Multi-core Computer Architecture - Storage and Interconnects

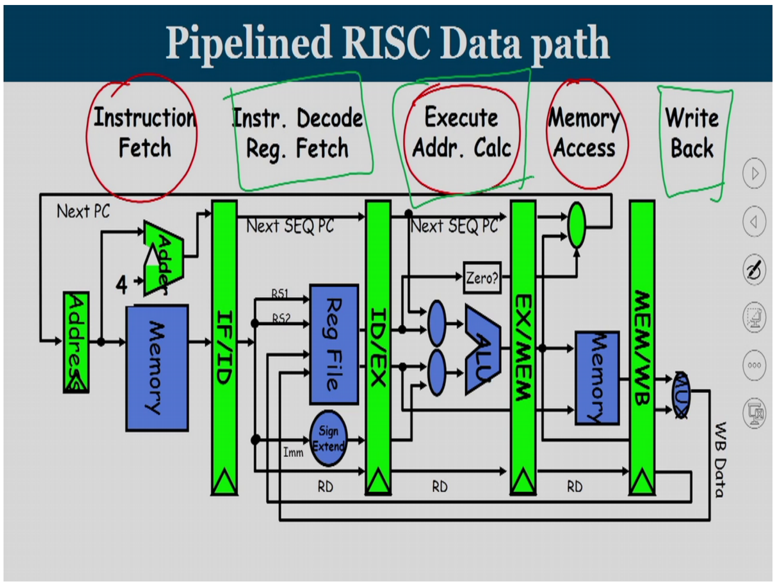
Dr. John Jose
Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati, Assum

Lecture - 03
Introduction to Superscalar Pipelines

Welcome to the third lecture of this course. In the last 2 lectures our focus was mainly on the design of an instruction pipeline. In the last lecture we have seen couple of hazards, which are circumstances in which normal execution of instructions will not happen. Or if you do execution of instruction in its assigned slot, then we will get incorrect result. We have seen structural hazard data hazards and control hazard. Today, we will see certain features of slightly advanced class of processors which are known as superscalar processors. We will see some of the feature of its pipeline, and what are the architectural support that is needed for a superscalar processor.

So, today's lecture title is Introduction to Superscalar Pipelines.

(Refer Slide Time: 01:48)



This is the conventional 5 stage architecture of the RISC instruction pipeline that we have seen; where, you have an instruction fetch stage followed by an instruction decode stage, and then we have the execute stage, memory access stage and the write back stage. The basic assumption that drives this 5 stage pipeline operation is, that we need only 1 clock cycle to complete each of these pipeline stage; whether it is instruction fetch or it is a decode operation, or let say it is an execute operation or mem operation or write back operation, we assume that the operation gets over in 1 clock cycle.

This is slightly unrealistic when you come to the real implementation of these units in hardware. We know that the first stage that is instruction fetch stage. And the 4th stage the memory access stage, both the memory access stage and the instruction fetch stage; these 2 are not completely inside the processor. The operations of instruction fetch and memory access are not completely inside the processor, because we have to interact with the memory. So, accessing the memory sometimes may not yield the result exactly at the same time. Certain instructions and data may be present in memory, where are some other instructions may not be present in the memory.

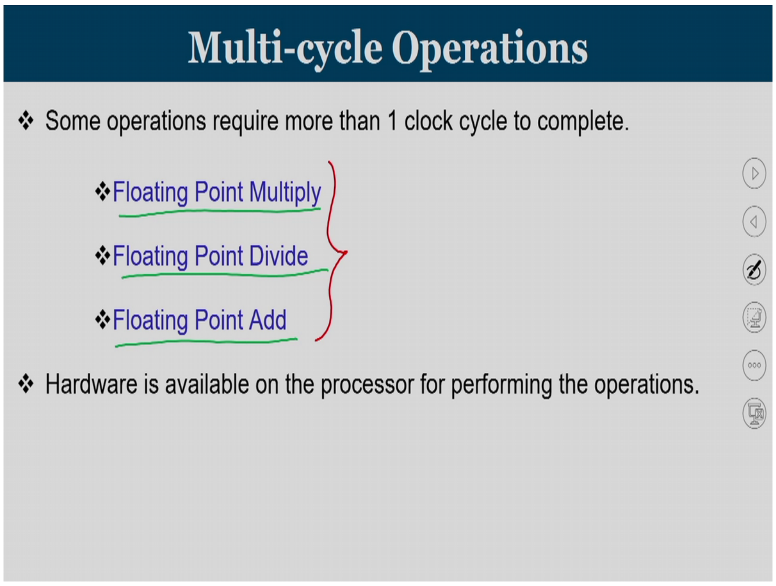
We will learn further about it when we discussed and explore about our cache memory, which is found one of the main component of this course. We will try to focus on slightly different aspect today. Let us focus on the execute side; decoding since it is completely inside processor. It can happen in 1 clock cycle. We have to provide necessary memory level support such that instruction fetch and memory access will happen in 1 clock cycle. See in the second stage, there is the instruction decode and the last stage that is write back.

Since it is completely associated with the internal registers of processor, it can be completed in 1 clock cycle. We require sophisticated memory support high speed memory support to complete instruction fetch and memory access operation.

Now, today we will focus on the execute portion of the instruction pipeline. Our assumption while working with this RISC pipeline was execute stage will take only 1 clock cycle but we know that there are different types of instructions. Some will work on integer operands, some will work on floating point operands, some will work on double precision floating point numbers.

And we all are familiar with the floating point operations. And when we do these operations in hardware with the help of a combinational circuit, in certain advanced data types apart from integers like in the case of float and double, execution of the operation that is working with this operands will not always be over in 1 clock cycle. We will see about a few classes of such kind of instructions where execution stage itself require more than 1 clock cycle. And let us see how this pipeline is going to work.

(Refer Slide Time: 05:44)



Multi-cycle Operations

❖ Some operations require more than 1 clock cycle to complete.

- ❖Floating Point Multiply
- ❖Floating Point Divide
- ❖Floating Point Add

❖ Hardware is available on the processor for performing the operations.

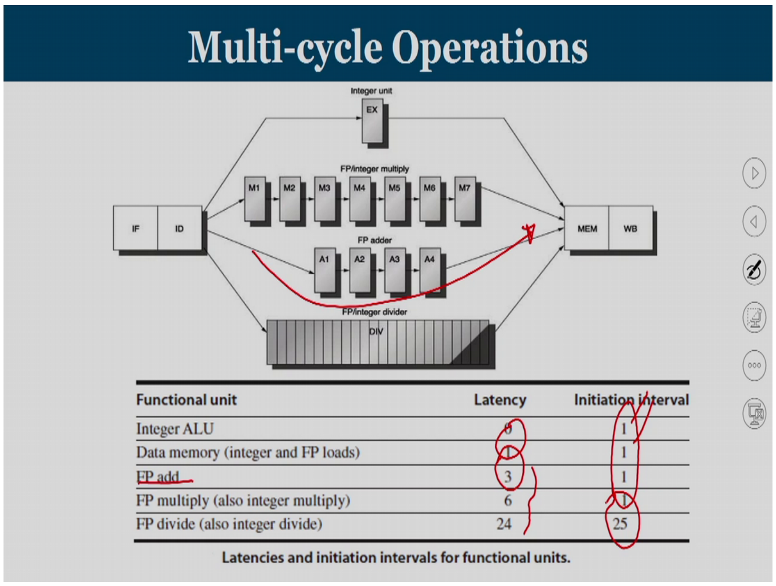
We will deal about multicycle operations.

Some operations require more than 1 clock cycle to complete; say, floating point multiply operations, floating point divide operations and sometimes floating point add or subtract operation. We all know that floating point numbers are being represented as a mantissa component and an exponent. In order to add 2 floating point numbers, first we have to perform some normalization, then we have to shift the mantissa accordingly such that the exponent portion remains same. Once the exponent is same, then only we can add the mantissa component. Let us say for example, consider the case that you are going to have a number, 2 into 10 to the power of minus 7, and other number is 3.6 into 10 to the power of minus 10.

While adding these numbers, you wanted to add these numbers it is not like adding the mantissa portion. We have to shift the exponents first such that make sure that both the numbers are having an exponent either 10 to the power of minus 7 or 10 to the power of minus 10. Once you make the exponent same then we can add the corresponding mantissa. This whole operation is typically involved in the case of a floating point arithmetic. So, to carry out such complex heavy operation, it may not be always possible in a single clock cycle. So, these operations will take more than 1 clock cycle, and we will see about this pipeline structure.

So, special hardware is available inside the processor for performing the specialized task for floating point multiplication, floating point division and floating point addition or subtraction.

(Refer Slide Time: 07:46)



Multi-cycle Operations

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Latencies and initiation intervals for functional units.

Consider this case, this is a classical architecture diagram of RISC pipeline with multi cycle operations. The instruction fetch, the instruction decode, the MEM and write back stages are not altered. Whatever pipeline we have seen in our last lecture was like this. IF, ID, EX, MEM and write back. This is what we call it as the integer pipeline; where the execution stage can be over in a single clock cycle. When you have instructions which consist of floating point operation, this single cycle integer unit may not be the right functional unit to carry out the task.

So, we have 3 more additional pipelines. One for integer multiplication, one for integer addition and subtraction or integer or floating point addition and subtraction and the last one is for division operation. So, once you fetch the instruction; while you performing the instruction decode operation, you will come to know whether this operation is on an integer operand or whether the operation is a multiplication operation or a division operation, or whether it is an addition or subtraction on floating point numbers.

Depending on the kind of operation, the appropriate EXE stage need to be called. So, upon decoding if you come to know it is an integer operation where you are going to add or subtract 2 integer registers, then it has to be carried out like this. The output of the ID stage, you have to put the values inside ID EX register. You have another pipeline register which is called ID M 1 register if it is a multiplication operation, rather than sending to EX it has to pass through this channel.

If it is a floating point adding or a subtraction operation, then it will carry out through this channel. And if it is a division operation it is going to carry out through this channels. Now, what are the peculiarity of these units? From the diagram it is clear that the multiplied unit consist of 7 subunits M1 to M7. And each are separately connected by arrows just like the other pipeline structure. Similarly, your integer your floating point adding unit is also 4 independent units. This means that, the multiplier execution stage itself is internally pipelined which consists of 7 substages; that means, at the end of 1 clock cycle the contents will move from M1 to M2 air for it moves from M2 to M3.

It moves from M4 to M5 like that it slowly moving. Similarly, in the case of an add operation at end of 1 clock cycle, some kind of a processing will happen in A1 then it moves to A2. So, A1 is free A1 can take a new set of data from ID. This means these 2 functional units the multiplier unit and the adder unit are internally pipelined; that means, every cycle it can set a new set of operands. They will partially process the result and pass it on to the next sub stage; that means, the multiplication operation after the decode stage the multiplication operation itself will take 7 clock cycles. M1, M2 etcetera up to M7.

So, if the instruction fetch happens at clock cycle 1 decode happens at 2, 3, 4, 5, 6, 7, 8, 9. At the end of the 9th clock cycle multiplication is over and 10 and 11. You will complete the remaining 2 stages of MEM and write back. So, essentially your multiplication operation will take 11 clock cycles. Whereas, all the instructions we have seen in our last lecture where basically integer instruction, they all will get over in 5 clock cycles. If you consider this case, then our adding operation will get over in 8 clock cycles. Like already mentioned your multiplication unit and the floating point addition unit both are internally pipelined; that means, every new cycle I can say a new set of data, but each of this has to move through all the stages one after another.

But there is one small difference in the case of the division unit. The division unit itself is an unpipelined unit; which carries total of 24 stages. This unpipelined unit means once you put a data into the division unit, for the next 24 cycles division unit cannot take any more operands, division is in progress. Or once you give the result it takes 24 cycles to produce the result of this. Division during this 24 cycles no more operands are permitted to enter into the division unit. Or we can say that, the division unit is having a structural hazard for the remaining 24 cycles.

To sum up, we have now 4 parallel layer units; one is called the integer unit which is called EX, then you have floating point or integer multiplier unit which consists of 7 stages in the EX, we have A4 stage floating point adder. And we have a 25 clock cycle unpipelined division unit. Few more details are given here that is pertaining to the latency and the initiation interval of these units. 2 parameters are mentioned, one is called a latency and the other one is called the initiation interval. We will look into what is the peculiarity of this.

For an integer ALU, if it going to produce the result, then another value which is going to use the result should not be delayed any cycle more. That is called a latency of 0. So, latency of 0 indicates that for an integer ALU if the very next instruction is also an integer ALU operation, I can use this value without incurring any delay. What is initiation interval? Initiation interval means if I produce the result somebody is going to use the result. Or there should be one cycle difference between 2 adjacent pair of operands.

That is going to work in this integer ALU. Now we will see 2 important parameters defined as latency and initiation interval. For every operation there is a latency that is associated. For an integer ALU operation that is carried out in the EX unit, the latency is defined as 0; that means, any other instruction who is going to use this result, that is produced by this integer unit it have to wait additional 0 cycles.

That means if you work in a pipelined instruction the first instruction is going into a integer EX unit. The second instruction wanted to use the result produced by the previous instruction. And the result is available in EX unit by operand forwarding we can directly get the result. There is no additional delay that is been involved for example, consider the case you have an ADD instruction R 1 R 2 and R 3. Now you have a subtraction instruction, that is to R 4 and I am going to work with R 1 and R 3.

Now in this case we have a common operand, first instruction is going to write into R 1 that happens inside the EX stage. Second instruction is going to read from R 1 that is also going to happen inside the EX stage.

So, it is basically an EX unit to EX unit operand forwarding. And there is no delay or no extra stalled that you have to provide for the second instruction. Second instruction can run in it is assigned pipelined unit. Basically between these 2 instruction there is A1 cycle normal shift, that is happening in pipeline, no extra shifting is required. That is what is called latency of 0. Now we will see what is the next unit the next unit for discussion is the data memory. So, if you access the data from the data memory, and if you wanted to use the data in any other instruction, minimum there should be 1 cycle delay.

And that is what is called this latency. We have seen that, whenever you have a load operation immediately after the load operation if you have an ADD even though we use operand forwarding there is A1 cycle delay. This is what we have seen in our last lecture.

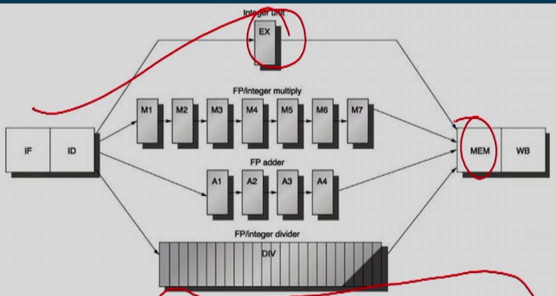
That is been defined as the latency of data memory. Similarly, the initiation interval concept means, what should be the minimum cycle delay such that the same functional unit can be used again. So, since it is been pipelined, the very next cycle that is the next cycle I can use the integer ALU. That was coming to data memory also. The very next cycle I can use the data memory. Now coming to floating point add we know that floating point add is this. So, if somebody is going to produce the result, the adjacent instruction need to have 3 cycled delay. Because since you have 4 stages the adjacent instruction will start anyway 1 cycle less. That is because of the normal pipelined shift. So, since I have to wait for 3 more cycles to get the result, then we define the latency was 3.

Similarly, you get 6 and 24 in the case of floating point multiplication and floating point division. You can see that since all the units are pipelined the initiation interval is only one. When you have an unpipelined unit the initiation interval is going to be 24. That is the normal latency plus 1. So, latency and initiation interval will help us to understand how long the subsequent instruction has to wait in order to use that functional unit. To repeat once again, latency of a functional unit means if a functional unit is going to produce a result, how much extra delay should be there for another instruction that is going to use this result. An initiation interval means; after how many cycles this functional unit can be used for operating on another set of operands.

Now, consider the case we are using an example.

(Refer Slide Time: 19:24)

Multi-cycle Operations



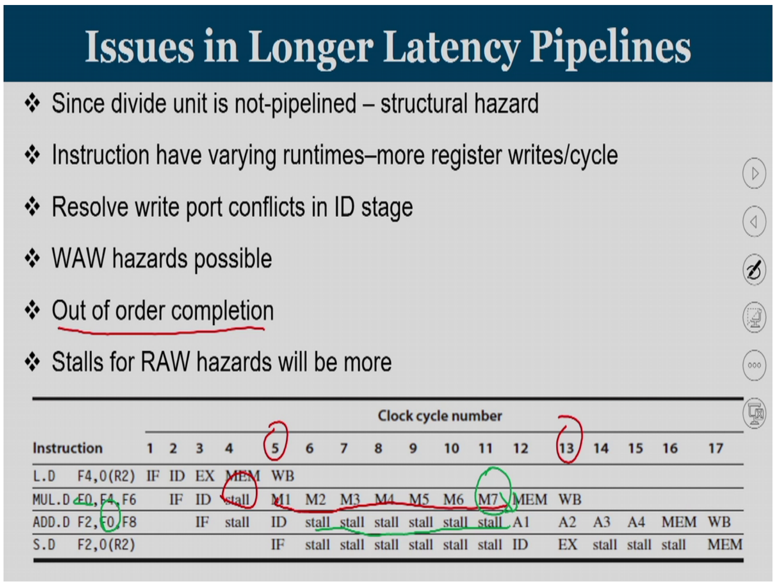
MUL.D	IF	ID	*M1*	M2	M3	M4	M5	M6	M7	**MEM**	WB
ADD.D		IF	ID	*A1*	A2	A3	A4	**MEM**	WB		
L.D			IF	ID	*EX*	**MEM**	WB				
S.D				IF	ID	*EX*	*MEM*	WB			

The pipeline timing of a set of independent FP operations. The stages in italics show where data are needed, while the stages in bold show where a result is available. The ".D" extension on the instruction mnemonic indicates double-precision (64-bit) floating-point operations. FP loads and stores use a 64-bit path to memory so that the pipelining timing is just like an integer load or store.

Going to introduce another new set of the instructions MUL dot D, it is a normal multiplication operation, but it is happening on double data type. That is called 64-bit operand. So, the first instruction is multiplication, like we can see that it has an IF ID and then 7 stages of memory. It goes to M1 to M7 then you have MEM and write back. It takes 11 cycles to complete this instruction. Consider the second instruction it is an ADD instruction. It starts one cycle less that is a normal pipeline structure, but add is having only 4 stages in the execution unit. And this instruction even though it started after the multiplication instructions because of the difference in the execution unit add is slightly getting before this multiplication.

Then we have load and store which works on the normal integer pipeline and they will take only 5 stages. This load and store are going through the integer pipeline, where EX unit is used for effective address computation, and memory access takes place in this MEM stage. This shows that, when you work with multi cycle execution instructions, then even though we start instructions in order, there is no guarantee that the instruction execution will be completed also in order. Now what are the issues when you have long latency pipeline like this?

(Refer Slide Time: 20:53)



Issues in Longer Latency Pipelines

- Since divide unit is not-pipelined – structural hazard
- Instruction have varying runtimes–more register writes/cycle
- Resolve write port conflicts in ID stage
- WAW hazards possible
- Out of order completion
- Stalls for RAW hazards will be more

		Clock cycle number																
Instruction		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L.D	F4,0(R2)	IF	ID	EX	MEM	WB												
MUL.D	F0,F4,F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D	F2,F0,F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	WB
S.D	F2,0(R2)					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

The first issue is a structural hazard. Since your floating point division unit is not pipelined; that means, for next 24 cycles, no other operation can use your functional unit, your division functional unit.

That means the division functional unit is not available for any other operands for the next 24 cycles, this lead to a structural hazard. Already my functional unit is busy I cannot use this functional unit for any other instruction. That is that will create a structural hazard. So, once you start a division for the next 24 cycles no more division permitted. So, if you write a sequence of code, if there are 2 adjacent division instruction, the second one has to way even though there is no dependency of any data then second one has to wait long.

And example can be let say division of A B C and the other one is division of D E F. The first one is operating on operands B and C. Second one is operating on operands E and F. There is no data dependency between the second instruction and the first instruction. But if you are using our unpipelined division unit, let say, if this division starts at clock cycle 1, then this division can start only at clock cycle 25. This is because from clock cycle 1 to clock cycle 24, the second instruction cannot use the same resource which is already been in use by the first instruction. And that is what is called a structural hazard.

Next point is instructions are wearing run times. We have seen in the previous case, when you have a multiplication instruction it has 7 stages in the execution part. Whereas, an add operations as only 4 stages in its execution part. So, when you have a multiplication instruction that is issued in clock cycle 1, I am having a multiplication instruction and in clock cycle 2 I am having an ADD instruction.

Then this multiplication instruction will take longer time. And ADD instruction will complete before the multiplication is over. So, since the instructions are having wearing runtimes there can be more writes per cycle after. Sometime this get over by 11th clock cycle. Now you can have some other instruction which will start at some point, it can also get over at the 11th clock cycle.

They may start the multiplication instruction. And let us say this can be a subtraction instruction. The subtraction may start a little late, it may take only 5 clock cycles, but both the multiplication and subtraction can complete in the saying clock cycle. That will create another issue, how many a writes I can perform, both will reach the write backstage exactly at the same clock cycle.

So, how many writes I can do that is another issue. We need to how more write ports, this course will not go into deeper regarding these issues, I am just giving you an overview of what are the problems associated with multi cycle pipelines. So, if such a kind of a problem what we previously discussed, you have a scenario where multiple instructions reaches the write backstage, then how can I resolve?

Then at the decode stage itself I have to understand that there is first instruction I, that will complete it is write back stage maybe at clock cycle n. After sometime I got one more instruction that is also going to complete it clock cycle n that should not be allowed. So, the second one I have to properly adjust in such a way that, no 2 instructions reaches the write backstage early.

This kind of an adjustment should be done as early as in the decode stage itself. So, I have to add up some extra stalls, such that no 2 instruction will reach the write backstage at the same time. Now third one is WAW hazard. What you mean by a WAW hazard? WAW hazard is a scenario where the second instruction is going to write into a register before the first instruction writes on it. So, consider the case, you have a multiplication instruction, that is going to write into R 1, and then you have an ADD instruction that is going to write into R 1.

Since the multiplication instruction even though it started before the ADD instruction, it may be completing only in the 11th clock cycle. Whereas, your ADD instruction will get over in the 9th. So, this gets over in the 8h clock cycle. So, I am actually waiting for a scenario in which my multiplication instruction is completing only at clock cycle 11.

Whereas, my ADD is getting over a clock cycle 8 this is exactly what you mean by a WAW hazard. Now we will see the next question is out of order completion. Out of order completion means, even though I issue instructions in order, since they are taking variable latencies, then certain instructions will complete out of order.

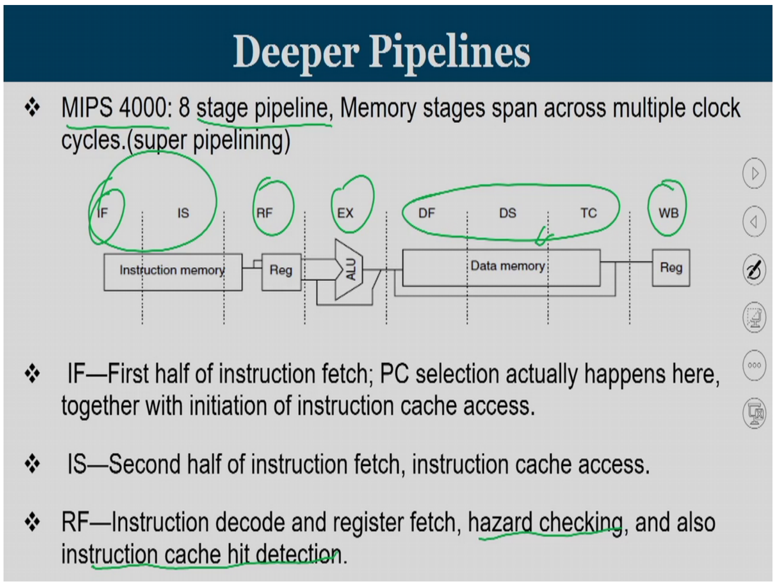
This out of order completion also will create us few issues; such that we have to manage the memory and so, register consistency. So, we fetch instructions in order we issue instructions or decode instructions in order, we execute instructions out of order and that result to out of order completion. So, consider this case we have a load instruction, it is actually on a 64-bit data. So, the effective address is computed on integer registers 0 and R 2 and the loaded value is loading into a floating point register; where F stands for a floating point register.

So, we are loading a value into F4, and then you are using that value in order to find out a new value F0 so, F0 is F4 into F 6. We can see that the first is a load instruction load instruction takes only 5 cycles as it uses the integer pipeline, and then we have a multiplication instruction. Since it is going to use a data after load, I have a stall even after operand forwarding I cannot run it in the normal slots. And since it is multiplication it has 7 stages in the execution unit. So, the instruction gets over only at clock cycle 13.

Now, the peculiarity of this is the third ADD instruction is going to use the result of the multiplication. So, you are going to produce a result in F0. Now this F0 value which is produced by the multiplication unit is going to be used by the adding unit. So, the adder has to wait until the result of multiplication is over the result of multiplication available only at M7. So, using operand forwarding you can forward from M7 to A1. Till that much time you are going to encounter stalls. So, we are going to have more number of stalls in this scenario. This problem is not there if you work on normal 5 stage pipeline.

So, once you are going to approach, the real instructions it is a taking little bit longer time, lot of design issues are going to come.

(Refer Slide Time: 28:29)



Deeper Pipelines

- MIPS 4000: 8 stage pipeline, Memory stages span across multiple clock cycles.(super pipelining)

IF
IS
RF
EX
DF
DS
TC
WB
Instruction memory
Reg
ALU
Data memory
Reg

- IF—First half of instruction fetch; PC selection actually happens here, together with initiation of instruction cache access.
- IS—Second half of instruction fetch, instruction cache access.
- RF—Instruction decode and register fetch, hazard checking, and also instruction cache hit detection.

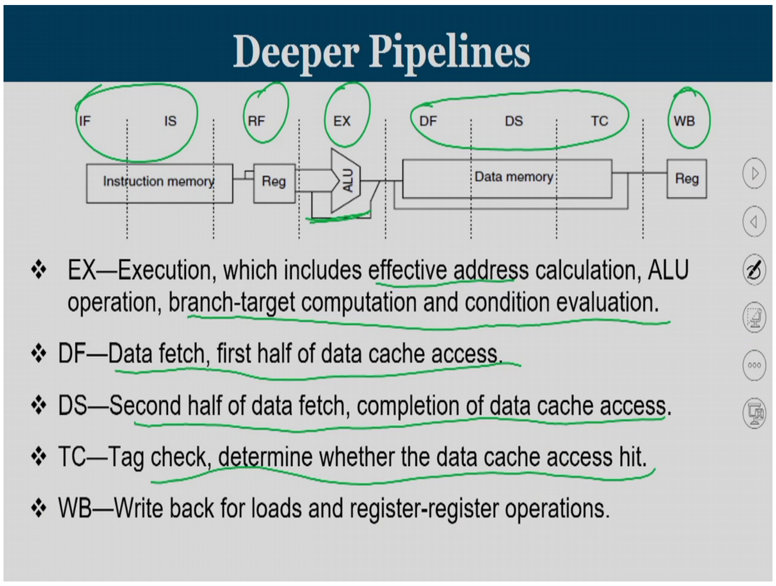
Now we will take one classical example of a deeper pipeline. Here we are going to work with MIPS 4000 pipeline it is having 8 different stages. So, these are the 8 stages you have 2 stages to fetch the instructions. Then you are going to have the decode stage which is also known as reading from the register. The EX stage then you have 3 stages for accessing the memory, and one stage for the write back.

Here also we assume that all the instructions are getting over in execution in 1 clock cycle. Now let us see what are the various sub stages? The IF stages is a first half of instruction fetch PC selection happens here. Together with you just start with the operation execution. And second one is the second stage of the instruction fetch so, the first stage will get the appropriate value of program counter, you start the operation of cache axis the instruction cache axis and only in the second stage. So, you how to go to memory and then perform the corresponding word selection and then the word is being transferred back to the processor.

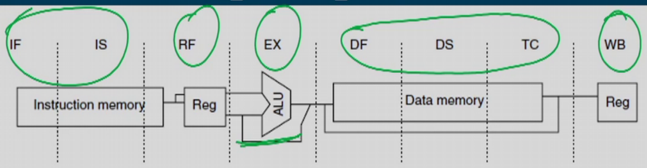
So, it takes 2 clock cycle so complete the instruction fetch operation. And then you have the register fetching the instruction decode and register fetch are happening in this stage, any hazard checking whether it is a ROW hazard or WAR hazard. And you take the value that is already a sheet in the instruction cache. So, to summarize the first 3 stages, it takes

2 cycles for fetching to get over it is take one cycle for instruction decode followed by reading from the registers.

(Refer Slide Time: 30:10)



Deeper Pipelines



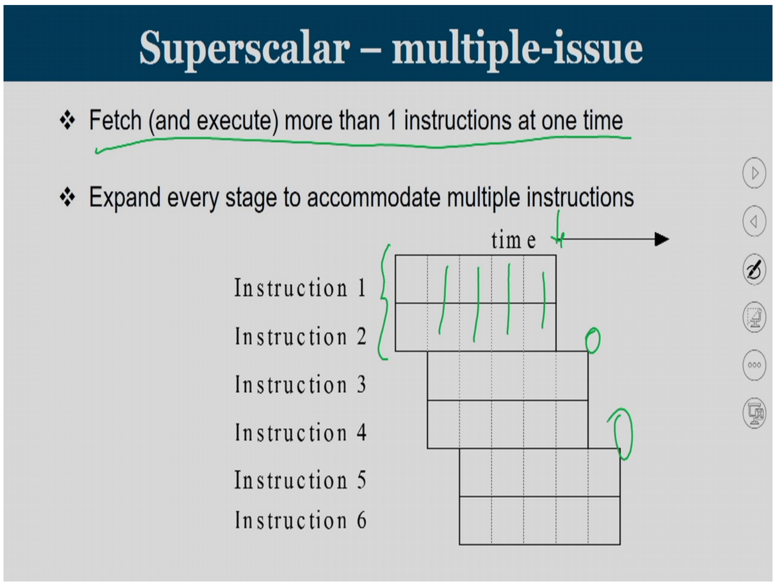
- EX—Execution, which includes effective address calculation, ALU operation, branch-target computation and condition evaluation.
- DF—Data fetch, first half of data cache access.
- DS—Second half of data fetch, completion of data cache access.
- TC—Tag check, determine whether the data cache access hit.
- WB—Write back for loads and register-register operations.

Coming into the subsequent stage the EX stage will take care of your normal execution, if it is an ALU operation, or if it is a load or a store operation it computes the effective address. If it is a branch instruction, then it computes the branch target computation and condition evaluation.

And then you have 3 stages to access the data memory, you have a data fetch that happens keep the MDR value ready, and then second half will take care of the second portion of data access, and then you check it out the tag comparison and all. We will learn more about a working of tax and all once we discuss about cache memory. So, together 3 stages are required to access the data memory and 2 stages are required to access the instruction memory, one stage for decoding, one stage for execution and one stage for write back.

Now, if this functional unit this pipeline if it has to be suitable for multi cycle operation, then this EX will be converted to M1 M2 up to M7 in the case of multiplication A1 to A4 in the case of floating point addition and 24 stages of division, and the last stage is W back.

(Refer Slide Time: 31:27)



Superscalar – multiple-issue

- Fetch (and execute) more than 1 instructions at one time
- Expand every stage to accommodate multiple instructions

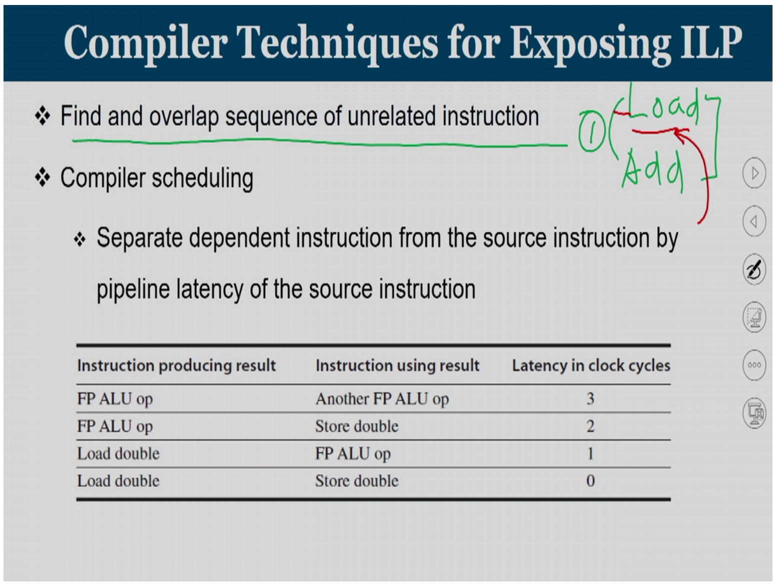
time
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction 6

Now, coming to what are superscalar processors. With the normal processors with the instruction pipeline so far we have learned; we are fetching only one instruction. So, at most one instruction is going to complete and the CPI what we are targeting Cycles Per Instruction is 1; that means, with a normal scalar pipeline. We are trying to achieve one instruction getting over in every clock cycle.

Can you improve more than this? And yes, it is possible. These category of processors are called superscalar processors, what they do is, fetch and execute more than 1 instruction at a time. So, you can see that you are fetching 2, you get decoding 2, your executing 2, you can perform memory access to 2 and you can complete writing. So, at the end of every clock cycle, 2 more instructions are getting over. This is a case where CPI is equal to 0.5 means we are taking one an average only 0.5 clock cycles for completion of an instruction.

Such category of processors which can complete more than 1 instruction unit time are called superscalar processors.

(Refer Slide Time: 32:46)



Compiler Techniques for Exposing ILP

- Find and overlap sequence of unrelated instruction
- Compiler scheduling
  - Separate dependent instruction from the source instruction by pipeline latency of the source instruction

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

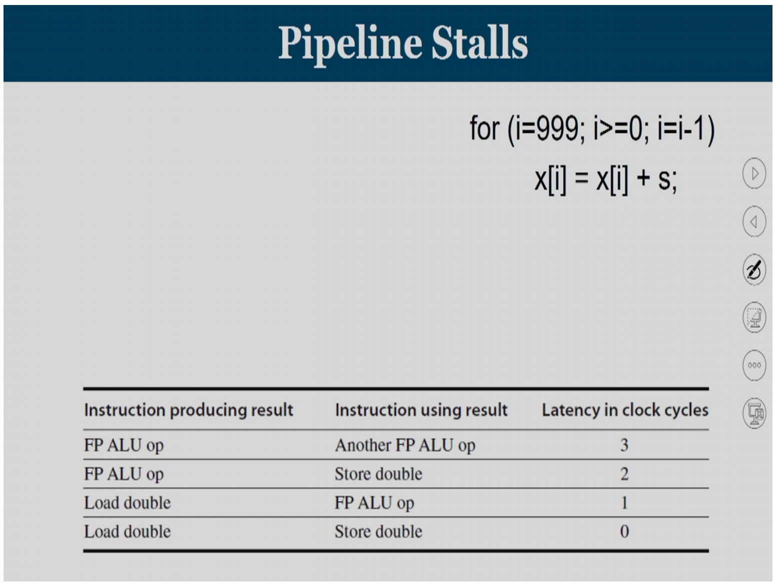
Now we are going to see certain kind of architectural improvements that is suggested, on superscalar processors, in order to increase its performance. So, the first one is called static scheduling, compiler is going to assess the hardware in improving the performance. So, what compiler does is find and overlap execution of unrelated instruction. We know that there are certain instructions which cannot run in the very next clock cycle as that of the previous instruction; like, when you have a load instruction, and then when you have an ALU instruction, that is going to make use of the result of the previous load instruction even though if you used for operand forwarding, we require minimum of one clock cycle gap between these 2 instructions.

Under this case there is a stall. So, if compiler could reorganize in such a way that, if I can fit in some other instruction between this load and this ADD instruction, then there is no stall that is going to happen. Such kind of reorganization done by the compiler to reduce the stalls in the hardware is known as compiler scheduling. So, compiler scheduling means you are going to separate dependent instruction from the source instruction by pipeline latency of this source instruction. In this case, when you have a load and then when you have an add which is comes immediately after this load, we know that there should be separated by minimum of one there is A1 cycle stall.

So, here what we have to do is, compiler has to separate this load and add by minimum of one instruction; that means, a new instruction has to be added between this load and

(Refer Time: 34:38) a new instruction which is somewhere down the instruction sequence, how to be found out and they has to put in between this load and that. So, the second instruction can run immediately after load as it is not dependent on the load. And since load and add are sufficiently separated there will not be any further stall. So, consider this case, where I am going to have a chart which will help us in working with the remaining examples.

(Refer Slide Time: 35:09)



Pipeline Stalls

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

So, consider this loop where you are going to store a scalar value into an element in the array. Which is basically a program where you read from an array which is called x. Take the value add a scalar value a constant value in to it, store it back. Let us say take a number add 10 to it store it back. Go to the next number in the array add 10 to it store it back. Go to the next number, again do 10 to it and store such a kind of program is that we arranging the value of i from 999 all the way up to 0.

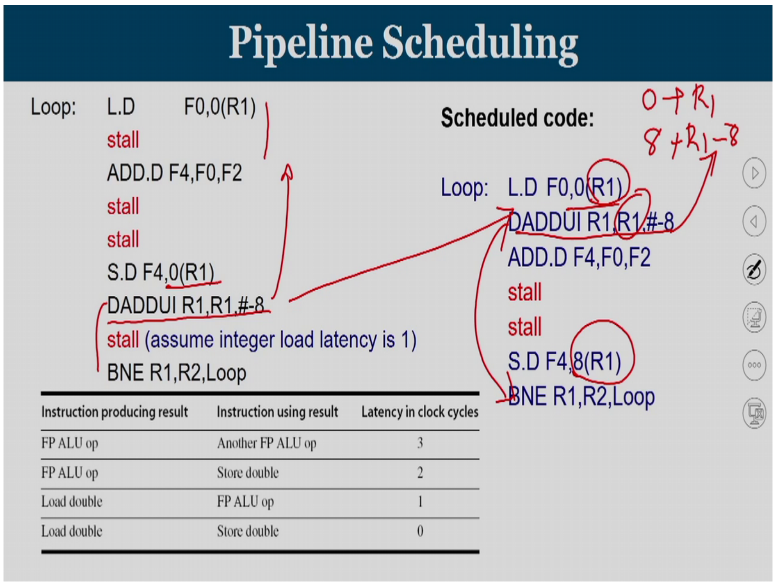
Let us say how the corresponding MIPS code will be. You are loading the first value into F0. We will assume that the data is of double precision type. So, you are loading the value into F0 and 0 plus R 1 will give you the effective address. Now once the value is loaded I have to add. So, the value is F0, and I have to ADD an F2, this F2 corresponds to your s. The value of s is stored in F2 I am going to add that constant value s into the previously loaded value I get the result in F4.

Now, once I perform an add operation it is a floating point add operation, you can see all the operands are floating point registers F4, F0 and F2. There should be minimum of 3 cycle stall if you wanted to use that value that is what you can see that. If a floating point operation is going to produce a result. And if you want if the store operation wanted to use the result there will be minimum of 2 stalls. That is what you can see this 2 stalls; that means, only after 2 stalls, the store instruction can be issued. And then whatever is the result in F4, you are going to store back into the same memory address what we have previously computed.

So, with this the operation of one element is over. I will summarize once again. Take the first number, add a constant to it you get the result in F4 store the result back into the same address. So, with that one element in the array is over now I have to go to the next element. You can see that this array is counting backwards. And each element will take 8 byte, because it is going to be stored in a double precision floating point number. So, I am going to subtract whatever is the ALU of R 1; R 1 is equal to R 1 minus 8. Once I get it, then I have to check whether the value of R 1 and R 2 is same this is the loop existing condition.

So, if it not, now I have a new value of R 1 go one load that number perform the scaler addition store it. Decrement the index to the next number, check whether I have reached the exit state, and you are going to repeat this so, this is the code that you have. Now what is the peculiarity of this code for each iteration I have 4 stalls. Now can compiler help in rearranging this such that I can reduce the stall? That is what you are going to see.

(Refer Slide Time: 38:18)



Pipeline Scheduling

```
Loop:   L.D      F0,0(R1)
        stall
        ADD.D F4,F0,F2
        stall
        stall
        S.D F4,0(R1)
        DADDUI R1,R1,#-8
        stall (assume integer load latency is 1)
        BNE R1,R2,Loop
```

Scheduled code:

```
Loop:  L.D  F0,0(R1)
       DADDUI R1,R1,#-8
       ADD.D F4,F0,F2
       stall
       stall
       S.D F4,8(R1)
       BNE R1,R2,Loop
```

0 + R1

8 + R1 - 8

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

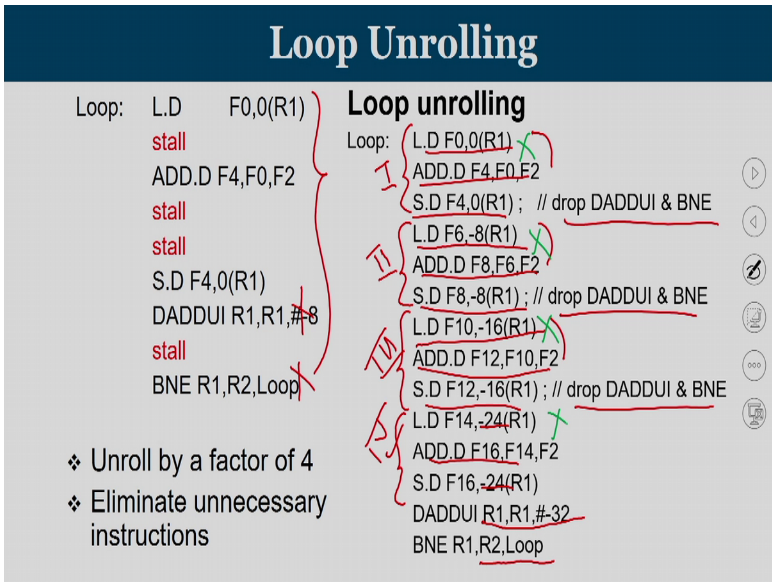
So, this is the program that we have just discussed. Now in what way compiler is going to help. You can see that, there is 2 cycle delay between your add and your store and between this load and add there is one stall.

So, once you load a value, you have to insert one stall between any instruction that is going to use the value, because of the dependency on F0. Now having said this how can I go into get the benefit? What I will do is whatever was my this instruction that is R 1 is equal to R 1 minus 8, that I will keep it in between your load and add. So now, load and add is separated by one instruction already. And I am going to bring an instruction which is not going to create any point; it is not dependent on load because in the load instruction is going to read from R 1. This adding instruction is also reading from R 1, and both the reads happened one after another so, it is not a problem.

So, why this rearrangements? So now, the value of R 1 is updated. Once you value of R 1 is updated, you have to see that rather than 0 of R 1, if you wanted to store into the same location you how to do with 8 of R 1. Why? Previously, it is 0 plus content of R 1. That is what is creating a load. Once it come to store if you wanted to store in the same location anyway my R 1 is reduced by 8. So, this instruction will so, reduce R 1 by 8. So, if you wanted to get the same result, then I have to pay it as 8 of R 1.

So, since this R 1 and the branch instruction as sufficiently separated previously, there was 1 minimum stall. Now this instruction is going up there is sufficient delay so, all together the 4 stalls that we had is been reduced to 2 stalls. So, when compiler is going to reorganize instruction without changing the meaning, then from 4 stalls we are able to bring down the number of stalls to 2. Can compiler do more? In this context within the single iteration of 4 loop compiler cannot do more. That is the case where we are going to work with a concept called loop unrolling.

(Refer Slide Time: 40:47)



Loop Unrolling

```
Loop:  L.D      F0,0(R1)
       stall
       ADD.D F4,F0,F2
       stall
       stall
       S.D F4,0(R1)
       DADDUI R1,R1,#-8
       stall
       BNE R1,R2,Loop
```

- Unroll by a factor of 4
- Eliminate unnecessary instructions

Loop unrolling

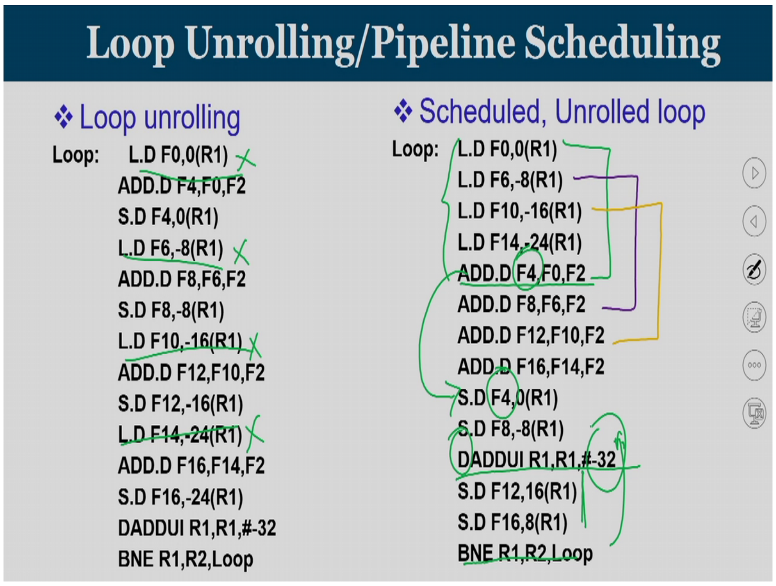
```
     Loop:  L.D F0,0(R1)
  I         ADD.D F4,F0,F2
            S.D F4,0(R1) ;   // drop DADDUI & BNE
            L.D F6,-8(R1)
  II        ADD.D F8,F6,F2
            S.D F8,-8(R1) ; // drop DADDUI & BNE
            L.D F10,-16(R1)
  III       ADD.D F12,F10,F2
            S.D F12,-16(R1) ; // drop DADDUI & BNE
  IV        L.D F14,-24(R1)
            ADD.D F16,F14,F2
            S.D F16,-24(R1)
            DADDUI R1,R1,#-32
            BNE R1,R2,Loop
```

Compiler unrolls the loop so, that it will get more flexibility to work on it. So, this is the normal instruction what we have seen. Now compiler is going to unroll this loop by a factor of 4. So, this load add and store, you remove these 2 instructions, you go and find out what is the next load in the next iteration. So, next load is load of minus 8 of R 1 then you have to add you have to store. Then you go to the third iteration minus 16, then you go to the 4th iteration.

So, in all this cases you have only load add and store this is iteration 1, this is load add and store that is iteration 2. Again this is load add and store this is iteration 3, and you have iteration 4. In each iteration even though I have 5 instructions, I could effectively drop the update and the branch comparison check. Because I am myself writing or adjusting my operands in such a way that, I am able to access the next iteration values. And then you in one short you reduce the value of R 1 is equal to R 1 minus 32 then you loop. This principle is called loop unrolling.

But we know that simply by unrolling, there is already a stalled at comes between this load and add. So, every pair of load and add is going to have stalls. So, wherever I am going to mark with this greening, there is going to be a stalled. Now can you get rid of this? This is the place where compiler scheduling is going to help.

(Refer Slide Time: 42:29)



Loop Unrolling/Pipeline Scheduling

❖ Loop unrolling

```
Loop:   L.D F0,0(R1)
        ADD.D F4,F0,F2
        S.D F4,0(R1)
        L.D F6,-8(R1)
        ADD.D F8,F6,F2
        S.D F8,-8(R1)
        L.D F10,-16(R1)
        ADD.D F12,F10,F2
        S.D F12,-16(R1)
        L.D F14,-24(R1)
        ADD.D F16,F14,F2
        S.D F16,-24(R1)
        DADDUI R1,R1,#-32
        BNE R1,R2,Loop
```

❖ Scheduled, Unrolled loop

```
Loop:   L.D F0,0(R1)
        L.D F6,-8(R1)
        L.D F10,-16(R1)
        L.D F14,-24(R1)
        ADD.D F4,F0,F2
        ADD.D F8,F6,F2
        ADD.D F12,F10,F2
        ADD.D F16,F14,F2
        S.D F4,0(R1)
        S.D F8,-8(R1)
        DADDUI R1,R1,#-32
        S.D F12,16(R1)
        S.D F16,8(R1)
        BNE R1,R2,Loop
```

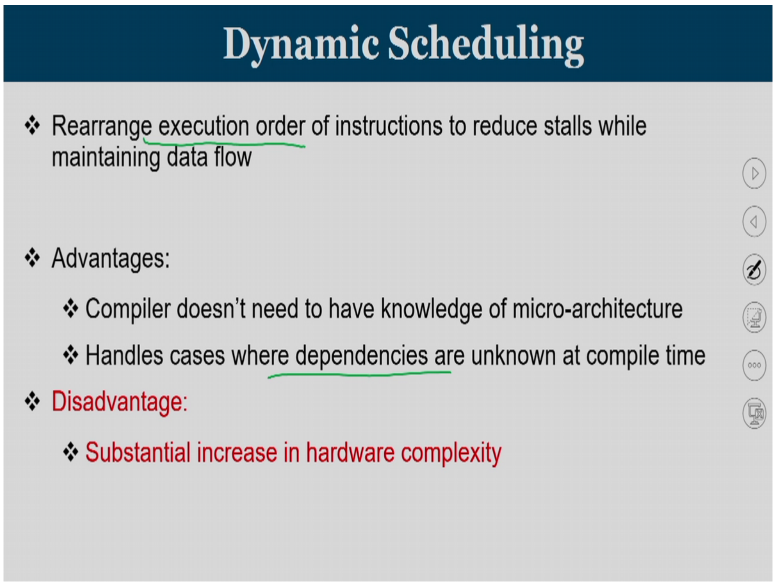
So, this is the unrolled loop. These are the places where we are going to have stalls. Now if you unrolled a code all the loads are been put together, all these loads are going to be put together, and the load and adds the corresponding adds this is the corresponding add when you work on this load and this add is 1. Now if I put this load and the second add is another 1, the third one is going to be dependent on the.

So, every corresponding dependent add and the corresponding load is been sufficiently separated. So now, all loads altogether, all adds altogether by the time the add is over. Now if you see this is an address which is going to produce the result in F4, and that add and the corresponding store that is going to make use of F4 are sufficiently separated. So, you do not have any stall in this case. In order to adjust is branch and the update value of R 1, this is slightly put above. So, all the stores coming after that, this 32 shifting has to be properly accommodated.

For those students who find it difficult to graphs the concept, I request you to go through the text books where there are lot of sold exercises that is being given. And feel free to contact us back if you have still doubts in this topic. So, we have seen a couple of techniques like rearranging the code, and then unrolling the loop rearranging an unrolled loop, all these are compiler techniques also known as static scheduling. The peculiarity of all these approaches the compiler has to know about the architecture, the number of stalls that is going to happen between ith and jth instruction has to find out using it intelligence some instruction and how to reorganize.

Lot of intelligence has to be fed into compilers. And moreover the architecture should be shared with the compiler. Then only so compiler and architecture will go hand in hand. Only in this context we are able to get performance. We will see yet another class of optimizations that is done on superscalar processors; where compiler is not involved let us compiler generate whatever code it wants. The architecture will take care of this dependencies and find out solutions; this is called dynamic scheduling.

(Refer Slide Time: 45:01)



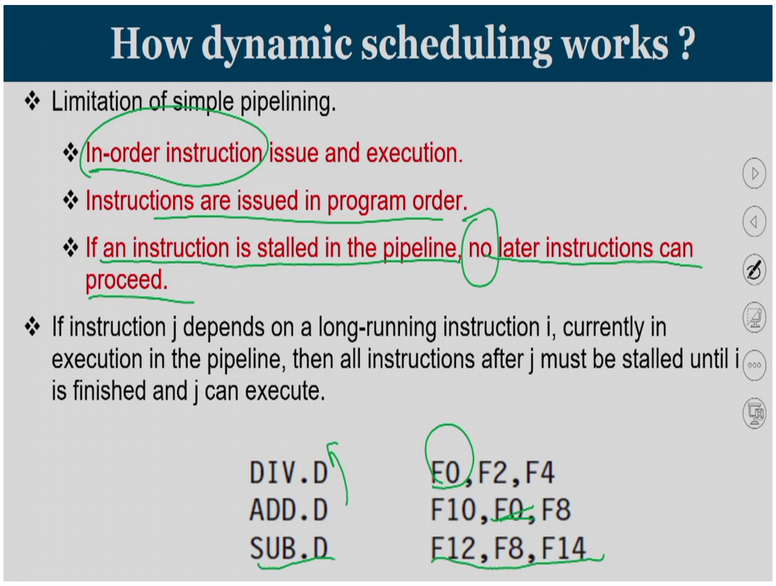
Dynamic Scheduling

- Rearrange execution order of instructions to reduce stalls while maintaining data flow
- Advantages:
  - Compiler doesn't need to have knowledge of micro-architecture
  - Handles cases where dependencies are unknown at compile time
- Disadvantage:
  - Substantial increase in hardware complexity

Dynamic scheduling is rearranging the execution order. And no longer rearranging the instruction, the instructions are in sequence, but rearranging the execution order of instructions to reduce stalls while maintaining the data flow.

The advantage is compiler need not have knowledge about the microarchitecture or architecture company should not reveal its internals to a compiler company. And it handle all the cases of dependency at the proper time. Now disadvantage is hardware is going to become complex. Your hardware need to have intelligence that has to detect dependency has to resolve it, has to take care of operand forwarding, has to take care of control hazards and has to take care of structural hazards.

(Refer Slide Time: 45:46)



How dynamic scheduling works ?

- Limitation of simple pipelining.
  - In-order instruction issue and execution.
  - Instructions are issued in program order.
  - If an instruction is stalled in the pipeline, no later instructions can proceed.
- If instruction j depends on a long-running instruction i, currently in execution in the pipeline, then all instructions after j must be stalled until i is finished and j can execute.

```
DIV.D     F0,F2,F4
ADD.D     F10,F0,F8
SUB.D     F12,F8,F14
```

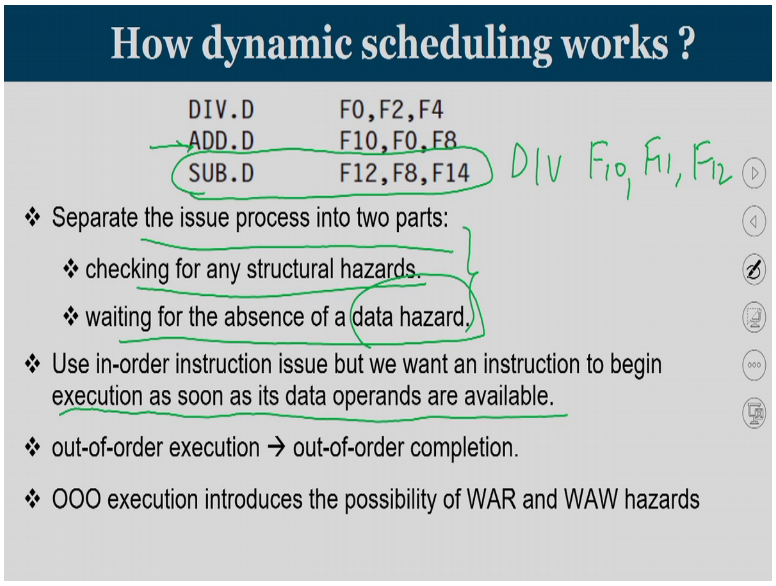
Now, how dynamic schedule in works? There are certain limitations of a simple pipeline When you have in order instructions and in order executions you fetch in order, decode in order, execute in order and complete in order that is your normal 5 stage pipeline. The problem with this an in order pipeline is instructions are always issued in program order if an instruction is stalled in the pipeline this very, very important. If an instruction is stalled in the pipeline no later instruction can proceed, no later instruction can proceed. So, if I have the tenth instruction that is going to have a stall because of a data dependency.

Then even though the 11th or 12th instruction is not having any dependency issues they are prevented this is the problem of in order pipeline. So, if instruction j depends on a long running instruction i, currently in execution in the pipeline, then all instructions after j must be stalled until i is finished then only j can execute. Consider the case of this course you have a division which is going to produce the result on F0. You have an ADD instruction that is going to make use of the value in F 0, and then you have a subtraction instruction which is independent of the division and the addition instruction.

In the case of an in order pipeline, that is what we have seen in order execution. The ADD instruction is dependent on division. So, the division takes 24 cycles. Until the division is over the ADD cannot proceed because it is a data dependency. No operand forwarding is possible in this case. So, ADD will be having 24 stalls because it is waiting for a data which is still in the pipeline. The division unit is producing the data it may take 24 more clock cycles.

But see the subtraction instruction, subtraction instruction is not dependent on add. So, why should the subtract weight, that is the problem of an instruction pipeline which works in order sequencing. So, this can be adjusted if you go for out of order execution.

(Refer Slide Time: 47:52)



How dynamic scheduling works ?

```
DIV.D      F0,F2,F4
ADD.D      F10,F0,F8
SUB.D      F12,F8,F14
```

DIV F10, F11, F12

- Separate the issue process into two parts:
  - checking for any structural hazards.
  - waiting for the absence of a data hazard.
- Use in-order instruction issue but we want an instruction to begin execution as soon as its data operands are available.
- out-of-order execution → out-of-order completion.
- OOO execution introduces the possibility of WAR and WAW hazards

So, consider this case how dynamic scheduling is going to work the problem is a subtraction instruction which is independent which is not having any data dependency, maybe subtractor is free. So, there is no structural hazard, there is no data dependency then why should the subtractor be waiting.

So, an adjustments modifications where suggested on in order pipeline to make it out of order pipeline. So, how will you make out of order to work? Separate the issue process into 2 parts. So, issue means your second stage of the pipeline; instruction decode which consists of you check whether there is any structural hazard. If this is a division operation rather than subtraction, if that would have been a division operation on F 10 F 11 and F 12. There is no data dependency, but because of it division, and division unit is already busy, it is a structural hazard.

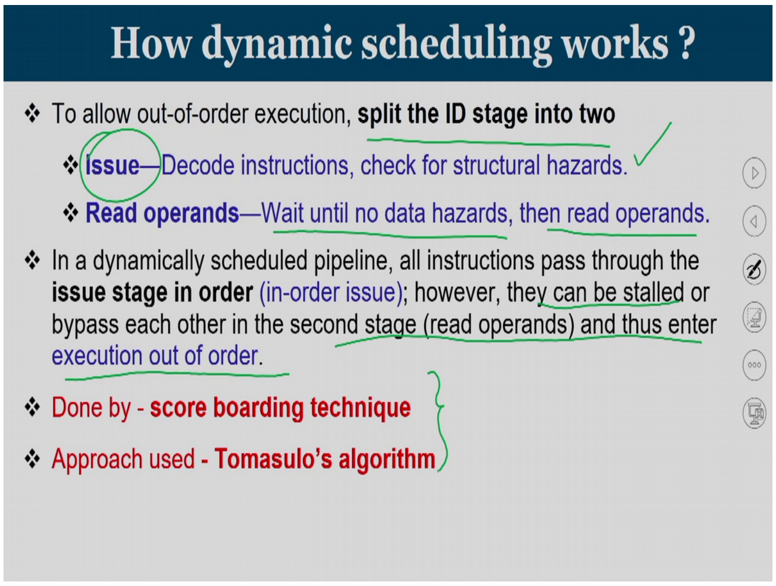
In this case it is a subtraction operation there is no structural hazard. My functional unit is free. So, check whether there is any structural hazard if there is no structural hazard, see are you waiting for any data there is no data hazard. As long as there is no structural hazard, as long as there is no data hazard, then we will not delay and instruction. We will allow the instruction to execute. So, my previous instruction add is still waiting.

My subsequent instruction subtraction is not waiting for anybody so; subtraction can start execution. So, even though I fetch them in order I decode them in order, or I call it as issue in order, but execution starts out of order. So, used in order instruction issue, but we want an instruction to begin execution as soon as data operands are available. And that extra facility that you provide is called out of order execution.

Out of order execution means your instruction can complete. So, when subtraction start execution before addition, subtraction will always complete before additions. So, out of order execution means out of order completion OOO in architecture community means, Out of Order Execution it introduces the possibility of WAR and WAW hazards. We have seen 3 different types of data hazards one is the row hazard which is the direct data dependency, and then we had WAR hazard and WAW hazard, at the time of discussing these hazards, we mentioned that such kind of hazard will not happen it in an in order 5 stage pipeline.

Now, we have seen larger pipelines which takes more than 5 cycles in the case of floating point division and multiplication. And we are now seeing out of order pipelines. This can lead to WAR and WAW hazard, but dynamic scheduling architectures will take care of it.

(Refer Slide Time: 50:39)



How dynamic scheduling works ?

- To allow out-of-order execution, **split the ID stage into two**
  - **Issue**—Decode instructions, check for structural hazards.
  - **Read operands**—Wait until no data hazards, then read operands.
- In a dynamically scheduled pipeline, all instructions pass through the **issue stage in order** (in-order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order.
- Done by - **score boarding technique**
- Approach used - **Tomasulo's algorithm**

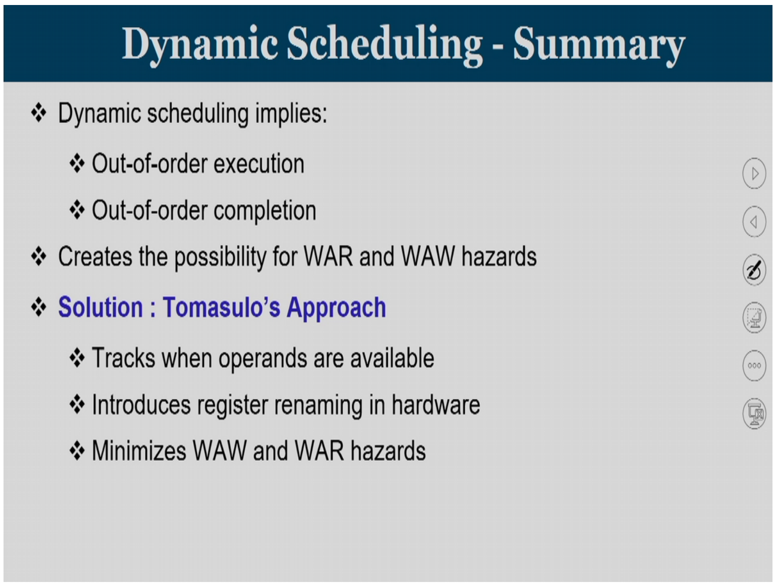
So, how dynamic scheduling works. To allow out of order execution we have to split the ID. The second stage into 2, one is called a issue stage, where you perform decode the instruction and check for structural hazards. So, if you have a structural hazard; that means, your functional unit is already busy, I cannot complete issue. So, completion of issue means decoding is over and there is no structural hazard on the functional unit where you are going to run the task.

The second one is you are going to read operands. So, wait until there is no data hazard then read the operands. So, how it is going to work in dynamically scheduled pipeline. All instructions passes through the issue stage in order so, if my previous instruction cannot be issued the next instruction is stalled. Once issue is over, then the other portion; however, they can be stalled or bypass each other in the second stage. Reading operands and thus enter the execution stage out of order. So, you can have stages where issue is over, but for a previous instruction it is waiting for a data it is having a data hazard.

But for the next instruction issue is over issue is over means decoding is over. And there is no structural hazard. Means, the functional unit where the operation is going to be carried out is available. So, once the decode is over once a functional unit is ready, and then there is no data dependency I can run. I can have a previous instruction where functional unit may be free, but it is waiting for data this permit a later instruction to execute before the beginning of execution of a previous instruction. So, this is done basically by score boarding technique. And those who wanted to know further about this, they can work on this approach this is called a Tomasulo's algorithm.

In our course, the main focus is on storage and interconnects. These 3 lectures are giving you a basic idea about how instruction pipeline works. And what are some of the design issues like hazards and instruction pipeline. And how can you improve the performance of instruction pipeline with the help of superscalar processors.

(Refer Slide Time: 53:04)



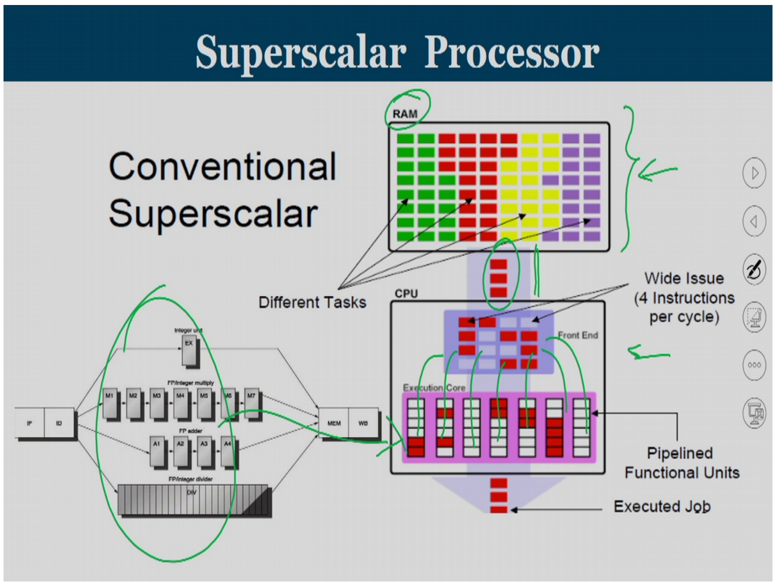
Dynamic Scheduling - Summary

- ❖ Dynamic scheduling implies:
  - ❖ Out-of-order execution
  - ❖ Out-of-order completion
- ❖ Creates the possibility for WAR and WAW hazards
- ❖ **Solution : Tomasulo's Approach**
  - ❖ Tracks when operands are available
  - ❖ Introduces register renaming in hardware
  - ❖ Minimizes WAW and WAR hazards

Before we conclude today's session, we will have a quick recap. Dynamic scheduling principle implies out of order execution, out of order completion. And this creates a possibility of WAR and WAW hazards; which the hardware has to take care of. And the solution is Tomasulo's approach. So, what Tomasulo's approach do is, it try to see when the operands are available. The moment the operands for an operation is available start executing on it. And it introduces the concept of register renaming to care of WAR and WAW hazards and it minimizes WAR and WAW hazards.

We are not going to learn deeper into a Tomasulo's algorithm. That is not coming under the preview of this course. But I request those students who are finding interest in this topic to read further. And the reference book is used this computer architecture by Hennessey and Patterson, which is already mentioned in my initial slides.

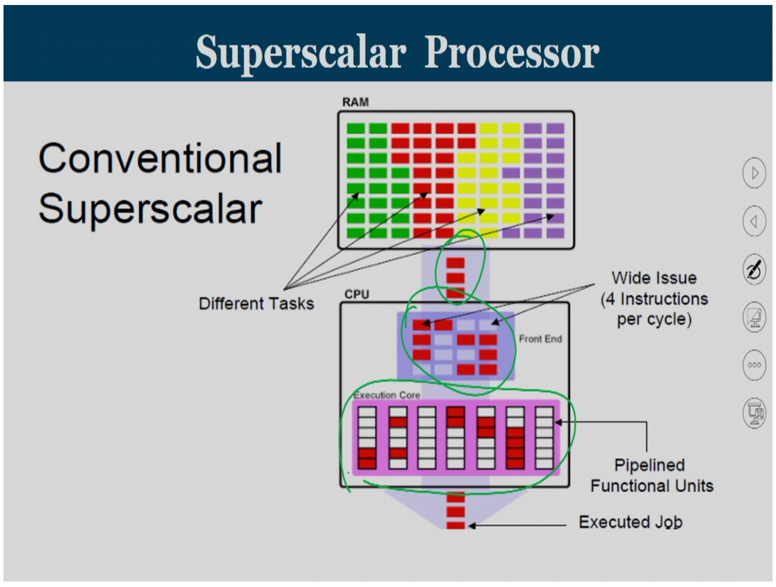
(Refer Slide Time: 54:01)



Now this is like conventional superscalar processor. This is a logical view of a superscalar processor. This different colour indicates different programs which is stored inside your main memory. So, you have different programs, you are going to fetch multiple instructions together into the processor. This is your memory and this is our processor.

So, multiple instructions are fetched from memory to processor. Their decoded together or issued together. And then you have functional units which will take care of this. So, these functional units can be compared, similar, you have different functional unit one for add, one for subtraction, there can be many adders, there can be many subtractors. So, whatever you have seen here this can be logically mapped to different execution course.

(Refer Slide Time: 54:50)

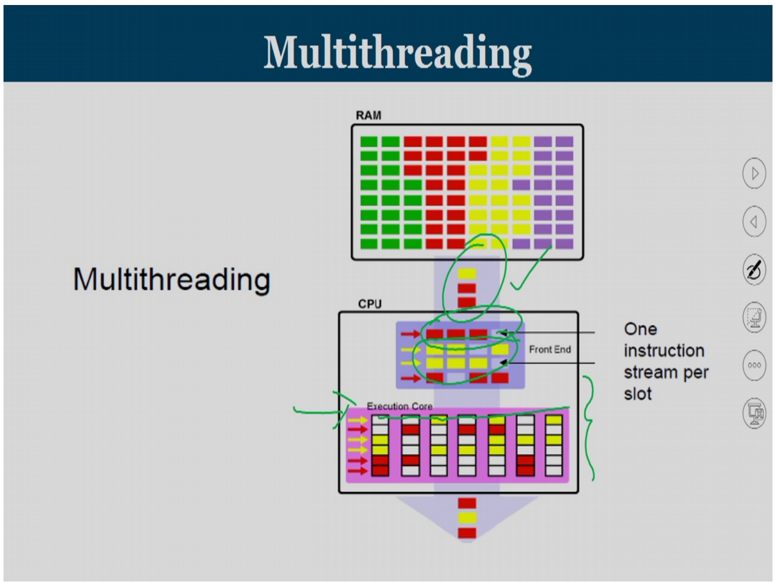

Superscalar Processor

Conventional Superscalar

RAM
Different Tasks
CPU
Wide Issue (4 Instructions per cycle)
Front End
Execution Core
Pipelined Functional Units
Executed Job

So, a superscalar processor means multiple instructions are fetched; multiple instructions are decoded, at the same time multiple instructions are also in execution.

(Refer Slide Time: 55:00)

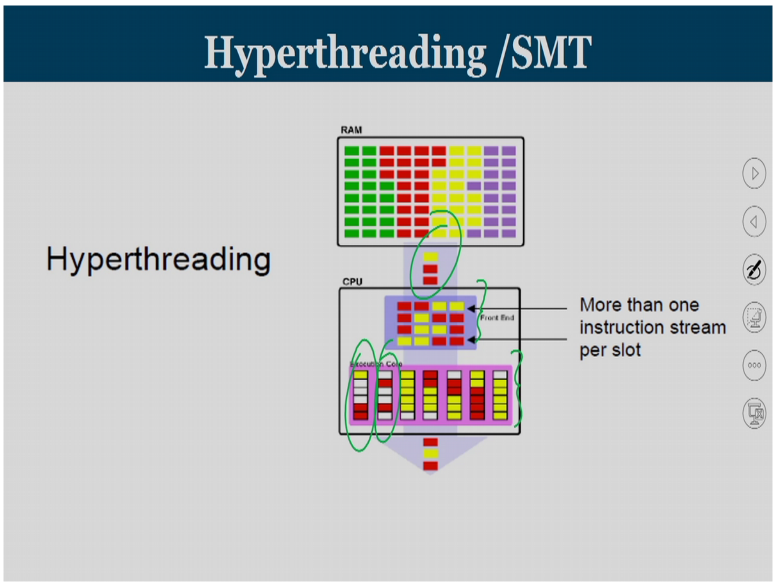

Multithreading

Multithreading

Now, we learn about what is the concept of multithreading. Multithreading is slightly an advanced version of superscalar; where you are bringing multiple instructions. These instructions need not belong to same program. I am bringing one instruction of the yellow program and 2 instructions of the red program. So, if the hardware can bring multiple instructions, but let them belong to multiple programs. And that is to be done in the fetching unit, when it comes to decoding, let us say in 1 clock cycle I decode only the red program or red instruction.

Next clock cycle I decode only the yellow instructions like that. So, given any time slot, either I will be working on instructions belonging to program A or I will be working on instruction belong to program B that is called 2 level multithreading. You can have 3 or 4 or multiple levels. So, the takeaways I am bringing multiple instructions, there can be belonging to multiple programs. But when it comes to decoding and execution, if you look at this horizontally, either it will be blank means, there is no instruction in the pipeline or it will be having only one colour either yellow or red.

(Refer Slide Time: 56:20)

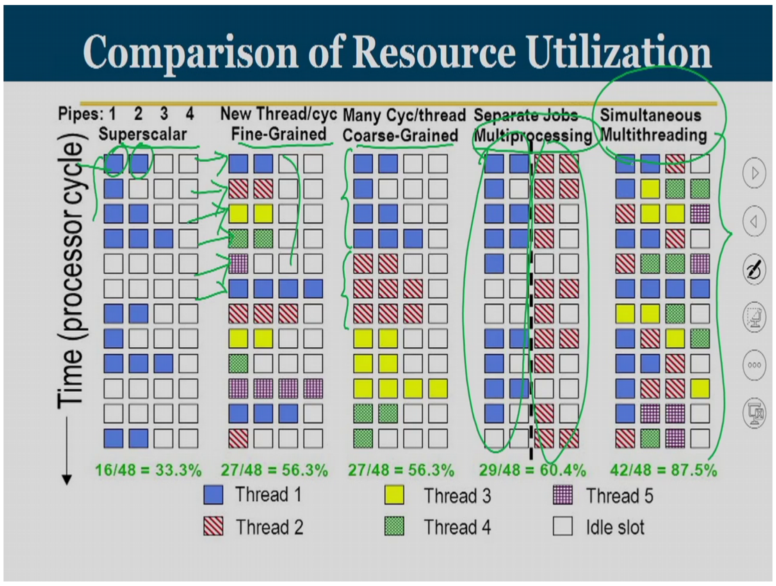


Hyperthreading /SMT

Hyperthreading

The advanced version of this is called hyperthreading. I can bring multiple instructions. They can be from different programs. I can decode multiple instructions together this instructions can be of different context, means different programs. At the same time I can execute one of the functional unit may be working with the yellow program. At the same time some other functional unit will be working with the red program. So, this is known as hyperthreading or it is also known as simultaneous multithreading.

(Refer Slide Time: 56:51)



This is a comparison which will tell about the terminologies that we have learnt. These are all your execution units. Superscalar means, wherever you see blue colour, the corresponding functional unit is busy doing some task of the instruction wherever you see white colour that is an empty slot.

So, in superscalar, I have multiple instructions see I have 2 instructions here I have multiple instructions, that is working in the functional units together. When it comes to multithreading, I can have 2 types of multithreading, one is called fine grained, other one is called coarse grained. See in fine grained this is one cycle, I am now working with the blue program, next cycle I am working with the red program. Next cycle I am switching back to the yellow program, then green program, violet program, again I come back to the blue program. So, every cycle I contact switch between different programs.

So, that all programs will get a feel of they are getting executed. Now coarse grained multithreading means the concept is same, rather than switching across same across adjacent cycles. I use blue program or I fetched and decode from blue program for say 10 clock cycles. At the end of 10 clock cycle is switch to the red program I continue it for another 4 or 5 cycles. So, the frequency of context switching is very high in the case of fine grained multithreading whereas, it will be slightly lower in the case of coarse grained multithreading.

Now when you come to multiprocessing; that means, you have 2 processors let us say one processor has 2 functional units. So, you have a blue program that is running in one of the processor. And you have another red program that is running in another processor. This is basically the concept of duel cores, where you have 2 independent set of pipelines, both are capable of fetching decoding and execution.

One will take care of my blue program other will take care of my red program. Now on top of this I can apply multithreading, where I can have 2 or 3 programs in one core itself which will be context switching very fast. It can be fine grained or coarse grained or the other concept is it can be simultaneous multithreading; where I have multiple programs, it can be fetched in any order decoder and executed in any order.

So, with this 3 lectures we have given a background of how a processor works. We have not gone deeper into the processors. We have covered material which is required in order to understand and appreciate the storage aspect of processors. Now our next day, our focus will be more onto the memory side. So, in memory side, we will focus on how instruction fetch happens, how memory access happens. Since, your instruction pipeline is interacting with the caches, without covering pipelined it will not be good to go into the storage aspect.

So, with this we are completing today’s lecture. I request all the candidates to go through the textbook. Find there are enough reading material on this. And get back to us if there are any queries. One assignment is already posted in the modal just go through the assignment and try to solve it by yourself.

Thank you.