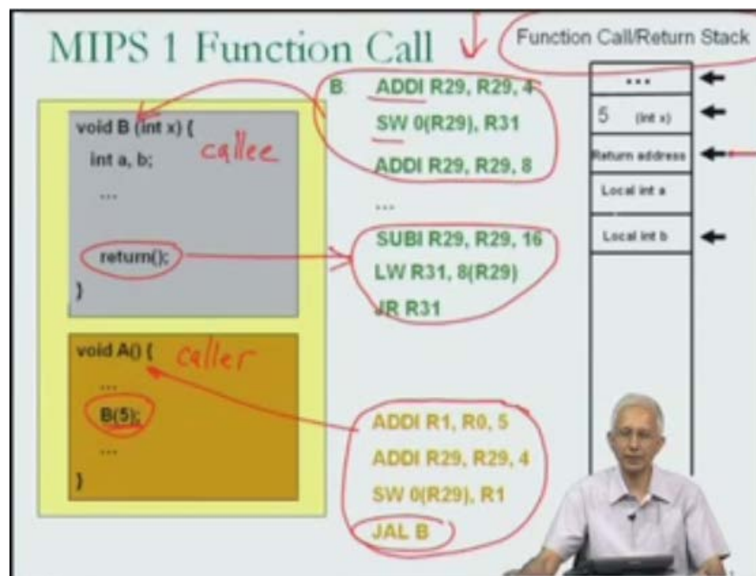
High Performance Computing

Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 02
Lecture No. # 09

Welcome to lecture 9 of our course on, High Performance Computing. We have come a long way in these first 8 lectures. Just to remind you what we saw in the previous lecture, we had studied the MIPS 1 instruction set architecture. This is going to be the machine language that we will use in understanding the behavior of programs. And we have looked at a particular example of the example of how a C function call and return would be implemented in the MIPS 1 assembly language or machine language.

(Refer Slide Time: 00:51)



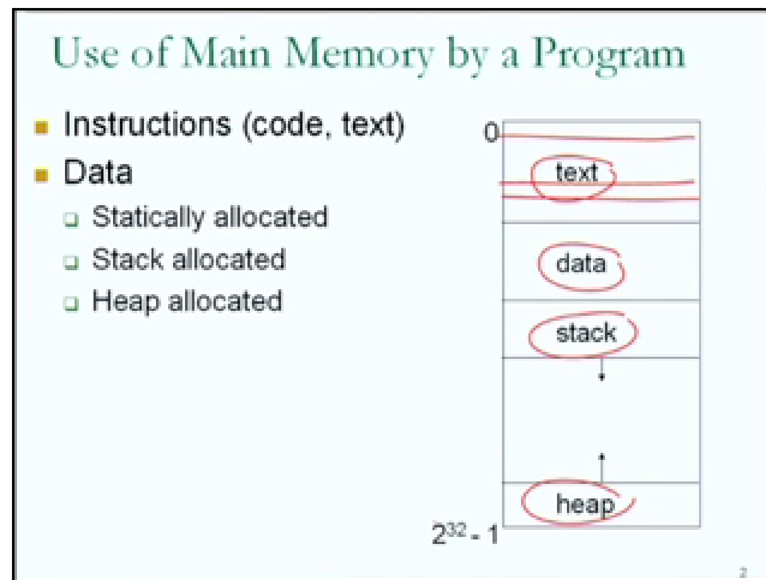
And let me just start off with the first slide where I just recap the sequence of events that have to happen to implement a function call. The function call, in this case, happens with in the function called A, I am circling the function call. And the function is a call to the function B, which is known as the Calle. So, the sequence of events that happens is shown by this column of MIPS 1 assembly language instructions.

So, associated with the function call there are a number of instructions, which will appear inside the body of the complied function A. In addition to that, to implement the function call some of the instructions may be present in the body of the function B. So, if you will look at the complied or assembled version of B, you may find some instructions which actually relate to the function call. Similarly, in function A, you will find some of the instructions, MIPS instructions which related to the function call.

Associated with the function return, one would find many instructions towards the exit point of the or the return point of the function, which is being called, in this case function B. And we understand both the individual instructions are from our study of the MIPS 1 instruction set like the add immediate instruction, the store word instruction, etcetera. And we understood that the functionality of the function call and return is largely implemented through one special MIPS 1 instruction, the jump and link instruction, which makes it possible to transfer control to a function and simultaneously remember the return address to which control is to be transferred upon return.

We understood that lot of the things that happen during the function call or implemented through a stack in main memory. That is why as we ran through the sequence of events we ran through the updates that would happen to the stack. So, this was a good example. It showed us, gave us some practice in understanding MIPS 1 instructions as well as our important function and important functionality, which will happen frequently when we execute programs. In other words, function call and return would be implemented.

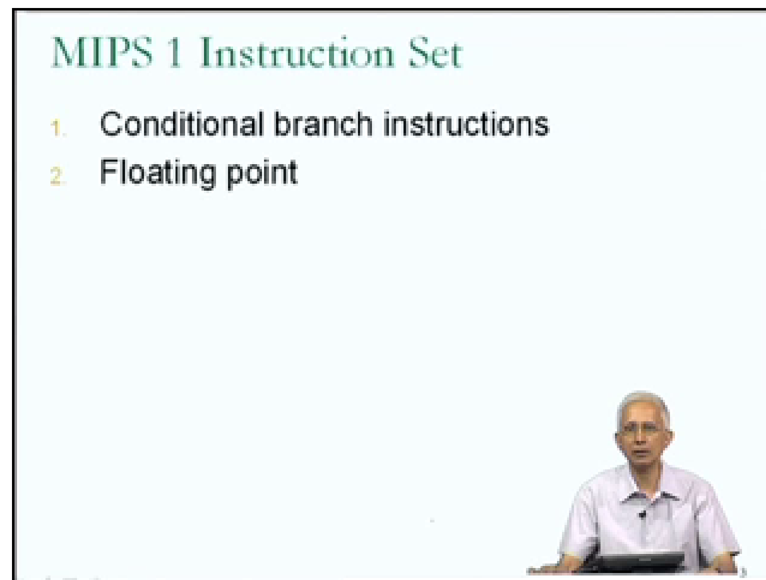
(Refer Slide Time: 03:00)



So, with this, we have a much clear understanding of what to expect when a program executes. And just to remind you that we clearly understand that when for program to execute, it must be present in main memory and that the different components of the program, which are required for its execution must all be present in main memory. Such as, its instructions, its statically allocated data, its stack allocate data, and its heap allocated data. This would all be present in main memory. In other words, for any one of the items, for example, for any one of the instructions of the program, that instruction would be present in 4 bytes of memory. Remember, that each instruction in the MIPS 1 instruction set is a size 32 bits or 4 bytes.

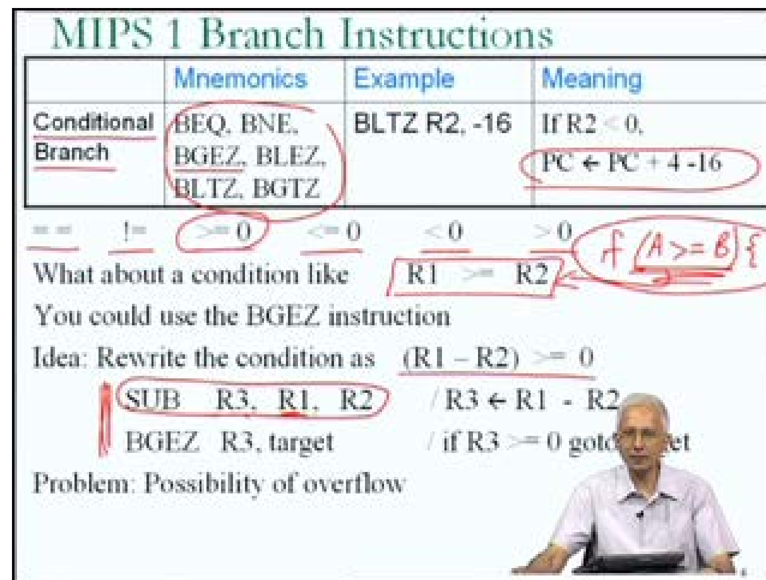
So, we could actually view the pictures that we have over here as describing the situation in memory, starting from memory address 0 going onwards. And we could assume without loss of generality that the lowest address, in other words, the first instruction of the text region of your program will be at memory address 0. And that at the other end to the last element associated with the program might be at the highest possible memory address. If it is a 32 bit machine, this might be 2 to the power of 32 minus 1.

(Refer Slide Time: 04:26)



Now, before actually moving ahead and looking how instructions are executed, I wanted to actually go back and fill in 2 gaps, which had been left in or understanding of the MIPS 1 instruction set. Specifically one relates to conditional branch instructions which we had looked at, but not adequately and secondly, I had made some comments about floating point. We had seen that the instructions of the MIPS 1, which we are studied all related to the integer operations or integer data and I did want to make comments about how floating point, could be dealt with, since some of more examples will have to deal with floating point operations.

(Refer Slide Time: 04:50)



Now, just remind you about the MIPS 1 branch instructions. I used the word branch and conditional branch. View the words, branch and conditional branch is being equivalent. So, there were the six instructions, B E Q B N E, branch if equal, branch if not equal and then branch if greater than equal to 0 etcetera, one example is shown. And in essence were these conditional branches allow one to do is to transfer control by changing the program counter, depending on whether a condition is true or not. And the various conditions, which could be checked, as listed by these six instructions or whether two registers are equal, whether two registers are not equal, whether a particular register has a value that is greater than or equal to 0, less than or equal to 0, less than 0, or greater than zero. So, these are the six conditional branch instructions that we have talked about.

Now, as you would imagine, they are going to be conditions, which cannot be implemented using only these branch instructions. For example, consider a condition like R 1 greater than or equal to R 2. I may have situation where in a C program there is an if, statement of this kind. So, here I want to check whether the value the variable A is greater than or equal to the value of the variable B. And we realize last time that in order to operate on A or B, it is beneficial to have a copy of the variable in a general purpose register. And therefore, a condition like this would translate into a condition of this form; comparing two resisters R 1 greater than or equal to R 2.

Now, if you look at the collection of six conditionals that are available, none of them actually satisfies this objective directly. We have a mechanism for checking whether we have a branch of greater than or equal instruction, but that compares with 0, not two registers being compared with each other. Therefore, this will require a little bit of thought, but it does not take too much time to realize that we could used the branch if greater than or equal to 0 to create the condition that I have underlined over here; R 1 greater than or equal R 2 by the simple mechanism of rewriting the condition as R 1 minus R 2 greater than or equal to 0.

In other words, when if the compiler or if I had to achieve this condition check, then I could actually do the computation of R 1 minus R 2, and if R 1 minus R 2 is greater than or equal to 0 then I know that R 1 is greater than or equal to R 2. Therefore corresponding to this modified version of the condition, clearly I would need two instructions. The first would be a subtract instructions, which would compute R 1 minus R 2 and put the result into some other registers say R 3, and then a branch of greater than or equal to 0 instruction, which would check whether R3 is greater than or equal to 0. And if so then do whatever it would have done if R 1 was greater than or equal to R 2. To this, could be one way to get around the problem of the lack of the appropriate conditional branch instruction in the MIPS 1 instruction set.

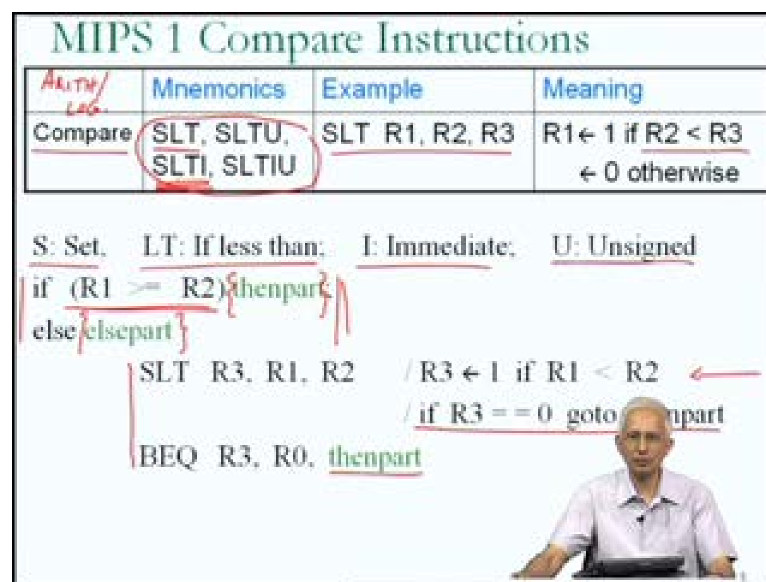
Unfortunately, this is not an adequate solution and the reason is that there is a possibility of overflow. In this particular example you will notice that in order to check the condition R 1 greater than or equal to R 2. I am doing an arithmetic operation. I am calculating R 1 minus R 2, and whenever an arithmetic operation takes place, there is a possibility that overflow will result. And if overflow does result, the program will not actually execute the way that we thought was going to execute.

Now, when you, in your program wrote the if statement, you clearly just wanted that at execution time, condition to be checked. You did not want any arithmetic to happen. And therefore, there was no thought in your mind that there was the possibility of arithmetic over flow. But, this implementation of the condition checking could result in an overflow. For example, if the value of the variable A. let us go Let us talk about the resisters, so if the value of resister R 1 is very high, positive value and the value of the negative R 2 is an extremely high magnitude negative value. Then when I subtract R 2 from R 1, I end up with situation where that value could well exceed the possibilities of

what could be represented within the 32 bits of an integer or a register. And if also does occur then this program will not run as it should and therefore, this has to be viewed as being an unacceptable way to implement the condition.

So, in general, in trying to implement the various conditionals, the various kinds of comparisons that we may want, we have to somehow make use of the limited set of branch instructions available without doing any arithmetic, and that in general is difficult. In all cases, it is not possible. Therefore, to help us in this particular kind of a situation, the MIPS instruction set actually has additional instructions, which I had eluded to when we talked about the arithmetic and logical instructions, but I had not talked about. At this point, it is it is important that we do.

(Refer Slide Time: 09:48)



So, this class of additional arithmetic and logical instructions and this particular table could be added to the arithmetic and logical table that we had. They are called as “Compare Instructions”, and there are four instructions, which I am mentioning. They are the S L T, S L T U, S L T I and S L T I U and in order to read and understand these, once again, I will just let you know what the abbreviations used are in general.

The letter S in these instructions does not stand for store, but stands for set. The L T stands for less than as you would expect. I, stands for immediate, as we have already seen. U stands for unsigned and therefore reading the S L T, I might read it as set if less

than. S L T U is set if less than unsigned. S L T I, is set if less than immediate and so on. The example that we have, S L T I R 1, R 2, R 3, the meaning gives a way the use of the instruction. So, basically what is going to happen is this instruction will cause R 2 to be compared with R 3 without doing any arithmetic.

So, if R 2 is less than R 3 then R 1 will be set to 1, but if R 2 is not less than R 3 then R 1 will be set to 0 or reset. Therefore, the correct way to read or read this instruction might be to say that it is set if less than R 1, R 2, R 3 or most specifically set R 1, if R 2 is less than R 3. In other words, set R 1 to 1, if R 2 is less than R 3. And this gives us a way of implementing conditions without doing arithmetic and raising the possibility of overflow. And that is the sole reason that these instructions are included in the instruction set.

So, let us just look at the example which we had just seen. So, in a program that you have written you have an if, and you have to check a condition of the form R 1 greater than or equal to R 2, and I am fleshing it out a little bit more over here. Specifically, I am in a situation where I have this if, then, else. If R 1 is greater than R 2, I have to executes statements of the thenpart, (Refer Slide Time: 12:02) and I am not going into what the statements of the thenpart are, but could be any collection of statements. Otherwise, in other words, if R 1 is not greater than or equal to R 2, then I have to execute; I want to execute this statement of the elsepart. And I want to check the condition without doing any arithmetic, so I will be using the S L T.

So, the way that this could be done is, first of all, I could see… Now one thing to note here is the condition that I can check is less than, so I can check if R 1 is less than R 2, but, specifically, here I am interested in finding out if R 1 is greater than or equal to R 2. So, one has to think a little bit carefully about this, but let us just see what code I have written and we will check if it sounds correct. So, this S L T instruction that I have written is going to set R 3, if R 1 is less than R 2. Here, I have written a comment, just to explain what the intent of this instruction.

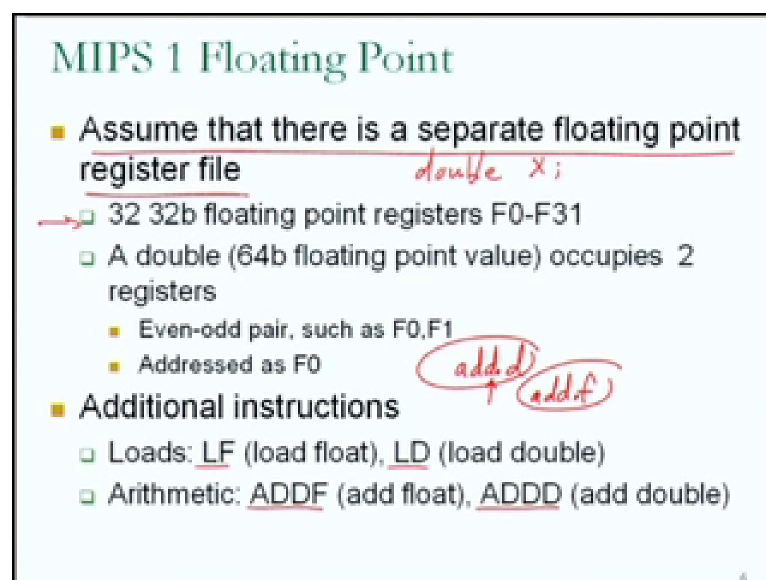
The intent is that if R 1 is less than R 2, then R 3 will be set otherwise R 3 will be reset. We will have a value of 0. What is it mean if R 1 is less than R 2? This means that R 2 is greater than or equal to R 1. And therefore, if R 1 is less than R 2 and that is the situation where I actually want to execute the elsepart. Bearing this in mind, if R 3 is actually set

equal to 1, I want to execute the else part, which means that if R 3 is not set to 1 by this instruction, I really want to execute to thenpart.

So, with little bit of thought, we realize that I can use the check for less than, in order to handle this particular condition, but depending on whether in this particular case if R 3 is equal to 0, I want to execute the thenpart, and if R 3 is 1, I want to execute the elsepart. If I invert follows I check whether R 3 is equal to 0, using the branch of equal instruction, and if R 3 is equal to 0, I transfer control to the thenpart, as we had just reasoned.

Therefore, this is again, I suggest you to try to look at this more carefully. Look at the various possibilities and satisfy yourself that this is a correct implementation of if, then, else and further there is no danger of overflow happening, as a result of the condition checking. So, in general, the S L T, the compare instructions, were included for this purpose. I should mention that there is no additional set of set if greater than instructions. These are the sole compare instructions that are available. Suggesting that with this set of instructions, in other words, just the capability of setting if one register is less than another, one can actually achieve all the possible condition checking that could be required along with a branch instructions listed in the previous slide.

(Refer Slide Time: 15:10)



There is also the possibility of the comparing a register with an immediate value. So much for complete understanding of the branch instructions, the second issue relating to

the MIPS 1 instructions, which I wanted to complete before moving forward relates to Floating Point. You will remember that our discussion of the MIPS 1 instruction set discussed only integer operations, integer registers, and so on. But I had mentioned in passing that their extensions are in MIPS 3, MIPS 4, MIPS 5 instruction sets, there was floating point arithmetic available.

So, for the moment we are going to assume that where that the MIPS 1 implements floating point instructions, is using a separate set of floating point registers. In other words, the floating point values are not loaded into the registers that we called R 0 through R 31, those of used only for integers, the floating point data will be loaded into a separate set of registers. And regarding the separate set of registers, we will understand that there are 32 bit floating point registers, just like they were 32 bit general purpose registers called R 0 through R 31. There are 32 bit general purpose floating point registers and we might hence for prefer to R 0 through R 31, as the integer general purpose registers or as the integer registers.

So, we have F 0 though F 31, each of which can contain a 32 bit floating point value represented using the I triple e floating point representation. And in this case we … So, now we do know that there is a possibility of double floating point values, to the question does arise of what about a 64 bit floating point value. Such as would have resulted, if I had a C program in which there was a variable called X declared as double (Refer Slide Time: 16:49). In this case, the way that things are implemented in the MIPS 1 would be that that value would be loaded into a pair of registers; two registers adding up to 64 bits and the way that this would be done is that an even, odd pair of registers will be used. What I mean by this is the pair F 0, F 1 with F 0 being register that is used to name the pair.

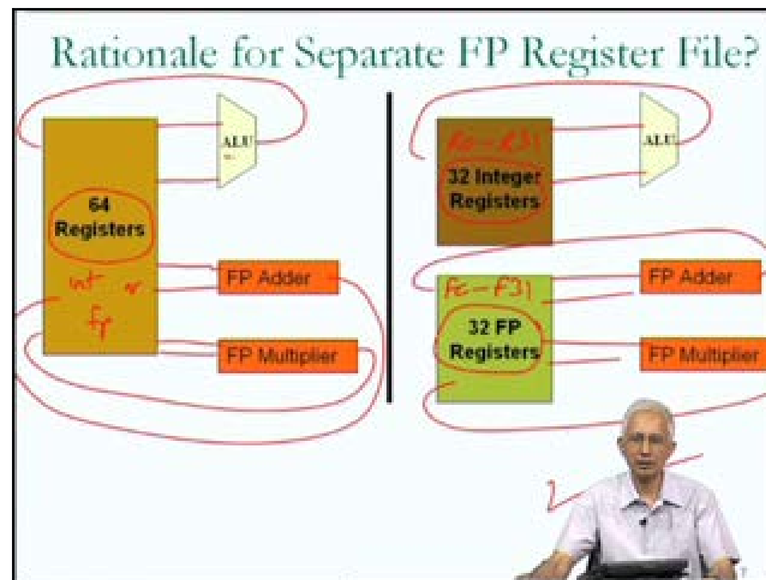
So, one can also handle double precision or 64 bit floating point values using this convention, and this could be what is implemented by the hardware, which takes care of the programs. So, that is about the registers. There will obviously have to be additional instructions, because I cannot add two floating point values in registers F 0 and F 1, using the add instruction for integers, because the add instruction for integers, related to hardware that could do integer addition, so it must be additional instruction for floating point.

Now, let us mention a few of them. First of all, there must be load and stored instruction. Just as we had related to the integer general purpose registers R 0 through R 31, they must be a separate set of load and store instructions, relating to the floating point general purpose registers. In our examples, I may refer to them as load, float, which will load of 32 bit value from memory into one floating point register or load double, which will load a 64 bit floating point value from memory into an even-odd pair of registers such as in F 0, F 1.

So, there will be have to be loads and stores, because frequently used data will still have to be copied into registers for the benefit of our program. Then, in addition to this, they must be arithmetic instructions and the kind of notation, which I will use is that I will assume that there is an additional add instruction for floats, which I will call ADDF and since we will also concerned about doubles, there might be additional add instruction for doubles or 64 bit floating point values, which I will refer to as ADDD, which you can read these as add float and add double. In some books, you may find different notation used. For example, we might use in notation add dot d for add double or add dot f for add float. (Refer Slide Time: 19:02) But, bearing in mind that these are just assembly language type notations, it does not really to matter too much what we use in our examples. Since, I have not used this dot notation in our examples; I did not want to add it at this point.

So, we will use notation like this ADDF, ADD D and so on. Now just coming back to our assumption over here assuming that there is a separate floating point register file I had again mention this in passing when we talked about general purpose registers, I did want to make specific comment about this.

(Refer Slide Time: 19:39)



Addressing the issue of why is there or why is it a good idea to have separate floating point register file? In other words, one side of general purpose registers for integers R 0 through R 31, and in other set of general purpose registers F 0 through F 31 for floating point values. And I will just show you this in terms of some pictures. So, let us suppose, I did not have a separate register file for floating point values. Since, I know that a single program might use both integers and floating point registers; I might assume that the people who are designing the machine would include more than 32 registers, may be 64 registers. And that all of these 64 register could conceivably be used for either integers or floating points, and this is one possible way to design the registers, if one wants to have both integers and floating points supported.

The alternative is as we had seen the MIPS 1 does a separate set of 32 registers which we call R 0 through R 31 for integers and another one for floating point values, which we are calling F 0 through F 31. So, these are the 2 possibilities. Let us think about the possibility on the left (Refer Slide Time: 20:57). Now, if there is a collection of 64 registers, which could be use either integers or floating points inventively, then as far as the integer A L U is concerned, the inputs to the integer A L U, which is the piece of hardware does the arithmetic, associated with the integers, they will be have to be connections from the register file into the integer A L U. And the same holds for different floating point pieces of hardware.

For example, the piece of hardware, which does floating point addition, takes it operand out of the floating point, out of the registers and therefore, they will have to be connections into the floating point ADDF from the 64 registers. Similarly, for the floating point multiply.

What about the result from the A L U? So, the A L U takes two values from registers, add R 1 R 2 R 3 and put the result back into the register file. The same will have to be true of the floating point adder and of the floating point multiplier. So, we see that there is large number of connections. The connections between the floating point register between the register file and the different functional units and they could be additional functional units. For the moment we are looking only at three and A L U, which does all the integer functions, the floating point adder and a floating point multiplier.

And if I had a situation where I separated the integer and floating point registers, then the connections into the A L U would come only from the integer register files and the connections into the floating point pieces of hardware will come only from the floating point register file. And therefore, as far as the designers of register file is concerned, things become a lot simpler, the complexity of the interconnection becomes a lot simpler. And therefore, from the hardware designers perspective, the second option is the winner (Refer Slide Time: 22:43) and this is actually the reason that we find the idea of separate floating point register file being done.

So, we seen that in the MIPS 1, we are going to assume that there is a separate integer and separate floating point collection of registers, in both cases 32, in both cases the width of the registers is 32 bits and what the different kinds of instruction in float might be available in floating point.

(Refer Slide Time: 23:04)

MIPS 1 Floating Point Code Example

```
double A[1024], B[1024];
for (i=0; i<1024; i++) A[i] = A[i] + B[i];

Loop: LD      F0, 0(R1)
      LD      F2, 0(R2)
      ADDD    F4, F0, F2
      SD      0(R1), F4
      ADDI    R1, R1, 8
      ADDI    R2, R2, 8
      BNE     R1, R3, Loop
```

We quickly go through an example. Just make sure we have already seen examples using the integer MIPS instructions. Therefore, I thought would be good to do one using the floating point MIPS instructions.

So, here we are going to look at what code might result from this kind of collection at declaration in for loop in C. So, this particular piece of code is dealing with two double position, in other words, 64 bit floating point arrays, array A of size 1024, array B of size 1024, and then there is this for loop, which is basically adding element by element. An element of A to an element of B and that sum is the new value of the element A, and this happens for all the 1024 elements of B arrays.

So, this is a vector sum operation. The size of the vector is 1024. Now it has happen is this piece of code could be implemented using this kind of sequence of MIPS 1 floating point instructions. You will notice that there are floating point instructions, the load double, load double, add double, store double, but the remaining instructions are from (( )) integer instruction set, which is why we need to comment on this a little bit. Even to do floating point operations, to implement this piece of code, which looks like purely floating point code, some integer instructions may be needed. In other words, instruction from our previous discussion of the MIPS instruction set with a little bit of thought, you will realize that much of this must be coming from the for loop.

Because to implement the for loop, I must have a conditional branch instruction. As long as the condition of for loop is true, we have to branch back to the beginning of the loop. So, we are not too surprise to see a conditional branch instruction and similarly, to implement the iterations of the loop, in other words, in a sense, incrementing of the loop count, we will obviously need some add instructions and that is approximately where the additional add instructions are coming from this. I will describe this example in more detail later.

But, this is a typical kind of an example using floating point instructions to implement a floating point loop, and since it is a floating point loop, they will also be some integer instructions in it. So, the entire MIPS instruction set will be of use to us in many of our examples. So, with this, well it might actually make sense for me to go through this little bit because of the addressing mode issue. So, you notice that this piece of code seems to be starting with some assumptions. The first instruction is loading at double, which must be loading the value of A (I), into register F 0, F 1.

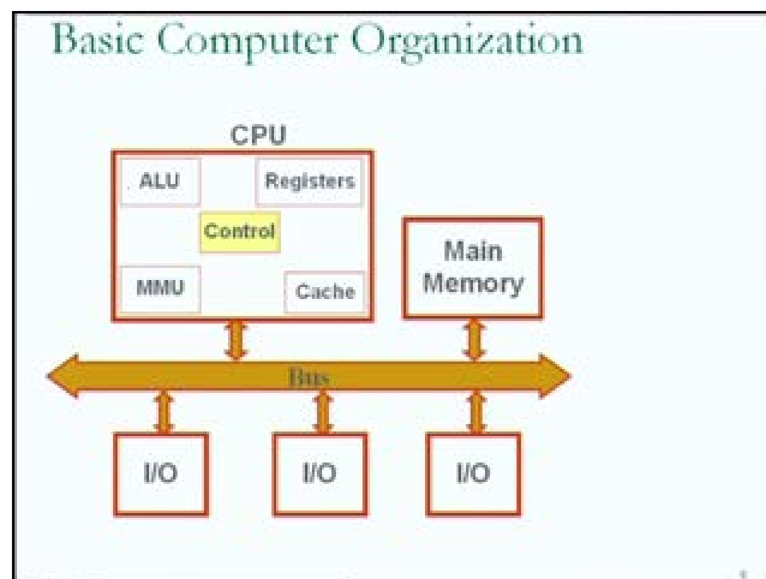
Remember, these are double values. So, whenever I refer to F 0, it is actually the pair F 0 F 1. F 2 is a pair F 2, F 3; F 4 is the pair F 4, F 5 etcetera, which is why you find only even numbered floating point registers used in this example. The first load double instruction is loading the value of A (I), into the register F 0. And the starting assumption of this piece of code is that the address of A (0), is available in register R 1, which is why we just start by loading from base-displacement at 0 of R 1 into register F 0. Later, in this loop, you notice that we increment R 1 by 8, and the effect of the incrementing R 1 by 8, is that R 1 will now contain an address which is 8 more than what is use to contain and therefore, it now contains the address of A (1) and this is how we are actually setting through the loop.

We increment from A (0) to A (1), using this ADDI instruction, changing the address inside register R 1. Similarly, the other instruction increments us or steps us through array B and finally, there is another assumption at the bottom, which is that we set up the termination condition of this loop using register R 3 and therefore, it is sort of piece of suppose that register R 3 contains an address toward the end of array A, possibly the last element an array A and in fact, now register R 3 contains an address beyond the last element in array A. So, that we I am sorry yeah. So, that as soon as we find out that R 1 is not equal to R 3, as long as R 1 is not equal to R 3 we know how to keep looking back,

but as soon as we have incremented R 1. So, that it becomes equal to R 3. We know that we have just gone past the last element in the array

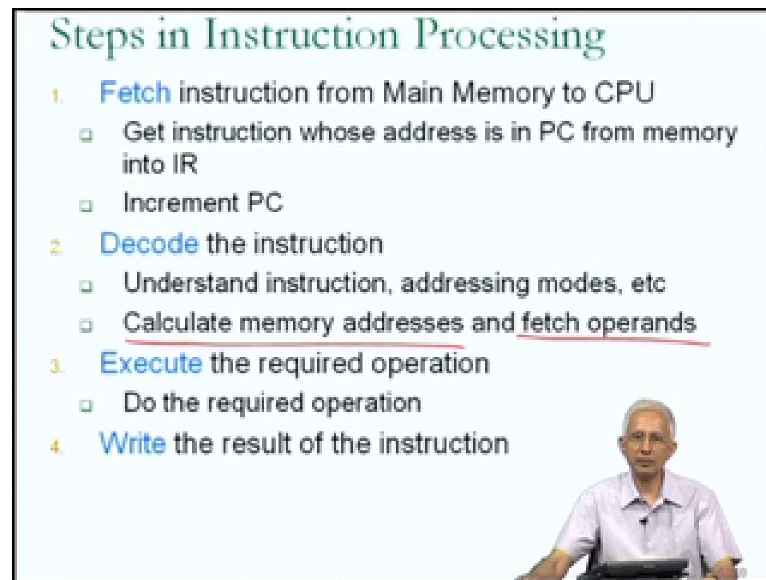
So, R 3 is initialized to contain the address just beyond the end of array A. So, there were these base assumptions that we made in writing this piece of code and this base assumption would be implemented by 2 or 3 instructions just outside the loop, which initialized R 1, initialized R 2 and initialized R 3 as appropriate, so much for this example. Now, with this we have a much better understanding of the MIPS 1 instruction set and we will go back to our discussion of what happens when an instruction has executed.

(Refer Slide Time: 28:12)



So, we are back to our picture of the hardware. We are trying to understand what are the sequences of steps which must be done by the hardware, in other words, what is the control hardware do in order to execute an instruction.

(Refer Slide Time: 28:26)



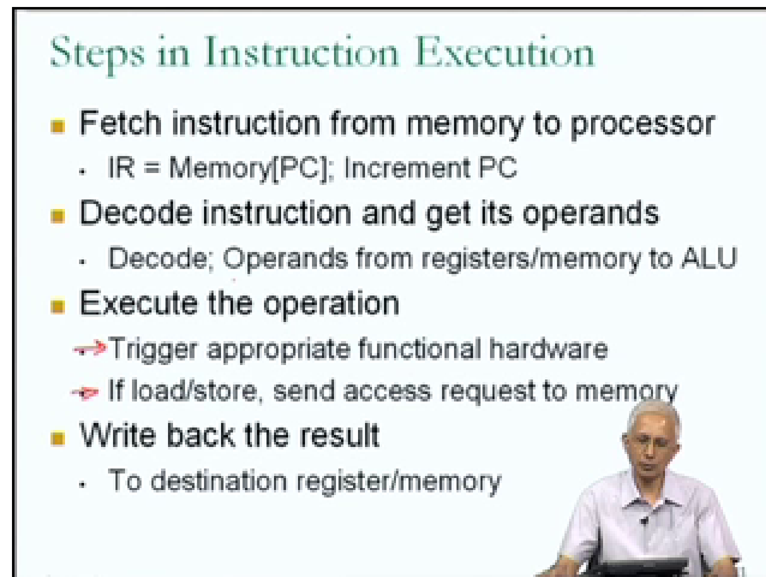
And towards the end of the previous lecture we had outlined the series of steps. We saw that the instruction is initially present in main memory; it must be fetched into the CPU. After that it must be completely understood. Then the operation associated with that instruction can be done and finally, the result of the instruction can be return back to its destination.

And the fetching of the instruction will involve communicating the address of the instruction, from the program counter to the main memory. Main memory will return the instruction bits, which can be stored into the instruction register, inside the processor. After this, we will increment the program counter and in other words, when I say we, I am referring to the control hardwares, so that the next instruction in memory will by default be executed next. So, the program counter will be incremented. The instruction decoding involve understanding the bit pattern inside the instruction and interpreting them as the operation like what addresses, what operand, what addressing modes are associated with this particular instruction, using information from the instruction format section of the instruction set architecture manual.

Subsequently, if there is a memory address and it is an addressing mode, which requires some computations, such as the base-displacement addressing mode, the calculation will have to be done, such as adding the contains of the base register to the sign displacement

from the instruction and the operands will have to be fetched. Subsequently, the required operation can be executed and the results can be return back.

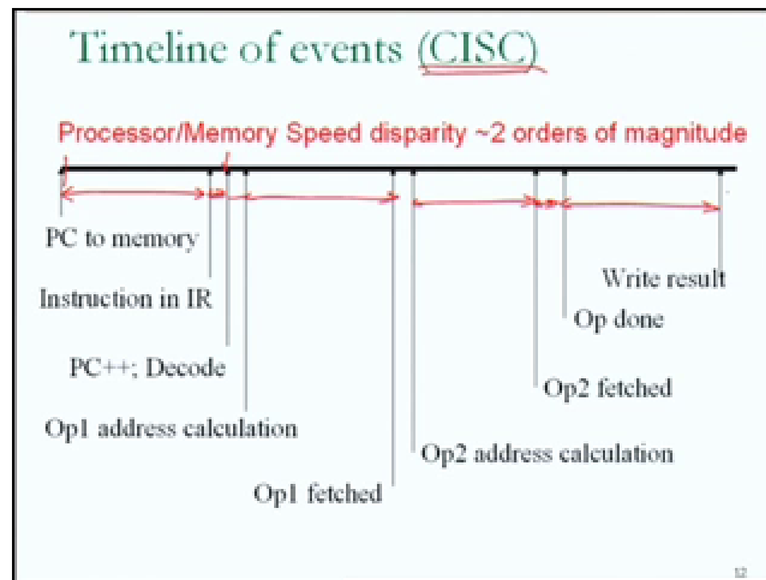
(Refer Slide Time: 29:51)



So, we had outlined this in a little more detail, these 4 steps in this form and I will just try your tension to the fact that in this executing the operation will involve triggering the appropriate functional hardware. For example, if it is an add instruction, the A L U must be triggered, to do the particular operation, the add operation. Note that typically an A L U, as from its name arithmetic and logic unit, the A L U is capable of doing many different kinds of operations and therefore, when I say trigger appropriate functional hardware, I mean, the appropriate functional hardware with the appropriate operation clearly specified.

Now, some of the instructions as we have seen may be instructions, which do not have specific hardware of this kind, circuitry associated with them. For example, the load and store instructions through which data values are copied from main memory into registers; do not have functional units or pieces of hardware associated with that. So, executing the operation in the event of a load or store, involves sending the address of the piece of data to memory and that is a specific part of executing a load or store instruction.

(Refer Slide Time: 31:10)



Now, let me just run through this series of events on a timeline. This will be useful exercise, and to start this of, forget that we are currently concerned about the MIPS 1 instruction set. Let us think about more general instruction set or specifically a CISC like instruction set. Now, in a CISC instruction set, remember the operations could be arbitrarily complicated, the instruction could be arbitrarily complicated, and the particular example that I am going to show, I am going to assume that this is a CISC instruction set where all the operands are specified using memory addressing modes.

In other words, there is an add instruction which may have add x y z. Three Memory operands as it is, operands two source operands in memory, one destination operand in memory. Now, what I mean by a timeline of events is I am actually going to show the sequence of events in terms of the order which they might happen.

We are going to draw this timeline, showing intervals for the above appropriate amount of time in any one of these steps might take so that the first thing that happens is that as we saw the program counter value has to be sent to memory, and since memory is operating on a time scale which is two orders of magnitude slower than the processor, it will take a fair amount of time, which I am showing by this large interval in the timeline. So, this might be the hundred nanoseconds that I was talking about. So, at the end of a hundred nanoseconds, hundred nanoseconds after the program counter value has been

sent to memory, the memory will respond with the instruction and the instruction will end up in the instruction register.

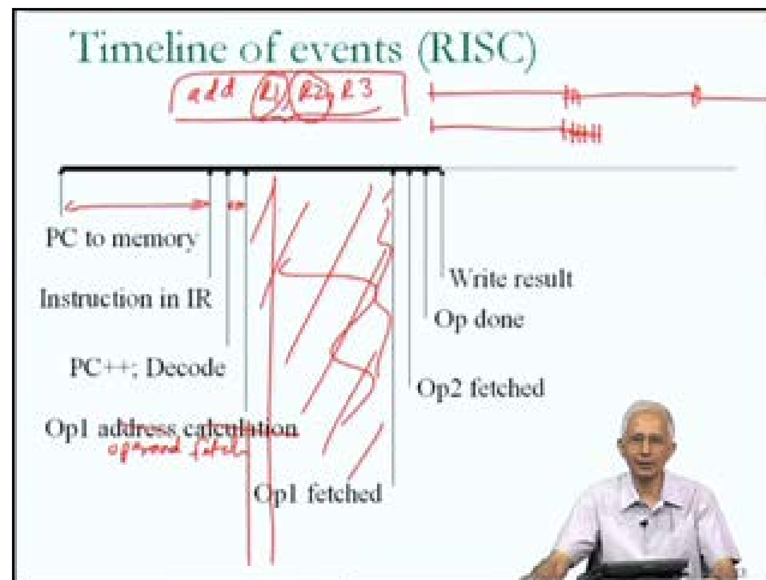
The next thing that can be done is the program counter can be incremented and that is not going to take much time therefore, I show it by a very small interval on the timeline. This could be that one nanosecond as well as the instruction being decoded. So, both of these might happen in this much amount of time. So, the way you should look at this, is this small time interval, may be about 100, the time interval to its left.

What happens next is the instruction at the end of this point in time the instruction has been understood. Therefore, it operands can be fetched. Now, for each of its operand given at this is a CISC instruction set, they may be the need to calculate its address. Each of the operands might is a memory operand and it is possible that the operand has to be, address has to be calculated using a base-displacement addition. So, there is this small amount of time to do the operand one. In other words, the first source operands address calculation that might be the amount of time to do an addition of base register value plus displacement. And it might take that much time because its approximately the amount of time to do a signed integer addition.

Subsequently that address can be sent to memory and memory will respond with the value of the first source operand. So, that is that 100 nanosecond time interval that we talked about. The same thing will happen with the second source operand. So, its address will have to be calculated. It too might be specified in base-displacement addressing mode. A long time in which the operand is fetched once again the hundred nanoseconds for the memory to respond, then there is at this yeah then the operation itself will happen. If this was an add instruction then amount of time do the add, which is a small amount of time, because it is happening at its CPU speed. Subsequently, the result will be written back to the destination and once again the destination is in memory. Therefore, it takes a large amount of time.

So, just remember this is a timeline, which is drawn assuming that we have about two orders of magnitude difference in the speed of the processor and the memory. That factor of a hundred; one nanosecond versus hundred nanoseconds that I have just referred to. So, this is approximately what the previous outline would mean in terms of the sequence of events on a timeline.

(Refer Slide Time: 35:16)



We could draw something similar for RISC scenario. In the RISC scenario, the base assumption would be the timeline (( )), I am showing the timeline much small. But the base assumption would be that the less it is a load or store instruction, the operands would be coming out of registers.

And that the only situation in the execution of an instruction where memory would be involved, would be in fetching the instruction from memory in the first place. Therefore, here we are considering something like an ADD R 1 R 2 R 3 RISC instruction. So, associated with the operands there is no need for any memory operation, but associated with fetching the instruction, there is a need for memory operation and therefore, the sequence of events would initially look very much like they did for the CISC instructions machine. In other words, in the beginning the program counter value does have to be sent to memory and memory will respond a 100 nanoseconds later with instruction, but subsequently, the instruction can be the program counter can be incremented and the instruction can be decoded at processor speed, very small amount of time. The operand address calculation can be done. In this case, the operand address calculation is actually done as part of the decode, just determining that the first source operand is R 2, and reading R 2 out of the register file, which may take about this much file (Refer Slide Time: 36:31).

So, instead of calling this address calculation, I might call it as operand one, operand fetch. In other words, the amount of time that it takes to read the value of register R 2. So, this interval of time would not be necessary. Let me just quickly correct this. So, just forget about this interval of time. This is not necessary. So, shortly after operand one has been fetched, which will happen over here. Operand 2 will be fetched, which will happen somewhere over here. So, just ignore that large graph that is a mistake and very soon after the operation will be done.

So, in effect, when I look at the timeline, I will redraw the timeline over here. We started off with a large interval of time for fetching the instruction from memory. After that there was a small interval of time for fetching, for incrementing the program counter and decoding the instruction. The small interval of time for fetching R 2, a small interval of time for fetching R 3, possibly. A small interval of time for doing the operation, and then a small interval of time for writing the destination register R 1. As suppose to the CISC timeline which looked well beyond the edge of the screen.

So, just to compare the two, even with this error in this particular diagram, the amount of time to execute the one RISC instruction was much smaller than the amount of time to execute that very complicated CISC instruction, which gives us a little bit of an idea about why the RISC instructions set is, as I had indicated is little bit more popular than CISC instruction sets today. Let us just think about this a little bit more. The only way to think about this is to actually look at some pieces of code or piece of code, which is written both in the RISC and in the CISC instruction set and trying to see what happens, because from these timelines all that we can understand is that we might expect that the amount of time to execute a simple RISC instruction could be much less than the amount of time to execute a complex CISC instruction that is largely dominated by the need to access memory multiple times, but could be due to other factors as well.

(Refer Slide Time: 38:46)

So, as a quicker side, let us just look at a small code segment and try to look at the CISC versus RISC equivalents, in terms of what might the result of compilation of the code segment might be. I am taking a somewhat odd example. Here, we have 2 C instructions, which relate to operations on an array A and an array B. The first instruction is adding the ith element, the first statement is adding the ith element of A to the ith element of B, and making that the new value of the ith element of A. As the side effect it increments the index into A. It increments the variable I and I plus plus, the second instruction takes the Bth element, the ith element of B, subtracts one from it. Decrements the index I and makes this the new value of B (i).

So, sum 2 instructions I am sorry sum two statements. Now, in a RISC instruction set, as you will imagine there is the need to load each of the operands. In other words, A (i) will have to be loaded into a register, B (i) will have to be loaded into a register. After that the addition can be done, after that the result can be stored into A (i) and so on. So, we expect that each of these references to an array element will cause either a load, if it is on the right hand side or a store if it is on the left hand side, instruction to be generated in the case of the RISC instruction set.

And briefly, the net result might look something like this (Refer Slide Time: 40:25). Two load instructions corresponding to the references to A (i) and B (i) in the right hand side of the first statement. Then there is the ADD corresponding to the add of those two

values. After this, there is a store corresponding to the storing into A (i) and other store corresponding to the storing into B (i), and the updating of the values of, well in this particular case, the values of the array (( )). So, this may not be entirely necessary, but some sequence of instructions, as you would quickly understand. Whereas, in the case of the CISC instruction set, as I indicated, if each of the instructions can have multiple Memory operands, then the first instruction might just be an add, and over here, we have the notation that we had used for post increment addressing mode (Refer Slide Time: 41:24) and the post increment addressing mode might be useful for the i plus plus operation over here.

Similarly, the post decrement addressing mode might be useful for this i minus minus addressing mode over here. The net effect in spirit is that there may be very few instructions that have to be executed in order to achieve these two statements in a CISC instruction set. So, in the previous slide, we had seen that the amount of time to execute a single RISC instruction is going to be much less than the amount of time to execute single CISC instruction. But we now learn that if you look at the piece of code, C code, it might compile into much larger number of instructions in RISC instruction set, then instruction in a CISC instruction set. Therefore, the question of which of these two kinds of architectures will cause the program to run faster still remains open, less time per instruction, but no instructions for RISC more time to instruction, but less instructions for CISC.

So, just give us some idea of where this is heading. In this particular example, there are 8 instructions, as far as the RISC equivalent is concerned. They are only 2 instructions as far as the CISC equivalent is concerned, but I will point you to another attribute of these 2 pieces of code which is the number of memory access is involved in each of these species of code. Now, in the case of the CISC instruction set there are 2 loads, and 2 stores, so that there are 4 memory operations involved. In the case of the CISC instruction set there are 1, 2, 3, 4, 5 memory operands, which are involved in this piece of code. In other words 5 memory accesses which have to happen in addition to the instruction fetch.

And therefore, it is not entirely clear which of these will result in a faster program. Many more instructions for the RISC, but fewer memory accesses as far as data is concerned. So, that is why people often talk about the RISC versus CISC debate, and we would not

delve into that, but for the purpose of simplicity of our examples, we will from this point on not talk about CISC any more, we will exclusively talk assuming that with dealing with a RISC instruction set and a RISC implementation of the instruction set or RISC implementation of the instruction set.

(Refer Slide Time: 43:33)

We will assume that ...

1. Activity is overlapped in time where possible
   - PC increment and instruction fetch from memory?
   - Instruction decode and effective address calculation
2. Load-store ISA: the only instructions that take operands from memory are loads & stores
3. Main memory delays are not typically seen by the processor
   - Otherwise the timeline is dominated by them
   - There is some hardware mechanism through which most memory access requests can be satisfied at processor speeds (cache memory)

Our next subjective is to understand more about what happens when the instruction is executed, in terms of what kind of hardware will be associated with that, and what kinds of properties we can expect that hardware to show. We do need this understanding because we want to understand adequately what happens inside the hardware, inside the software when our program executes. So, I am going to make some assumptions at this point. These assumptions we stand pretty much for the reminder of the course.

Now, for the first assumption which we will make is that in the implementation of the hardware, wherever possible, if there are two activities they can be done simultaneously, in other words, overlapped in time, in other words happening at the same time; then we will assume that the implementation of the hardware will do that.

So, if activity can be overlapped in time. In other words done at the same time, then we will assume that is done. Please consider a simple possibility. Now, we saw in our description of the steps involved in execution of an instruction, we saw that in the very first major step may be you are trying to fetch the instruction from memory instruction;

the program counter value was send to the main memory. Memory responded by providing the instruction and then the program counter was incremented.

So, the question is it possible to do the incrementing of the program counter and the fetching of the instruction from memory at the same time? In other words, can these two activities be overlapped in time? Until now, when we do the timeline we were showing all the activities is happening one after the other, the timeline progressed to the right. If I can overlap two things in time then instead of the sequence of events taking four, if two of them can be overlapped in time then they will happen at the same time and therefore, the amount of time for that sequence of steps will be reduced. Therefore, this is actually a very important issue whether activities can be overlapped in time.

So, can be operation in the memory and the incrementing of the program counter happen overlapped in time? So, I end this with a question mark. Because, they cannot be entirely overlapped in time, in the sense that we know that the fetching of the instruction from memory starts by reading the value of the program counter and sending it to memory. And the incrementing of the program counter involves modifying program counter value and therefore, the incrementing of the program counter causes the increment to happen before the addresses gone to the memory then the overlap cannot be done safely.

The two operations cannot be initiated at the same time. But we know that the amount of time to do an increment, incrementing by 4, in order to do the increment, the value of the program counter will have to be read and then 4 will have to be add to it. And at the time that the value in the program counter is being read, it can be used both for incrementing or adding 4 to it and for sending to the main memory.

So, the sending of the address to the main memory can be done with the same read from the program counter and therefore, it is the case these two can be safely overlapped. There will actually no risk involved. Let me just look at one more example. In the second stage of the instruction processing, there was a need to fully understand the instruction, which I described as instruction decode and subsequently the operands had to be fetched. But there is a possibility that one of the operands is a memory operand for such as in the case of a load or stored RISC instruction.

In this case, an address calculation would have to be done, adding the contents of the base register to the displacement from the instruction. So, can the understanding of the instruction happen in parallel with the competition of the effective address? Now, we note that what it means by understanding of the instruction, which is examination of the instruction while it is inside the instruction register. In order to calculate the effective address of the operand, the fields of the instruction are necessary. In other words, the identity of the base register, the value of the displacement and those are likely to be read from the instruction at the same time that the instruction is being decoded. Therefore, both of these activities are using the instruction as it resides in the instruction register and it does not conflict with each other. Neither of them modifies the instruction as it reside resides the instruction register.

And therefore, once again these two activities can safely be overlapped if so desire. So, in each case where the person, who is building the hardware, has to make a decision about how soon he can initiate the next step, these kinds of questions have to be asked, but, we will always make an assumption of possibility of overlapping operations if in our mind they will seem to be independent of each other, and by doing this, we will hopefully have an implementation of the processor, which is faster than the otherwise be. Now, second assumption that will make is going to assume that we have a RISC instruction set, which is load store. In other words, the only instructions they take memory operands a load or store instructions.

We are not going to talk about CISC instructions sets at all, in terms of studying the instruction processing, the implementation of the hardware. Now, another somewhat that assumes that will make is that we will assume that main memory delays are not typically seen by the processor. Now, this is a significant assumption, and may not be a realistic assumption, because main memory delays are significant. Main memory, as we have seen, could be a 100 times slower than the processor and even to fetch an instruction from memory into the processor is going to take that amount of time.

Therefore, this assumption seems to be unrealistic. Let me just point out that if we do not make this assumption then as we have seen the timelines are going to be dominated by the memory delays and therefore, talking about how fast a processor could become point less. Ultimately, everything will be decided by how fast the memory is. The timeline for

CISC and in fact, even the timeline for RISC was dominated by the 100 nanoseconds they did to take to fetch the instruction from memory.

Therefore, this turns out is an important assumption that we must make and that raises the question of why make an assumption? The assumption might be important for the speed of the efficiency of our ideas. If it not realistic, how can you make such an assumption? Fortunately, it is a realistic assumption. We are not going to see how for the movement, but I will just state that if we assume that there is some kind of a hardware mechanism through which most memory access requests can be satisfied the processor speeds. Note the word, most. We do not need a guarantee that all the memory access requests are satisfied at their processor speeds.

Then this assumption turns out to be more or less ok, because the assumption must that main memory delays are not typically seen and that relates to our most over here. So, for example, if we have some kind of an idea that 95 percent of the time memory request can be satisfied their processor speeds, then the assumption is actually to some extent justified and therefore realistic and therefore justified.

Now, the name of the hardware mechanism, which is going to make this possible, is cache memory and we will be studying more about cache memory later. But, we now understand why there was a box labeled cache as C A C H E. I will pronounce that as cache inside our block diagram further A L U. Right one of the two blocks, which I had added in one of the earlier lectures inside the not inside the A L U, but inside the processor was a box labeled C A C H E or cache and we now realize that it is a very important and integral part of the processor, which makes this set of assumptions possible, reasonable for us to make.

Now, at this point, we are ready to start looking at the implementation of the hardware inside a processor, and we will be continuing with these three assumptions, and we start looking at what kind of simple hardware and what module of the hardware that might be present in the processor, will give us a good enough understanding of how this steps in the execution of an instruction would be achieved towards a better understanding of how a program executes.

Thank You.