

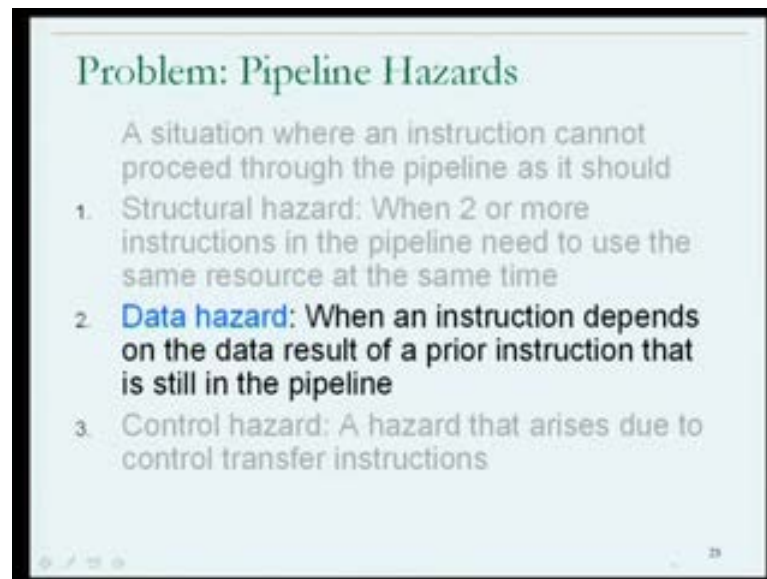
High Performance Computing
Prof. Matthew Jacob
Department of Computer Science & Automation
Indian Institute of Science, Bangalore

Module No. # 05

Lecture No. # 23

This is lecture number 23 of high performance computing. In the previous lecture we had looked at some of the problems that could arise in a pipeline. Pipeline is an implementation of a processor or a piece of hardware to execute instructions in which, in case of a pipeline processor the design objective is to make the throughput or the rate of execution of instructions faster than they would have been in a non pipeline processor by using the different pieces of hardware in some of an overlap node to the extent possible.

(Refer Slide Time: 00:52)



Problem: Pipeline Hazards

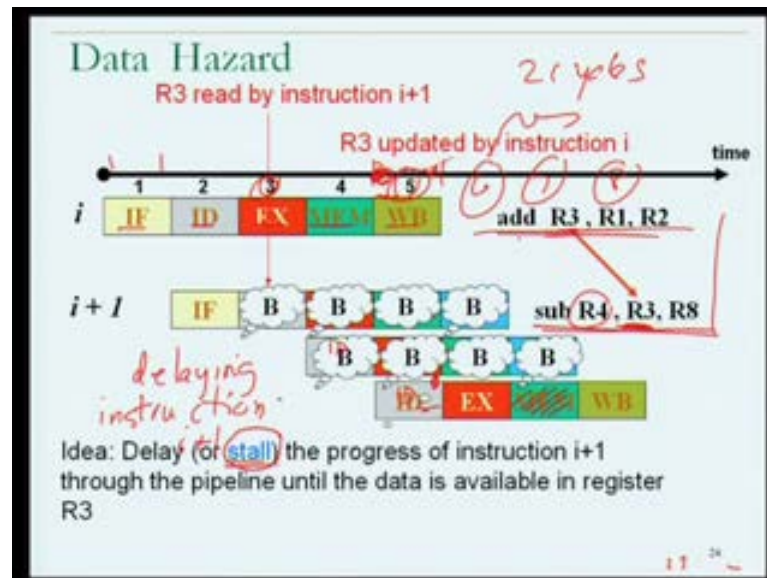
A situation where an instruction cannot proceed through the pipeline as it should

1. Structural hazard: When 2 or more instructions in the pipeline need to use the same resource at the same time
2. **Data hazard:** When an instruction depends on the data result of a prior instruction that is still in the pipeline
3. Control hazard: A hazard that arises due to control transfer instructions

Now, we had looked at an introductory, we took an introductory view on the different kinds of hazardous situations that could arise in a pipeline in the previous lecture and I had seen that there are class of hazardous situations called structural hazards which can be handled by far the most part by appropriate design of the pipeline. By making sure that their resources of the pipeline are designed in such a way that they can be used by as

many instructions as may need them at the same time. And, we were then starting to look at the data hazard which is the situation where an instruction in the pipeline requires a result or a piece of data which is in competition or being produced by an instruction which is ahead of it in the pipeline.

(Refer Slide Time: 01:32)



We were looking at the simple example of an ADD instruction which has R 3 as its destination and is followed by a subtract instruction which has register R 3 as its one of its sources. The problem as we saw it was, we know that based on our definition of what happens in each of the 5 pipeline stages, I refer to I f ID EX MEM w b as the 5 stages of the pipeline. So, we know at what point in time instruction I writes the result into register R 3. We also know at what point in time instruction I plus 1 reads its operand its source operand from register R 3.

For example, we know that the updating of register R 3 as far as instruction I is concerned, happens in cycle number 5 and we know that the reading of register R 3 by instruction I plus 1 happens in cycle number 3. And therefore, instruction I plus 1 is reading its source operand before instruction I has actually written the value that is supposed to be used. Hence, the designation of this has a hazardous situation.

A situation in which, unless the pipeline is corrected or unless something is done, the result of executing simple program instruction sequences like this would be wrong. In

this particular case, the instruction I plus 1, the subtract instruction would get an old value the previous value from register R 3. Right now, based on our analogy from petroleum pipelines we realize that one way that this hazardous situation could be handled is essentially by preventing instruction I plus 1 from reading its operands in cycle number 3. In other words, if instead of allowing instruction I plus 1 to use the ID hardware in cycle number 3 I, pump a bubble of air into the pipeline and that bubble of air will progress to the pipeline in subsequent cycles.

Now the question that arises is, is it safe for instruction I plus 1 the subtract instruction to read its operand in cycle 4? So, by pumping one bubble of air into the pipeline, In other words, by delaying instruction I plus 1 by one cycle I have prevented it from reading the old value of register R 3 in cycle number 3. But, now it will go to the ID stage in cycle number 4 and the question is is it safe for instruction I plus one to read its operand R 3 out of memory out of the register file in cycle number 4 and the answer is no that still too early.

Therefore, I actually need to delay instruction I plus 1 by another cycle. So, after the first bubble of air, another bubble of air must be inserted into the pipeline. In other words, it looks like instruction I plus 1 must be delayed by the next cycle as well. What about the cycle following this? Is it safe for instruction I plus 1 to read register R 3 in cycle number 5? In other words, is this safe for instruction, the subtract instruction to go to the ID stage in cycle number 5? The answer is, we could make it safe for instruction; the subtract instruction I plus 1 to read this operand in cycle number 5. If we designed the write-back stage and the ID stage hardware in such a way that the updating of register 3 happened early in the cycle and the reading of the register R 3 in the ID stage happens late in the cycle.

Remember, I suggested that we are not going to look at the details of what happens within any one clock cycle but, this is an exception because, it should be possible for us to think of designing the hardware in such a way that, within the cycle number 5 it is guaranteed that the write-back will happen early in the cycle and the reading will happen late in the cycle and then therefore, it would be enough for it. It would be safe for us to assume that the instruction I plus 1 could be allowed to use the ID hardware in cycle number 5. So this is something that the hardware designer would have to guarantee but,

it is something which for the moment we assume that can be done; it does not sound unachievable.

So, with this assumption that the write in the write-back stage, the updating of the hardware register happens early and in the ID stage the reading of the register file happens late. So, with this assumption we can actually allow the subtract instruction to use the ID hardware in cycle number 5. Subsequently, it will do the subtraction in cycle number 6. It will occupy the MEM hardware without actually doing any relevant activity in cycle number 7 and we will write its destination register r 4 in cycle number 8.

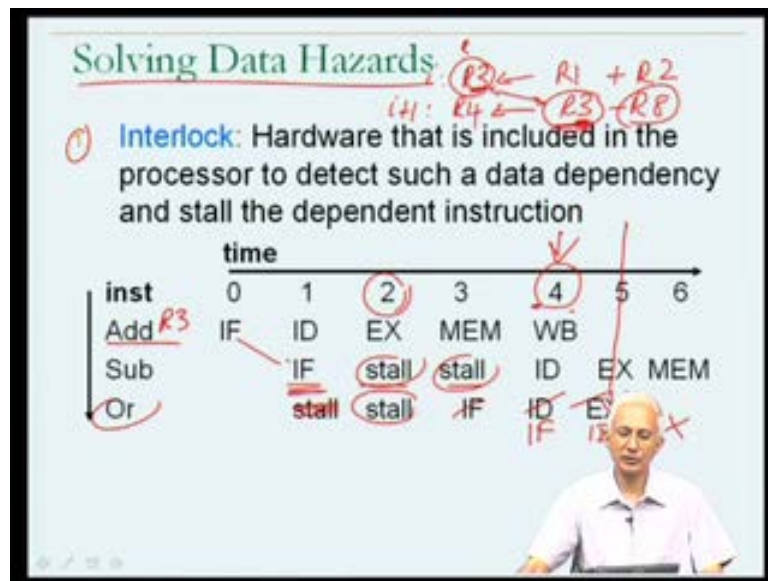
In other words, with a penalty of having two cycles, cycle 6 and cycle 7 in which no instruction completes, we could actually handle this data hazard situation. Note that, an instruction I completed in cycle number 5 after that instruction I plus 1 completed in cycle number 8 which means that there were two cycles where no instruction completed; which is something which we would like to avoid. Therefore, this is not our first choice as far as a solution to the data hazard is concerned. So, just note that what we are talking about here is delaying the progress of instruction I plus 1 through the pipeline that was the suggested solution.

Now, the question which will arise in your mind is this delaying of instruction I plus 1 is obviously not going to be done by the programmer. The programmer is just writing a program or the compiler - the compiler is just generating the machine instruction that therefore, is doing the delaying of instruction I plus 1. The answer is, this is clearly something that has to be done by the hardware. It must be something that is built into the pipeline. Therefore, there must be hardware within the pipeline which can identify but, this is a situation where there is a need to delay an instruction.

Now, the idea here is that we are delaying and a more technical term for this kind of delaying is to refer to it as stalling - s t a l l and this is a verb. So, we could talk rather than talking about delaying an instruction, we will talk about stalling an instruction and the idea here was that we needed to stall or delay the progress of instruction I plus 1 through the pipeline. In this particular case preventing it from using the ID hardware for 2 cycles.

Only then was it safe for it to read the register R 3 and get the right result in terms of what the instruction I was supposed to generate. So, we will henceforth use the term stalling an instruction you know, I just mentioned that the decision whether or not to stall an instruction is not a programmer decision; is not a compiler decision; very clearly it is a hardware decision. So, there must be a piece of hardware; they decide whether or not to delay an instruction in its progress through the pipeline.

(Refer Slide Time: 08:31)



Such a piece of hardware is known as an interlock. So, interlock is the term used for a piece of hardware that is included in the pipeline or in the processor. The purpose of the piece of hardware is to detect such data dependencies and to implement the stalling of the dependent instruction. Now the I am numbering this as one possible solution to data hazards. So, very clearly we are going to see others; let us think a little bit about how complicated this interlock hardware might have to be. You will note that the situation that we had was there was the ADD instruction and there was the subtract instruction; this was instruction I, this was instruction I plus 1, so how will the interlock hardware detect that there is a problem here?

The interlock hardware would have been able to detect the problem, the data hazard by noting that one of the source registers of instruction I plus 1 is the same as the destination register of instruction I and therefore, by comparing the source register identifier of instruction I plus 1 with the destination register identifier of instruction i.

In other words, with a very simple comparator it is possible for the interlock to identify such situations obviously it will have to compare not only the first source operand but, also the second source operand therefore, two comparators and possibly more. But, some number of comparators are what would be required a comparator is a piece of hardware which can compare two things and see if they are the same. So, it is just checking to see if the first source operand of instruction I plus one which is one of the bit fields within the instruction itself.

We remember the instruction format; so its comparing that bit field with the destination operand bit field in instruction. So, just comparing a few sequences of bits is what is involved in this interlock hardware. It is not a very complicated piece of hardware; it does a number of comparisons of that kind. So, if it does find out that there is this kind of a dependence between instructions; then it basically has to cross the dependent instruction to be delayed and has to have a mechanism for rather than allowing the instruction to proceed through the pipeline, cause non activity or a bubble to proceed through the pipeline and so people do talk about pipeline bubbles.

The term bubble was not was possibly motivated by the idea of pumping air into a petroleum pipeline. That is commonly used in pipeline discussions today talking about stalling a pipeline or inserting a bubble into a pipeline. So the situation as we have it if I look at what is happening with a timeline and I look at the different instructions is that the ADD instruction goes through here the cycles are number from 0 through 4. Note that it does not really matter how we number the cycles but, the ADD instruction goes from I f to ID EX MEM write-back in consecutive cycles is followed by the subtract instruction which got fetched in the in time but, then had to be stalled by two cycles before it could be decoded.

So, this is slightly more concise notation; then the previous notation we were using with the diagrams colored block boxes and I may sometimes use notations like this. Since, we now understand pretty much what the content intent in having the colored diagrams the same information is contained in this kind of a diagram. We notice that the I f the subtract instruction is if delayed by 2 cycles is all since safe updating of register I at R 3 as well as reading of register R 3. The update happening at the beginning of the cycle 4 and the reading happening at the end of the cycle 4 ok.

Now, the question which now arises is what if the next instruction we have to worry about the next instruction. Let us suppose that the next instruction also requires use of R 3 and the question that arises is what about the next instruction. Now, if we think about the next instruction we can this can be left out but, if you think about the next instruction there conceivably the OR instruction could have been fetched in cycle number 2.

Unfortunately, the fetch hardware is still not available for use since the fetch hardware is still being occupied by the subtract instruction. When we say that the subtract instruction is stalled we meant that it was prevented from entering the ID hardware which means that it must still be occupying the I f hardware; which means that it is not possible for the OR instruction to be fetched in cycle number 2.

So, the OR instruction starts life by being delayed by a cycle but, we could argue that it could be fetched in the next cycle if the buffering between the IF stage and the ID stage I had shown the buffers the storage between the IF stage and the ID stage is appropriate. So, this is what one could argue but, one sees that if one did allow the OR instruction be fetched in cycle number 3 then it would require the ID hardware in cycle number 4 which would amount to a structural hazard and therefore, one does have to stall the or the OR instruction for 1 more cycle.

Therefore, as far as the OR instruction is concerned, the earliest that it could be fetched would be cycle number 4 followed by being decoded in cycle number 5 etcetera. Now, the question that arises at this point in time is what if the OR instruction also uses as its source operand. Register R 3 remember that R 3 is the destination of the ADD instruction. Is it safe to allow the OR instruction to just go through the pipeline or will it also have to be delayed further as far as entering the ID stage is concerned.

We notice that the assumption about the point in time at which the ADD instruction updates the pipeline updates the register file was that the ADD instruction updates register R 3 in cycle number 4 and therefore, it is obviously safe for the OR instruction to read the register file in cycle number 5. Therefore, there is no problem as far as the OR instruction is concerned, even if it has the same source operand register R 3.

You will remember that the result is the result of adding the contents of register R 1 to result of R 2 and that happens in the ALU during the EX stage of the ADD instruction and therefore, by the end of the EX stage of the ADD instruction the value which is going to go into register R 3 is known, it is available. In fact, it is available in a special purpose register which we refer to as ALU out at the time that we were looking into the inners of the different pieces of hardware. So, subsequently we understand that the actual value which was, which is computed and available an ALU out at this point in time will be returning to register R 3 in the early parts of the write-back stage of the ADD instruction. However, it is available in ALU out as early as this point in time if I look at the timeline.

Now, the next question that we could ask is, when does the subtract instruction actually require that value and the problem that we have is that the subtract instruction by default in the, if I assume that the ADD instruction goes to the pipeline in cycles 1, 2, 3, 4 and 5. The subtract instruction if it just follows the ADD instruction, would have gone through the IF stage in cycle 2. The ID stage in cycle 3, the EX stage in cycle 4 and the MEM stage in cycle 5, write-back stage in cycle 6, if it had just follow the subtract instruction.

Now, that would have been a problem because in the ID stage it would have read the value it would have tried to read the value from register R 3 towards the end of the ID stage which is before the value has actually become available which is only at the end of that cycle. But, we ask another question; we ask the question, when is the result of the previous instruction actually required by the subtract instruction? And, if one thinks about it little bit, one realizes that the subtract instruction is actually doing the subtraction only within the EX stage, that is where the ALU must have at its input.

The value that we are talking about over here, in order to subtract R 3 minus R 4 in the ALU stage, the instruction the subtract instruction is supposed to subtract R 3 minus R 4. In order to do that it must have at its ALU inputs the values from r register R 3 and from register R 4 and you notice that this is a point in time, after the point in time at which the desired value is actually available in the register called ALU out.

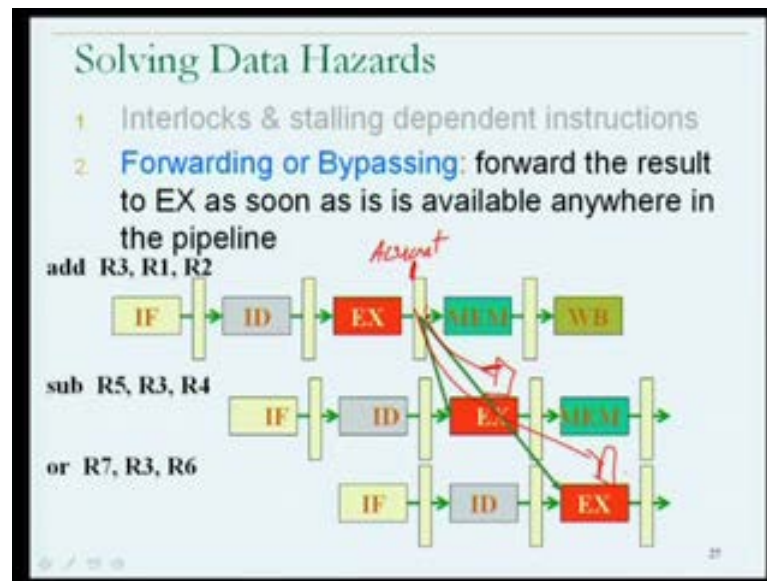
In other words, its looks like there is scope here for avoiding stalling an instruction because that if desired value is actually available somewhere in the processor before it is actually required for doing the subtraction. Therefore, if one could find a way of routing

the desired value from ALU output to the place where as computed by the ADD instruction to the place where the subtract instruction requires its use, then one could avoid having interlocking or stopping, I am sorry, one could avoid delaying and stalling the subtract instruction at all. This is the observation which we could make here and could result in a new technique for handling data hazards.

Now, we may ask what about the following instruction after the subtract which may also be an instruction that uses R 3 and we could note that, that instruction may have the same problem; that the subtract instruction has because, if it was to read its source operands at the end of its ID cycle that would be well before the ADD instruction had written the values computed value into the register R 3. Therefore, both the subtract instruction and the OR instruction suffer from the problem in the current situation.

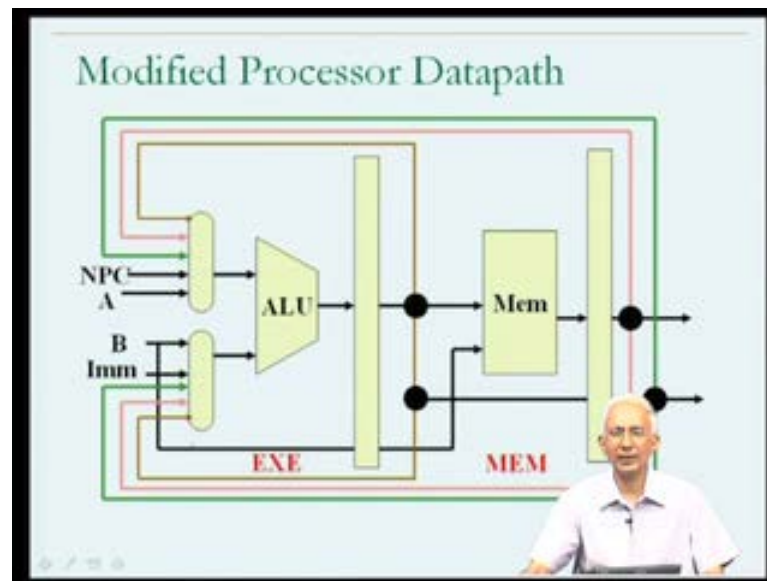
Notice that in this diagram I have not assumed that the subtract instruction is being delayed at all. In the previous diagram, we had the subtract instruction delayed by 2 cycles, stalled by 2 cycles which is why the OR instruction was safe to proceed through the pipeline on schedule. In this diagram, we are trying to think of a mechanism through which we can avoid stalling the subtract instruction at all. That is why I am, I started with the assumption that the subtract instruction proceeds to the pipeline just following the ADD instruction and the hope, the reason for hope was that we saw that the value which is going to go into R 3 is actually available for use well before it is needed by either the subtract or the OR instruction. And that therefore, if one could find a way to route it to each of those instructions by some kind of a connection within the processor within the pipeline then, one could avoid the need for interlocks delaying completely.

(Refer Slide Time: 21:32)



Now, this idea of solving data hazards is called forwarding or bypassing and the general idea is 1 1 should find a way in designing the pipeline to forward in or to root the result into the EX stage as soon as it is available anywhere in the pipeline. In our particular example, we were able to figure out that the desired result as far as the EX stage was concerned for the ADD instruction was available at ALU output over here. Therefore, if one could give a routing path from ALU output to the input of the ALU then, one could avoid the data hazard altogether in terms of the need to delay any instruction. So, what was this amount to in terms of the implementation of the pipeline? It would require modifying the hardware within the EX stage; what will the modifications look like?

(Refer Slide Time: 22:23)



Now, I will just take you back to what the x stage looked like until now. This is something which we have looked at many lectures ago. So, in general in the EX stage of our simplified pipeline, there is the ALU which does the competition whether it be for an addition or for a load word instruction. So, the ALU stage there would be used for calculating base plus displacement address. It could also be used for a branch instruction in which case it is used to compute the target p c in terms of p c plus the displacement. But, in any event it takes 2 inputs: the upper input and the lower input and we saw that there was a possibility that the upper input would have to come from the new program counter p c plus 4 or from the a register.

So, NPC A B and Imm are the names of special purpose registers which we came up with as we were talking about the design of the hardware for the non pipeline processor and then, depending on what particular instruction being executed is this piece of hardware would send either NPC or into the upper input of the ALU and this piece of hardware would send either the contents of the B register or the contents of the immediate register into the load inputs of the ALU (Refer Slide Time: 23:35).

Now, currently what we are talking about is complicating this hardware a little bit more because, there is the possibility that the upper input of the ALU as in our current example should not come from the register file at all. Because, it has been identified that the instruction such as our subtract instruction would have to be delayed and that is we are

able to bypass or short circuit the result which was computed by the ADD instruction into the input of the ALU. How can one do that? One could do that by actually taking the contents of ALU output and providing a path. In other words, a bypass path from ALU output into the piece of hardware out of which something goes into the upper input of the ALU.

Similarly there is also the possibility that the subtract instruction had not had R 3 as its first source operand but, had had it as a second source operand which is why I also show a A bypass path or a root from ALU out which is sort of indicated by somewhere in this region; which one of the special purpose registers in this region of the of the pipeline of the processor.

So, as a bypass path from ALU out to each of the possible inputs of the ALU and modifying this piece of hardware, so that in addition to using the decode information about what the instruction is, it also uses information such as that which will be calculated by an interlock hardware. In other words, it computes that it compares the source registers of the instruction currently being executed with the destination register of the previous instruction. So, this piece of hardware will use all that information to root either ALU out from the previous instruction or NPC or the contents of register A into the ALU upper input and similarly, for the ALU lower input.

So, this requires as we can see, modification of the processor data path but, not whole lot of complication. It requires some hardware similar to the interlock hardware in order to decide which of the various possible upper inputs to the ALU should actually be rooted to the ALU. So, **many** comparators in addition to the information available from the decode and conceivably, a little bit more complication because there may be other bypass parts that have to be included. As we had seen in previous diagram, it looks like if one looks on this diagram carefully, the instruction following the subtract instruction may also require the value computed by the ADD instruction as its source operand and you will notice that is it is not safe for that instruction to read from the register file because register file will be updated later. Therefore, it to requires a bypass path in order to safely get the value which was computed by the ADD instruction and not yet put into register R 3.

So, in order to take care of the next instruction it becomes necessary to include bypass paths from one cycle beyond. Now, the way to look at this is what I am showing you, what I have added into the diagram is the next stage in the pipeline. Remember, that in our pipeline we have the IF stage, the ID stage, the EX stage, the MEM stage and the write-back stage. So, the ADD instruction had progressed and the diagram which is being shown right now, is showing the situation that would have happened if the ADD instruction was occupying the MEM stage and the subtract instruction was occupying the EX stage; which is why we routed the output from the ALU which was whatever had been computed by the ADD instruction as the input to the ALU from the perspective of the subtract instruction.

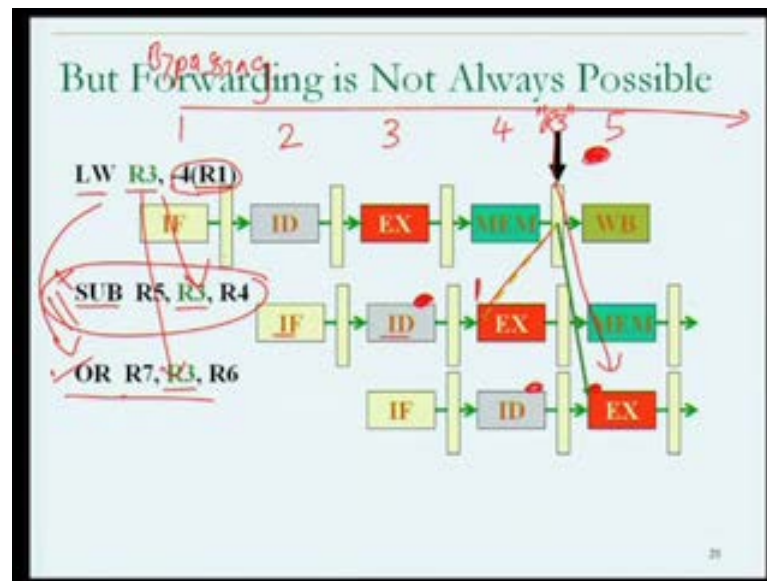
Now, if you are now worried about the OR instruction which follows the subtract instruction then, we realize that at the time that the OR instruction is executed the ADD instruction would have move to the write-back stage.

The subtract instruction will be in the MEM stage and the OR instruction will be in the EX stage and therefore, the value of ALU out which is required is not the value of ALU out as computed by the subtract instruction. What the value of ALU out as computed by the ADD instruction which will be carried along with the ADD instruction as it proceeds to the pipeline and therefore, will be available in the buffers associated with the next stage in the pipeline; which is why I will show two more bypass paths going from the next stage in the pipeline to the inputs of the ALU.

So, let me just clean this diagram up a little bit so the situation as we have it is that for the simple example that we had there seems to be reasonably large number of bypass paths that have to be added into the pipeline in order to take care of the data hazardous situation but, the net result will be that there is no need for interlock hardware which will delay instructions.

We now have a situation where the data hazards that we have identified will not prevent instructions from proceeding through the pipeline as they should. They just have to be correctly identified by the hardware and the correct bypass paths will have to be used to make sure that the correct data is used by the instruction. Now this is well and good as long as bypassing or forwarding can handle all possible scenarios of data dependencies and as it happens.

(Refer Slide Time: 29:12)



For our simple 5 stage pipeline and our MIPS 1 type instruction set, there is at least one situation where forwarding cannot solve the problem which is what the heading of this slide is going to give away. Forwarding is not always possible; so at least in our example forwarding is always not possible let you show me let me show you one situation where the forwarding does not work.

Forwarding remember is the other term for bypassing the idea of short circuiting a value from somewhere else in the pipeline to the input of the ALU in order to allow an instruction to proceed despite the fact that there is a hazardous situation ok.

Now, the example that I am going to construct, we are forwarding is not going to work; is going to be based on the load instruction. So, here we have a load word instruction which has as its destination register R 3 and recall that when this instruction progresses through the pipeline it gets fetched it gets decoded when its source register is read. It executes at this point; its effective address is calculated by subtracting 4 from R 1. It accesses the data cache and finally, updates register R 3 in cycle number 5. Now, let us suppose that the next instruction is subtract instruction which is dependent on the load word. Notice that it has as its first source operand register R 3. It is therefore, dependent on the load word instruction. In the normal cause of events, the subtract instruction will get fetched; it will just follow the subtract instruction, will follow the load word instruction through the pipeline; it gets fetched in cycle 2; it would get decoded in cycle

3 and therefore, in the end of cycle 3 it would read register R 3 and R 4. We know that register R 3 is going to be updated by the load word instruction. In the early parts of cycle 5 and therefore, if this subtract instruction is to read register R 3 in the late paths of cycle 3 then it is going to get the old value. But, that was before we learned about bypassing. So, we now try to figure out whether it is possible for the value to be bypassed or forwarded from the load word instruction to the subtract instruction; so that it becomes available in time for the EX stage to use it right. So, we are not too concerned that the ID stage and the write-back stage because, this was exactly the same situation that we had in our previous example; just have to understand to what extent the bypassing can be done.

Now, we need to understand at what point in time in terms of this timeline, at what point in time is the value which was read from the cache memory available within the processor for use by the ALU. And, if you remember when we looked at the MEM stage, there was this notion that in the MEM stage the cache memory - the data cache memory will be accessed and that once that access has happened the result will become available in a register called I think IMD out or something like that. We called it load memory data; there was a special purpose register called IMD and for all practical purposes the value would be available in IMD at the end of the MEM stage or in cycle number 4.

Now, so that tells us when the value is available this is when the value which is going to go into R 3 is actually available within the processor end of cycle 4. When is the value required by the subtract instruction. Unfortunately, the value is required by the subtract instruction at the beginning of cycle 4 and therefore, there is no possibility of using bypassing. Bypassing can be used whenever there is a forward arrow in time it cannot be used; when there is a backward arrow in time one cannot bypass a value into the past. Therefore, the very fact that this arrow is moving back tells us that this is a situation where bypassing cannot solve the problem.

So, in general bypassing sounds like a good idea and should be used by the processor designers, pipeline processor designers whenever possible but, we now realize that there may be situations where because of when the desired result is produced within the pipeline. The designers realized that a value cannot be bypassed or forwarded appropriately and then the question is what can be done in such a scenario; is it still

necessary to delay or stall the instruction? In this case the subtract instruction we note that.

If we had delayed the subtract instruction by one cycle, what would happen? Well, let me ask that in a slightly different way. If we consider the instruction which follows the subtract instruction and it also is using R 3 as a source. So it too is dependent on the load word instruction; then the normal cause of events, so we are postponing discussion about the subtract instruction for the moment we just consider the OR instruction.

So, it would have been instruction fetched and then it would have been decoded and its operands would have been fetched again too early. But, the question is could bypassing have been used to satisfy the requirements of the OR instruction and we notice that the OR instruction requires the value of the ALU input; the value which is going to go into R 3. It requires a value to be in the ALU input at the beginning of cycle 5 and the value is available at somewhere in the pipeline at the end of cycle 4. Therefore, this is a forward arrow and therefore, that value can, that problem can be solved by bypassing.

Therefore, the OR instruction requirements can be satisfied by the bypassing. Unfortunately, the subtract instruction requirements cannot be satisfied by bypassing. Therefore, we suspect that in an implementation of a processor using this idea we would see bypass paths in connection with load word instruction. In other words, the bypass path from the IMD register into the ALU inputs but, that is essentially (()) the instruction. Not the instruction following the load word instruction but, the instruction which is after the instruction following the load word instruction.

(Refer Slide Time: 35:29)

Solving Data Hazards

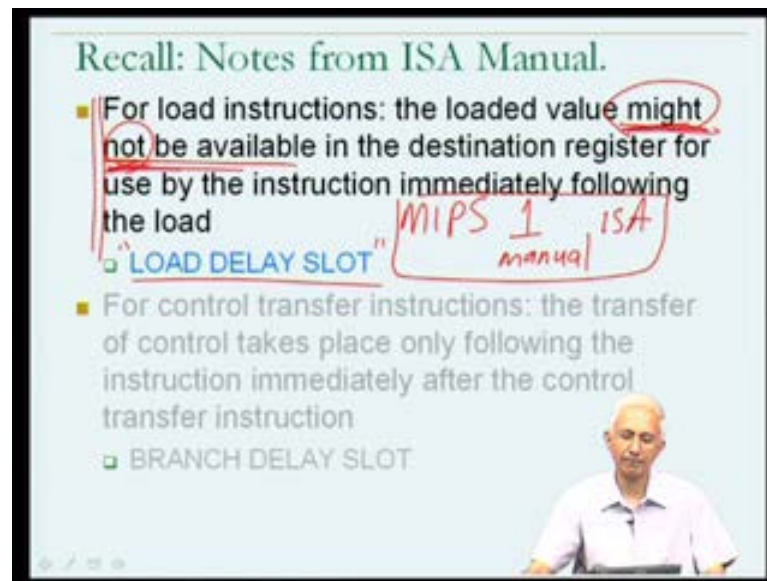
- 1 Interlocks & stalling dependent instructions
- 2 Forwarding or Bypassing *LW R3, 0(R1)*
SUB R2, R3
- 3 **Load delay slot**
 - Build the hardware to assume that an instruction that uses a load value is separated from the load instruction

What then can be done for the instruction following the load word instruction? Right now, this is a situation where forwarding a bypassing was taking as part of the, to an acceptable solution an acceptable solution is one way; there is no time penalty at all. But, in this case it looks like some kind of a time penalty is unavoidable. In other words, it will be necessary to delay the instruction by at least one cycle the subtract instruction in our example but, there may be a better way to view this problem and that is may be to refer to the solution which I will call the load delay slot and this may look like a familiar term.

But, let me first just define the load delay slot as an idea where we built the hardware to assume that an instruction that uses a load value is separated from the load instruction. In other words, we built the hardware in such a way that we assume that if there is a load instruction then the instruction following the load instruction will not use the loaded value.

In other words, we built the hardware to ignore this hazard; that is what this statement is saying. We built the hardware to assume that an instruction that uses a load value such as the situation that we had the load situation that we had was. The instruction that uses the load instruction is the subtract that is the instruction that uses the load word and to say that we built a hardware to assume that the load subtract instruction is separated from the load instruction is to say that we are going to ignore this data hazard completely.

(Refer Slide Time: 37:16)



We have seen this kind of a word earlier. We saw it and we were looking at notes interesting notes from the MIPS 1 instruction set architecture manual you remember that at that time I had put up this slide and now, highlighting the upper half of the slide where there was this interesting note. The note was for load instructions. The loaded value might not be available in the destination register for used by the instruction immediately following the load. So, this was labeled it as in our earlier slide, we had labeled this as the load delay slot; which is the same term that I am using for it. We now understand that what the load delay slot idea is as far as an idea for solving the data hazards. It is an idea which basically says, build the hardware to ignore this kind of a data hazard and in some sense, what is being done is to pass the data hazard as a warning to the programmer.

Let the programmer know that every time the load instruction is used, the instruction following the load instruction is not guaranteed to find the loaded value within the destination register. Since the seed of doubt has been sowed in the mind of the programmer, the programmer will ensure that the instructions are separated from each other and as we have seen, if the instructions are separated from each other such as the load word and the OR instruction then, bypassing can solve the problem. The problem was only with a situation where the instruction immediately following the load required the loaded value.

Therefore, in some sense the load delay slot is a solution to a data hazard problem in which the problem is avoided by just telling the programmer that this problem exists and assuming that the programmer as I said in this slide build the hardware to assume that an instruction that loses a load value is separated from the load instruction. If the hardware makes its assumption it is a programmer's responsibility to make sure that the instruction that this assumption now holds. Again, if the programmer reads the instruction set manual and understands that there some doubt about the availability of the operand he will do that, he or she will write the program appropriately.

Now, the question which may be in your mind is, why is there any a doubt or why is this word? It as might not be available; why do not they just say will not be available or definitely will not be available? The answer to that question is little bit more subtle. We should note that when we talked about this warning, we said that the warning was available in the MIPS 1 instruction set architecture manual and the MIPS 1 instruction set architecture manual could be used as a basis for a hardware designer to build hardware for a MIPS 1 processor.

Now in building a MIPS 1 processor, there is no requirement that a person building the processor build a pipelined processor or build a processor in which there is a 5 stage pipeline with the kinds of problems that we have seen. One could have an implementation of the MIPS 1 instruction set architecture in which there is no pipelining or in which this particular problem does not arise and therefore, in general when writing a program to run on a MIPS 1 processor the user must be aware of the fact that the some implementations of the MIPS 1 processor there may be a problem with this kind of an instruction occurrence

In other words, it might not be there but, in general MIPS 1 instruction set architecture manual does not give guarantees about the availability of the loaded value to the following instruction. Therefore, the programmer in general, could learn more about the particular processor that he or she is using and if he or she finds out that this is not a processor in which this is a hazardous situation could actually choose to write the instructions in a more aggressive way. Hence, that might not be available; this is just a statement of no guarantee coming from the instruction set architecture manual side

(Refer Slide Time: 41:11)

Solving Data Hazards

1. Interlocks & stalling dependent instructions
2. Forwarding or Bypassing
3. Load delay slot
4. **Instruction Scheduling**
 - Reorder the instructions of the program so that dependent instructions are far enough apart
 - This could be done either
 - by the compiler, before the program runs: **Static Instruction Scheduling**
 - by the hardware, when the program is running: **Dynamic Instruction Scheduling**

The slide includes a diagram of a pipeline with stages labeled IF, ID, EX, MEM, and WB. A red arrow indicates a data hazard between the EX stage of one instruction and the MEM stage of the next. A red circle highlights a set of instructions in the pipeline.

So, with this we have seen 3 solutions to the data hazard problem. Let me just quickly remind you there is interlocks where there is a special piece of hardware which compares the source and destination operand fields of instructions and determines whether dependent instructions have to be stalled prevented from progressing to the pipeline that would result in loss of performance. There would be some cycles in which no instruction was completing.

(()) was forwarding a bypassing where a result could be provided from wherever it was available in the processor such as the ALU output to wherever it is needed in the processor such as the ALU input, making it unnecessary for the instructions to transfer, to communicate data from producer to consumer through a register. It gets bypassed or short circuited from the point of production to the point of consumption. And, the idea of the load delay slot where the problem of the data hazard, could be sent in sort of just transfer to the programmer.

Now, another solution to the data hazard problem; if you are currently looking at is the technique called instruction scheduling, we have seen the terms scheduling before when we talked about the scheduling of processes. The scheduling of processes was the decision made by the part of the operating system about which process should run next; which process should be in the running state, next instruction scheduling we would guess as to do with the order in which the instructions should be executed.

Just like process scheduling was about the order in which processes should be running, instruction scheduling must have to do with the order in which instructions should be executed right.

So, the idea of instruction scheduling then must be that if you are given a program that may be you could change the order of the instructions so that, dependent instructions are far enough apart. Therefore, for example, in our example of the load word instruction and the subtract instruction, if I could find a way to change the order of the instructions, if previously the load word instruction was immediately followed by the subtract instruction if I could re-order the instructions change the order in which the instructions are present in the program for example, then I might be able to diffuse the data hazard situation. Hence, the idea of instruction scheduling as a solution to the data hazard problem now the re-order (()) instructions could be done at two points in time. It could be done either by the compiler. In other words, well before the program ever runs and in that case we would call the instruction scheduling static instruction scheduling since the re-ordering of the instructions is done statically one time before the program is running.

As an alternative, the re-ordering of the instructions could be done by the hardware when the program is running and this is what is known as dynamic instruction scheduling. I would not be commenting on dynamic instruction scheduling in this particular course. I will just give you a general idea about what dynamic instruction order in dynamic instruction scheduling might be like now the general idea here is that there is the pipeline processor it has different stages and the instructions proceed from one stage through the other in getting executed.

But, there may be some point in the pipeline at which data hazardous situation might be identified and it might be advantageous to change the order in which the instructions are executed. Therefore, if one could built the processor which could allow the instructions to proceed through the pipeline in their normal order, until that point in time and subsequently keep track of all the instructions which could be allow to run next and pick them one by one in order to proceed through the pipeline.

So we have we have some kind of a pool of instructions which have been gathered after having fetched and decoded then the dynamic instruction scheduler would pick

instructions from that pool, possibly one by one and allow them to go through the pipeline.

So, potentially very sophisticated piece of hardware which would dynamically re-order the instructions and many processors today have dynamic instruction scheduling but, we will try to understand the idea of instruction re-ordering OR instruction scheduling by looking at a static example. So again, just to summarize that many processors today do the re-ordering of instructions dynamically in other words, it is done by the hardware and the programmer does not have to worry about it and the compiler does not have to worry about it much either. But, to understand what the how the re-ordering of instructions might be advantageous from the perspective of data hazards.

We will look at an example from the side of static instruction scheduling. In other words, what could be done inside a compiler before a program runs when a program is being compiled to handle potential data hazards.

(Refer Slide Time: 46:13)

The slide is titled "Static Instruction Scheduling" and contains the following text and annotations:

- Reorder the instructions of the program to eliminate data hazards ...
 - or in general to reduce the execution time of the program
- Reordering must be safe

Handwritten annotations include:

- Top right: $i1 \rightarrow i2$, $i2 \rightarrow i3$, $i3 \rightarrow i2$
- Next to the first bullet: \checkmark ADD R1, R2, R3
- Next to the second bullet: \checkmark SUB R6, R2, R8
- A box around the original code: \times ADD R1, R2, R3; SUB R2, R4, R5
- Next to the boxed code: /* R1 = R2 + R3 */; /* R2 = R4 - R5 */
- Below the box: ~~SUB R2, R4, R5~~; ~~ADD R1, R2, R3~~

A small image of a man in a white shirt is visible in the bottom right corner of the slide.

In order to do this, I use an example; so, the general ideas that we will try to re-order the instructions of the program in order to eliminate data hazards. But, one could think of the objective of static instruction scheduling more aggressively. One could actually say that by eliminating data hazards, we are potentially reducing the amount of time to execute the program because, we may be eliminating many stall cycles many situations where an

instruction could not complete many cycles; in which an instruction could not complete and with a little bit of thought of may be the compiler could be used to analyze sequences of instructions and do re-ordering to come up with a sequence of instructions which results in thus possible program execution time.

So, that may be a generalization of the objective of static instruction scheduling. Now, one very clearly when we talk about re-ordering instructions for example, I have a program which has I say 3 instructions re-ordering the instructions statically means changing the order of the instructions. In other words, it could be that I change the program to read by re-ordering and change the program to read I 1 I 3 I 2 the program was originally I 1 I 2 I 3 the sequence of instructions I changed it by re-ordering into I 1 I 3 I 2.

In other words, I have re-ordered instructions 2 and 3. So, under what conditions can I re-order 2 instructions very clearly the re-ordering must not change the meaning of the program. If by re-ordering the instructions you cause the program to do something different then at re-ordering it cannot be viewed as correct.

So, first start we must do only re-orderings that are safe and here is a simple example of two instructions. Here is an ADD instruction followed by a subtract instruction you notice that the ADD instruction has as its destination R 1 sources are R 2 and R 3 subtract instruction has as its designation R 2 sources are R 4 and R5 and I may ask myself the question, can I re-order these two instructions and with a little bit of thought I realize that if I re-order these two instructions, results would be that the subtract instruction would come before the ADD instruction.

The subtract instruction has as its destination R 2 and if you look carefully you will notice that the ADD instruction has as its source register R 2. Therefore, if I were to rearrange these two instructions, I change the order of the add and the subtract. I come up with this sequence of instructions and in this sequence of instructions you will notice that the ADD instruction and the subtract instruction have a data dependence between them which was not actually the case in the previous sequence of instructions.

In that the previous sequence of instructions the ADD instruction was using R 2 as its source reading the value generated in R 2 by some previous instruction and the subtract

instruction was not using any value generated by the ADD instruction. In other words, if subtract instruction was apparently independent of the ADD instruction. But, when I rearrange these two instructions, I cause this ADD instruction to become dependent on the subtract instruction and therefore, this re-ordering these two instructions would not be safe. Therefore, in talking about instruction re-ordering instruction scheduling one must look at these kinds of scenarios and just to allow our terminology to be consistent.

I would note that in the sequence of these two instructions prior to the attempted re-ordering, you will notice that what these two instructions have in common is that there is a register which is used by both of them and the first the first instruction is used as a source and the second instruction is used as a destination.

As a result of which, it is not safe to just re-order these two instructions and on the other hand, if there had been a series of two instructions here I have written example with add R 1 R 2 R 3; the second instruction is subtract R 6 R 2 R 8. Here once again, these two instructions have R 2 in common but, looking at it carefully I realize that both of them are using R 2 as a source and therefore, by changing the order of these two instructions the meaning of the program will not change as far as these two instructions are concerned.

Therefore, these two instructions can be safely re-ordered. These two instructions cannot be safely re-ordered and we must understand that the relationship with these, between these two instructions is a dependency of some kind with a dependency where there is a read followed by a read of R 2 followed by a write of R 2. As a result of which, the re-ordering would not be considered; should not be considered as safe right. So, short of these kinds of dependencies between instructions it looks like instructions in many situations can be re-ordered. But, that the compiler in doing static instruction scheduling will have to do this checking to make sure that the instructions which it is pairs of instructions which it is thinking of re-ordering are in fact safe to re-order right.

So the kinds of scenarios that we have seen here can be looked at. They are only be a few possible scenarios there is a one where the data hazard that we saw was a situation with 3 3 was the ADD instruction was writing or using as a destination register R 3 in that example and the subtraction instruction was using as a source.

So the different possibilities could be analyzed by the compiler in order to determine what re-orderings are safe and we will continue with our discussion of the different techniques for handling data hazards. In the lecture that follows we will start off with a closing this discussion of static instruction scheduling. We look at a specific code example and look at in what ways compiler could re-order the instructions of that code example in order not only to eliminate data hazards. But, alters it conceivably to improve the execution time of the program containing that code example; thank you.