the threads of the processor.

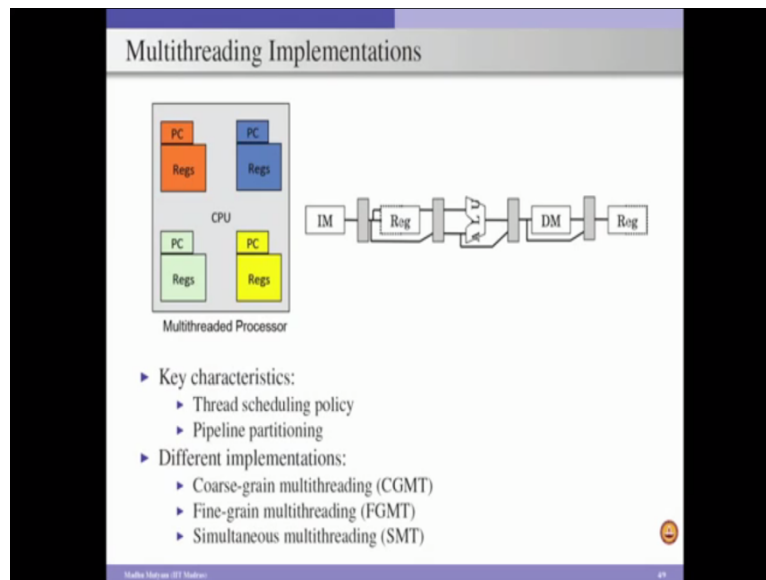
And in the case of cache memory, the physical registers, execution units and so on, we can actually share these units across multiple threads. The reason is because the multiple software threads running on your hardware threads of a multi threaded processor may be related in somehow, because all these threads belongs to same process and as a result there may be the data sharing across multiple threads of a process. So, as a result we need to consider this cache memory shared across multiple threads.

Similarly, we can consider these physical registers shared across multiple threads. And of course, the execution units we have to share, because we cannot have separate execution units for each of the threads. If you have separate execution units then the execution unit utilization may not be significantly improved. For example, if we consider one thread is actually executing integer operations and the other thread may be executing only floating point operations.

Now what is going to happen is, if we are considering a separate execution units for each of the threads and we can consider one instance of integer unit, one instance of floating point unit for each of the threads in our design, where we consider separate execution units across the multiple threads. If it is so then when I am executing a thread which is executing only integer operations then the floating point unit associated with that thread will be underutilized.

Similarly, if I am executing the instructions from the floating point thread then the integer units may not be utilized. So, as a result if we share all the functional units across all the threads then the resource utilization will be improved and as a result we can improve the overall performance of the system. And we do not incur significant hardware overhead because the functional units are shared across multiple threads. So, after discussing resource sharing across a multiple threads, now will see like what are the different types of multi-threaded processor designs we can consider.

(Refer Slide Time: 23:03)



So, our underline processor is consists of 4 hardware threads. For example and we have a pipeline design for processing our instructions. So, in order to consider different multi-threaded processor designs, so we have to consider these key characteristics. So, which thread to be selected for executing next instruction because we have 4 hardware threads and each hardware thread will have some software thread mapped onto it.

Now, whether we have to fetch the instruction from the software thread mapped onto this hardware thread or from mapped onto this hardware thread or a software thread mapped onto this hardware thread or the other one. So, how do we select whether we have to issue instructions only from software thread at any point of time continuously and then move onto the other one, or whether we have to apply a round robin policy across all these threads and then execute instructions one from one thread per cycle and so on, or otherwise we have to consider all these hardware threads to be active simultaneously and so on.

So, that means like our thread scheduling policy is going to determine our multi threaded processor design. If we consider at any point of time only one thread needs to be servicing the request, then we will come up with one type of multi threaded processor design or if we consider multiple hardware threads can be active simultaneously, then we can come up with different type of multi-threaded processor design and so on.

Similarly, once we have the instruction pipelining now whether this instruction pipeline can be partitioned across this multiple threads or whether we have to consider no partitioning

across multiple threads so, that also going to determine what type of design we have to come up with. So, once we have these key characteristics to decide on the different type of multi-threaded processor design, we can consider 3 different multi-threaded designs. One is coarse grain multithreading CGMT, the second one is fine grain multithreading FGMT and finally, the last one is simultaneous multithreading SMT. So, we are going to discuss these 3 in detail in coming foils.

(Refer Slide Time: 25:40)

The slide is titled "Coarse-Grain Multithreading". It contains a list of bullet points and a grid diagram. The grid diagram has "FU's" on the horizontal axis and "Processor cycles" on the vertical axis. The grid is 5 columns wide and 6 rows high. The top row has 3 orange cells, followed by 2 white cells. The second row has 1 orange cell, followed by 4 white cells. The third row has 2 blue cells, followed by 4 white cells. The fourth row has 1 blue cell, followed by 5 white cells. The fifth row has 3 green cells, followed by 3 white cells. The sixth row has 2 yellow cells, followed by 4 white cells.

- ▶ Thread scheduling policy:
 - ▶ Switch threads on long-latency events (L2 cache miss)
- ▶ Pipeline partitioning:
 - ▶ None, flush on switch
- ▶ Tolerate only long latencies
- ▶ Need 2-to-4 thread contexts for most benefits
- ▶ Ex: IBM Northstar/pulsar
- ▶ Simple, improved throughput, low cost
- ▶ Not suitable for out-of-order processing

So, we start with coarse grain multithreading. As the name suggest coarse grain. So, we always execute instructions from a single thread continuously. And this will continue till a point where any instruction from this thread is incurring a long latency event, such as L2 cache miss or L3 cache miss. Whenever such thing happens then, because in order to service this long latency event we incur significant amount of processor cycles and rather than wasting the processor cycles for such significant amount of time, what we can do is we can issue instructions from a different thread.

So, that means whenever there is long latency event occurred on currently running thread then we switch out that thread and then we resume the execution from a different thread. So, that means in this coarse grain multithreading our thread scheduling policy is like this. We switch threads only on long latency events. Why we are considering long latency events only because this is also determined by our pipeline partitioning method.

In this coarse grain multithreading we consider our pipeline will not be partitioned across multiple threads. So, that means whenever there is a context switch is going to happen for a currently running thread then, we have to flush all the instructions related to that particular thread starting from instruction which incurred this long latency event. And after that we can resume instructions from the different thread by fetching the instructions from the different thread. So, as a result there will be a significant amount of pipeline stages wasted in this process because our execution is happening somewhere in the middle of the pipeline and fetch happening at the front of the pipeline.

If we consider deeper pipeline design, then the number of pipeline stages between the fetch and execute will be significantly higher and as a result if we are always switching for a low latency events in this coarse grain multithreading design, then we are unnecessarily wasting processor cycles which leads to significant degradation in the performance. So, as a result we have to switch between threads only when we encounter long latency events.

So, as a result if this long latency event is incurring a significant amount of time to service then it is always better to go to a new thread even when we incur some number of pipeline stages are wasted in the process of flushing out and then fetching new instructions from the new thread. In other words in coarse grain multi threaded design we always consider context switch only for long latency events. And this long latency event is going to take significant amount of time which is much higher than the number of pipeline stalls that incur because of flushing out the pipeline and fetching new instructions from a different thread.

Because of this thread scheduling policy as well as the flush on a context switch, this coarse grain multi threading can tolerate only long latency events. It can tolerate only long latencies incurred by our instructions executed on the processor. And if an instruction which is incurring let us say 2 cycle latency which is going to create a stall in the pipeline by 1 cycle or 2 cycle, then this design cannot tolerate those latencies because for a low latency events we are actually not taking the context switch.

So, as a result we will waste those processor cycles in it. So, in order to reap in most benefits, so we have to consider 2 to 4 thread context in our coarse grain multithreading design. And so, in this particular example we are considering a 4 wide processor so as a result at every cycle we can issue at most 4 instructions onto 4 functional units. So, that means are

maximum ILP that we can achieve here is 4, but because of dependencies among the instruction software thread.

So, we may not issue 4 instructions every cycle and so on. So, as a result some of the processor cycles there are some functional units which are underutilized and also because we are actually issuing instructions from a single thread continuously and only when there is a long latency event then we move on to a different thread and we execute instructions from second thread and so on. Here the color coding indicates that we are actually executing instructions from 4 different threads. So, continuously we execute instructions from one thread, then move on to the other thread because of the long latency event incurred by one of the instructions executed earlier here.

After that we execute instructions from second thread and again there may be long latency event. So, we can move on to the different thread or otherwise we can start executing from previously context switched thread and so on. And this type of design is actually considered in IBM, north star, pulsar processor, but the current processors are actually not using this type of design. And it is simple to implement and it improves the performance in terms of the throughput. And it is easy to design.

So, as a result it is not going to take a significant cost because here we are considering only fewer number of thread contexts so as a result our hardware cost is also not significant. And because here actually we are taking a context switch on long latency events and we are stalling on low latency events. So, it is actually designed for in order processing. So, it is not suitable for out of order processing.

(Refer Slide Time: 32:11)

Fine-Grain Multithreading

- ▶ Thread scheduling policy:
 - ▶ Switch threads every cycle (round-robin)
- ▶ Pipeline partitioning:
 - ▶ Dynamic, no flush
- ▶ Tolerate pipeline latencies and cache access latencies
- ▶ Need enough # of threads to cover stalls
- ▶ Ex: UltraSparc T1 (Sun Niagara1)
- ▶ Conceptually simple, high throughput
- ▶ **Very poor single-thread performance**

	FU1	FU2	FU3	FU4
1	Orange	Blue	Green	Yellow
2	White	Orange	Blue	Green
3	Blue	White	Yellow	Orange
4	Green	Yellow	White	Blue
5	Yellow	Orange	Blue	White

So, now we move on to the second design, where we will consider fine grain multithreading. As a name suggests we actually switch between threads at every cycle in a round robin fashion. If you have let us say 4 threads in our processor, then in the first cycle we issue instructions from thread 1, the next cycle we will issue instructions from thread 2 then in the third cycle we will issue instructions from thread 3 and in the last cycle we are going to issue instructions from thread 4. And in the fifth cycle again we will the go back to thread 1 and will continue.

So, effectively we are actually rotating between all these available threads and by doing that we are actually going to hide the latencies incurred by our instructions. So in this fine grain multithreading, we switch between threads at every cycle and because at every cycle the thread context will be changing, so, as a result we have to consider a dynamic pipeline partitioning across multiple threads. So, effectively here one pipeline stage we may be executing an instructions from one thread, in the next pipeline stage we may be executing instruction from the other thread.

So, effectively in this fine grain multithreading concept, our overall pipelines will be shared by instructions from the different threads. One pipeline stage may be used by one thread, the other pipeline stage may be used by instruction from other thread, the next pipeline stage may be used by the instruction from the third thread and so on. Effectively, here, we are actually

sharing our pipeline stages across multiple threads. And we are also not considering any flush mechanism as that was considered in coarse grain mechanism.

So, as a result we are not going to incur any penalty also. And because we are switching between threads at every cycle, so, this fine grain multithreading can tolerate pipeline latencies as well as cache access latencies. When I say pipeline latencies, there may be an ALU instructions which is going to take 3 cycle to complete its execution. So, that also can be tolerated here. Similarly, there may be a load instruction which is going to take a 4 cycle latency to get the data from L2 cache. That also can be tolerated by scheduling instructions from the thread at appropriate time, but in order to hide this low latency as well as high latency because of the cache misses or instruction execution.

So, we need to have enough number of threads in our system. So, once we have enough number of threads we can switch across this large number of threads. So, that we can hide the latencies associated with this long latency events or short latency events. And the processor which actually implemented this type of fine grain multithreading is UltraSPARC T1 Sun's Niagara I processor. And the current NVIDIA GPU processors also actually use this fine grain multithreading. And this is conceptually simple to design and it provides a high throughput as long as we have enough number of threads to support the overall computation. But the main disadvantage with this fine grain multi threading is, the single thread performance is very poor.


The main reason is, even though, if I do not have any dependencies among the instructions of a single thread because of this threads switching across multiple threads and a system has multiple thread contexts. So, as a result I have to delay the execution of this particular thread, whatever I am interested in, because I will execute few instructions from this thread and wait for some time to execute instructions from other threads and only when my turn comes back and again I will execute instructions from my thread.

So, as a result single thread performance may not be improved and the overall performance of single thread will be significantly lower, but the overall throughput will be higher. So, we are trading single thread performance for improving the overall throughput of the system.

(Refer Slide Time: 36:48)

Simultaneous Multithreading

- ▶ FGMT + superscalar processing
- ▶ Thread scheduling policy:
 - ▶ Round-robin
- ▶ Pipeline partitioning:
 - ▶ Dynamic, no flush
- ▶ Tolerate pipeline latencies and cache access latencies
- ▶ Need 2-to-8 thread contexts for most benefits
- ▶ Ex: Intel P4
- ▶ Improved throughput, hide memory latency
- ▶ Resource partitioning:
 - ▶ static and dynamic
- ▶ Increased conflicts in shared resources



And finally, we consider a simultaneous multithreading. And this simultaneous multithreading is actually considering a fine grain multi threading as well as Superscalar processing mechanism. So, so far in the fine grain multithreading and coarse grain multi threading, the underlying hardware keeps only one hardware thread active at any point of time. So while we are issuing instruction from one thread, the other thread will not issue its instructions to the processor. So, as a result we cannot exploit the thread level parallelism to a full extent. So, but where as in the case of simultaneous multi threading we actually consider issuing instructions from multiple hardware thread simultaneously.

Also, the previous designs, fine grain multi threading and coarse grain multi threading are actually not considering out of order execution. So, here in this multithreading, we exploit the Superscalar processing, so, that is we can execute instructions in out of order fashion by considering our register renaming part, by considering dynamic scheduling and other things. As a result this simultaneous multi threading is actually built on top of Superscalar processors.

Because we are actually considering the fine grain multithreading concept, so, we consider thread scheduling policy as a round robin here. We can switch between multiple threads at a cycle granularity. And we can share our pipeline stages across multiple threads. So, that is what we are considering a dynamic pipeline partitioning mechanism. And we are not going to consider a fresh mechanism here. And because we are considering the round robin scheduling

as well as dynamic pipeline partitioning, so, this simultaneous multithreading can tolerate the pipeline latencies as well as cache access latencies, which may incur few cycles or which may incur 10 to 20 cycles.

In order to improve the overall performance we need to have 2 to 8 thread context in our system so that we can get maximum benefits and Intel Pentium 4 processor actually implemented this simultaneous multithreading under the name of “Hyper threading”. So, here we can see, so in cycle 1 we actually issued one instruction from first thread, one instruction from second thread, one instruction from third thread and one instruction from the 4th thread.

So, effectively in a single cycle we issued 4 instructions, one from each of the threads. So, that, we can execute these instructions on available functional units. So, as a result our functional unit utilization will be improved. In the second cycle again we can issue 4 instructions at max because this system is supporting 4 issue, but because of dependencies or because of not able to find independent instructions, here in this particular cycle we issued only 3 instructions from 3 different threads.

This thread is not able to issue a next instruction. Similarly, we are not able to find second instruction from any of these threads. And some times we can even issue all 4 instructions from this single thread also. So, effectively at each cycle we will see which thread can issue an instruction onto the functional unit. And if you can find multiple instructions from a single thread we can issue those multiple instructions onto the functional units and we can improve the overall performance.

So, as a result at every cycle we will look at all possible threads and see which thread can supply instructions. A thread can supply 1 instruction, 2 instructions or multiple instructions. And as long as we find some issue slot is free then we can go to next thread and see whether we can issue one instructions from that particular thread and so on. So, as a result we can minimize the resource wastage significantly and the overall performance can be improved by using this simultaneous multithreading.

So, this simultaneous multithreading can improve the throughput significantly and it hides the memory latency. Even if we are incurring an L3 cache miss we can hide the latency by servicing request from a different thread. So, as long as if a thread is waiting for some L3 cache miss to be serviced, we stop fetching from that particular thread. We stop fetching

instruction from that particular thread and we execute instruction from different threads. So, as a result we can improve the overall performance.

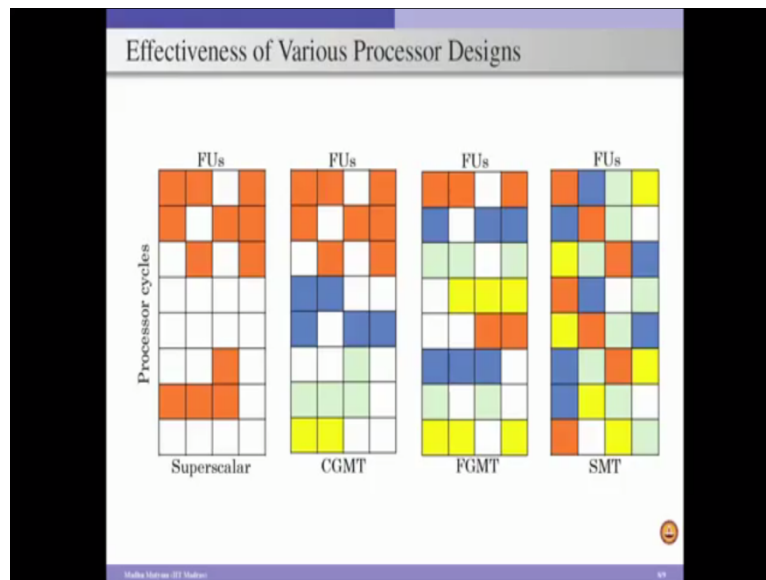
Here because in this simultaneous multi threading multiple hardware threads can be active simultaneously. So, we have to share our resources also. So, we have to have multiple fetch queue, we have to have multiple decode units, we have to have multiple the issue queues, we have to have multiple retire queues and we have to partition our rob across these multiple threads, so, that they can execute their instructions without having any difficulty.

Because resource sharing is happening in the simultaneous multithreading so, we can consider either the static partitioning method or we can consider dynamic partitioning method. And depending on the application requirements we can consider different proportion of our resources will be given to different threads. So, that is what can be considered in dynamic partitioning. And if you do not want this dynamic partitioning method we can go for static partitioning where we can divide our resources equally across multiple threads.

So, that everyone will get equal share and they can use these resources. So, again here in order to consider static or dynamic, we have to consider whether fairness is important for us or performance is important and depending on our requirements we can consider either the static partitioning or the dynamic partitioning. But because here multiple threads are active simultaneously and these threads can use the TLBs and the caches in a sharing mode and because of that there may be significant number of conflicts occur in the TLBs and the caches and that may degrade the performance.

So, as a result when we are dealing with simultaneous multithreading, we have to consider the efficient sharing mechanisms in these shared resources. So, that the performance degradation because of these conflicts can be minimized and we can improve the overall performance of the system.

(Refer Slide Time: 44:12)



So, in summary we can look at different processor designs and see how each of these designs are making use of the functional units in each processor cycle and improve the overall performance. So, if we consider super scalar processor because here instructions from a single thread will be issued on to the functional units and because of lack of instruction level parallelism we may not efficiently utilize all processor cycles and all the functional units in each cycle.

And in order to overcome this problem, we can go to different design that is called as coarse grain multithreading, where we can switch between multiple threads, whenever the currently executing thread encounters a long latency event we switch to another thread and issue instruction from the selected thread and continue execution. So, as a result we can minimize some of the resource wastage as compared to the super scalar processor, but again in the coarse grain multithreading we are switching between threads only on long latency events.

So, in order to further improve the overall performance we can actually go for fine grain multithreading, where switching happens at every cycle. So, we fetch instructions from one thread in cycle 1 and we switch to a second thread and issue an instruction from second thread and we will continue this process. But in a single processor cycle we fetch instructions and issue instructions from single thread. So, as a result still in that particular thread, if we are not able to identify enough ILP, then we waste functional units in that particular cycle and that is what we can see here.

There are still some empty boxes that indicate that some of the functional unit cycles are wasted if you use fine grain multithreading. So, in order to further improve the overall performance or in order to further improve the resource utilization, we can actually go for simultaneous multithreading where, in each cycle of multiple threads will be active and they issue instructions from the corresponding threads on to the functional units. So, as a result we can minimize the resource wastage, so that we can improve the overall performance.

So, as we move from Superscalar all the way up to simultaneous multithreading, we can improve the resource utilization which in turn improve s overall throughput and performance of the system. So, with that I am concluding this multithreading concept and in the next module we are going to look at multi-core architectures which exploit thread level parallelization at a bigger scale.

Thank you