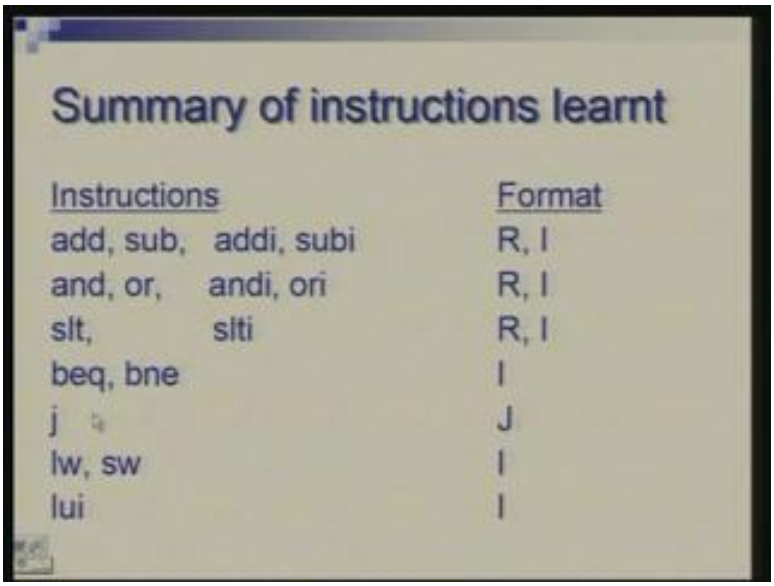
Computer Architecture
Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 4
Instruction Set Architecture - 2

We are discussing instruction set level architecture with the help of a very simple instruction set for MIPS processor. We have seen that this architecture has 32 registers each of 32-bits and they are very simple instructions with which we can perform arithmetic add, subtract and other logical operations and we have also looked at some branch instructions. We will continue further in that direction.

(Refer Slide Time: 1:44)



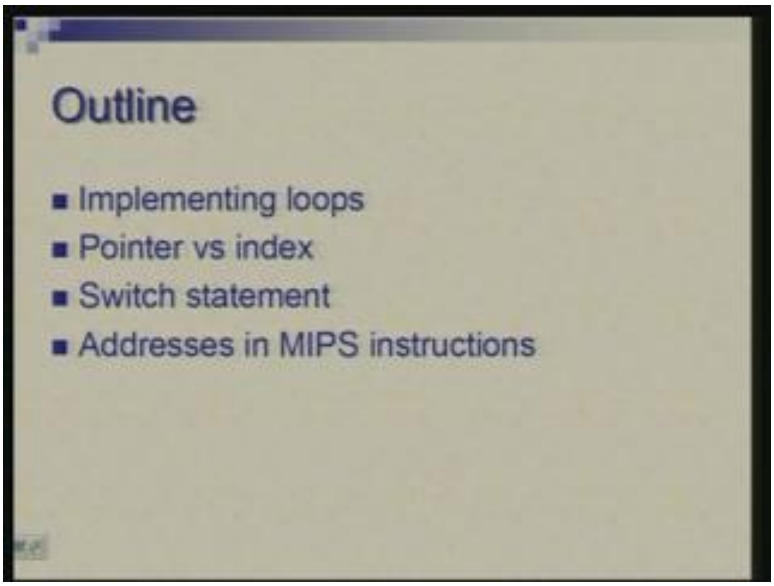
Here is the summary of instructions we have discussed so far; add, subtract, add immediate, subtract immediate the latter two are for operands which are constants. Similarly we have AND and OR and their immediate version: and immediate and or immediate; for comparison of less than or greater than type we have slt instruction which also has an immediate version.

So in all immediate instructions one operand is still a register and second operand becomes a constant. Then for equality and inequality comparison there are beq and bne instructions. For an unconditional jump there is a j instruction or jump instruction; for moving data between memory and registers there are load and store instructions. In particular we have seen load word and store word these two instructions.

When you have to load a constant which is a large 32-bit constant we need to do it in two parts and first we have to worry about load upper immediate which loads upper half and

then you can put back the remaining half of the right half. It is important to know what is the format of these instructions. Format is the way in which instruction is divided into fields. There are instructions which are of R-format or register format where apart from opcode you specify up to three registers. I-format is required wherever there is a constant either for performing an operation or for specifying an address. These are I-formats where there is a field of 16 bits so that a 16-bit constant can be provided. Yet another format was J-format which is for jump instruction where 26 bits which are left after having a 6-bit opcode are used for specifying address.

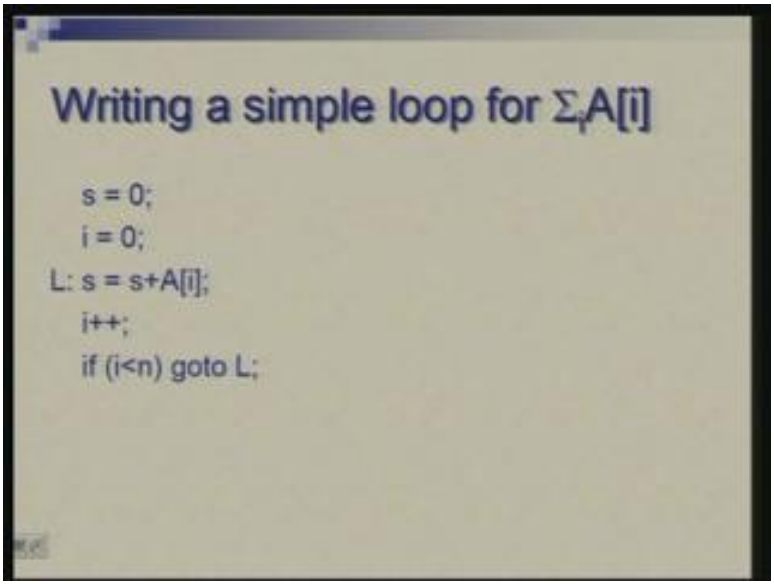
(Refer Slide Time: 3:53)



Today we will continue further from this point and talk of how we can have a further controlled construct. We have seen a case of simple (if then) or (if then else) type of structures where a simple check using typically beq bne instruction possibly in conjunction with slt could be used. Same instructions can be used for executing loops we do not need to introduce new instruction and we will see that. We will see that typically in a loop you access an array and that can be accessed with index or a pointer. We will see what is the difference between these.

We often need switch type of statements where you need to take a decision which has multiple outcomes so we will see that as well and finally we will go into little more depth of how addresses are actually given in various MIPS instructions. There is a variation from instruction to instruction and we will like to understand some of those subtle differences. Let us start with a very simple example typically probably the first thing you learn in your programming is summing elements of an array.

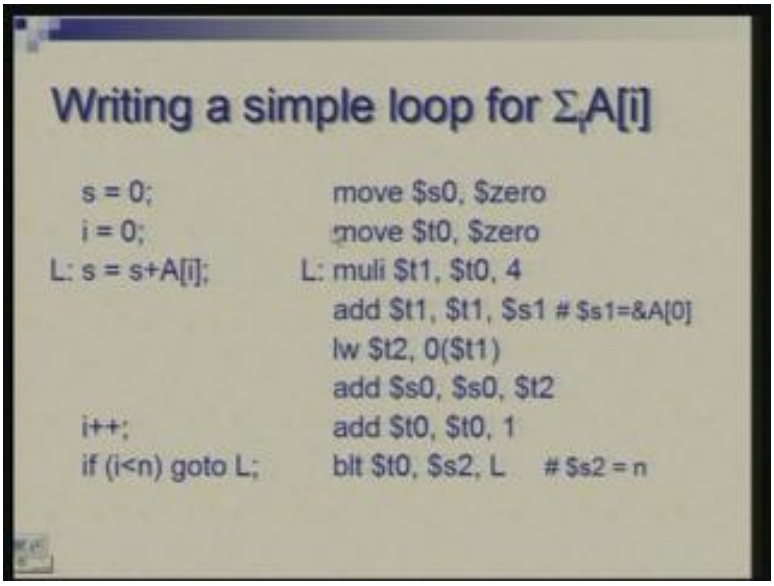
(Refer Slide Time: 5:09)



It is a straightforward thing although I have written in somewhat unusual form where instead of using a for structure I have put in terms of conditions and goto so that I can easily correlate with what we are going to write in assembly language.

A is an array and n is the size of the array, there are n elements. We begin by intialising a variable which is going to hold the sum then there is an index and this is the main body of the loop where you take element of array and sum it, update the index and perform loop termination check and go back. So L is the beginning of the loop and this is the body of the loop, so this is a header or prefix of the loop (Refer Slide Time: 6:07). Now how we do this in assembly is what is of interest today.

(Refer Slide Time: 6:18)



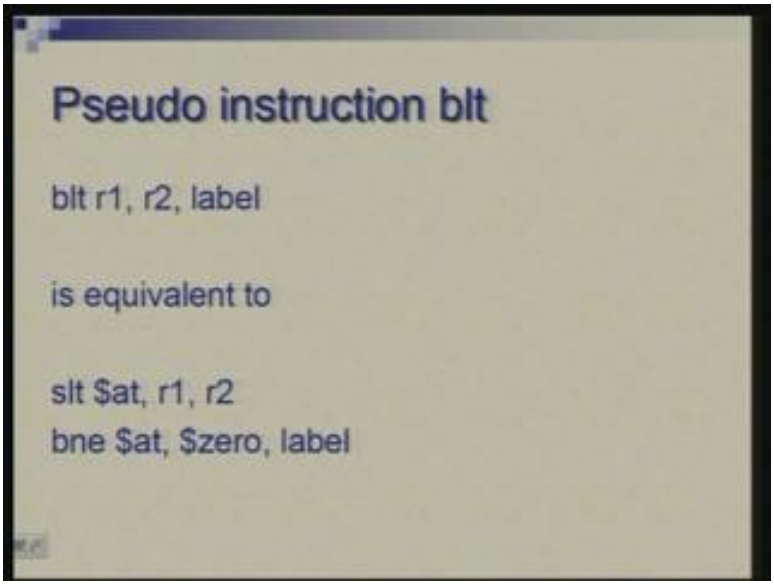
These instructions these steps are implemented by a very straight forward move instruction where you move a zero value into some register. Now remember the correlation between the variables and the register we are using. We are using s0 register for holding the value of the sum; we are going to use t0 register to hold the value of the index. Now, here we have written a single statement s equal to s plus a[i] but this needs to be broken down into a number of steps. First of all we need to prepare the address to make an excess to the array.

As you know that each element if you are talking of integers is going to occupy 4 bytes. So, from the starting point of the array the offset of A[i] would be four times i. So we actually multiply t0 with 4 and t1 now contains four times i. This is added to the starting address of A. Here I will just put a comment that s1 contains address of A[0] the address of first element (Refer Slide Time: 7:38). These two were added together to give you the address of A[i] and once we have that we can use load word instruction and the offset here is 0, the word loaded comes to t2 so t2 is a register which is temporarily used and finally this gets added to the running sum which is in s0 as you notice in the beginning so these two are added the result is in s0.

Therefore it is 1 2 3 4 these four instructions are actually doing the job of this statement and then simply update the index, compare with s2 I am assuming that s2 contains n the size of the array so t0 which contains i is compared with s2 and if t0 is less than s2 we branch of to L that means start the loop again.

Now I have not talked of blt instruction. blt is a pseudo instruction; it is not a real instruction. In some tutorials I have mentioned that there are pseudo instructions which may correspond to one or more real instructions. So what exactly is this particular one?

(Refer Slide Time: 9:00)



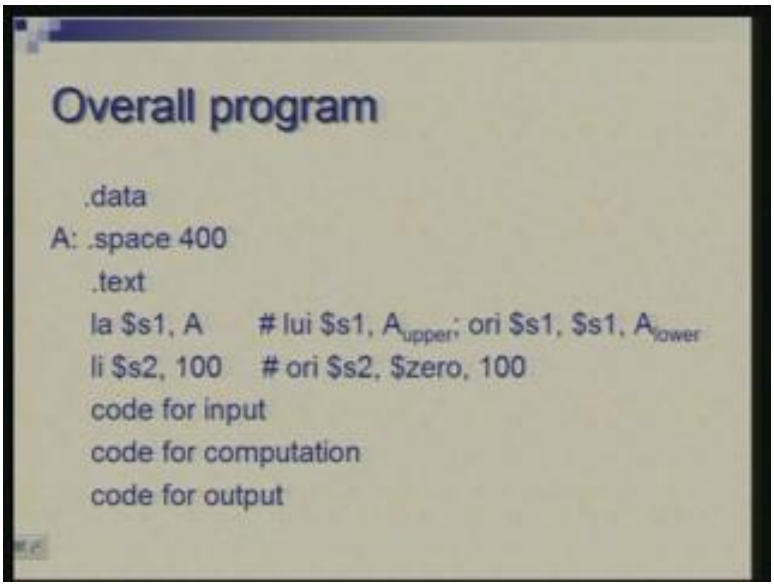
The blt compares two registers and if first one is less than second one it goes to the label specified here. This is equivalent to actually two real instructions: one is slt; slt compares these two but the result as a 0 1 value is stored in at then you can compare at with 0 and then make a branch. So what will happen is if r1 is less than r2 then register at will be set to 1 and it will not be equal to 0 and therefore branch will occur.

So in tutorial we are going to see many more instructions like this and see how they can be implemented using slt and beq or slt and bne. So, various kinds of comparisons can be done. Once you understand that we can simply use blt as long as we have assembler which expands it for us or we can expand it ourselves this works fine.

Now here I have introduced another register at; at stands for assembler temporary. So typically I am expecting that blt will be expanded by the assembler and as you know there is a need for a register to hold some temporary value some intermediate value and by convention one of the registers is actually dedicated for this and in terms of numbers this is register number 1. We have already seen that register number 0 has a special role which is by hardware; you cannot modify the contents of register 0 it always remains 0.

The at or register number 1 is used by assembler it is more by convention and there is nothing which hardware does in this case. If you want to use at for some other purpose you can do so hardware will not restrain you but then you can get mixed up with assemblers usage of it and your own usage of it so the convention is good to be followed so you would typically not use this directly and leave it for assembler to use. That was the core of the program which I talked of.

(Refer Slide Time: 11:39)



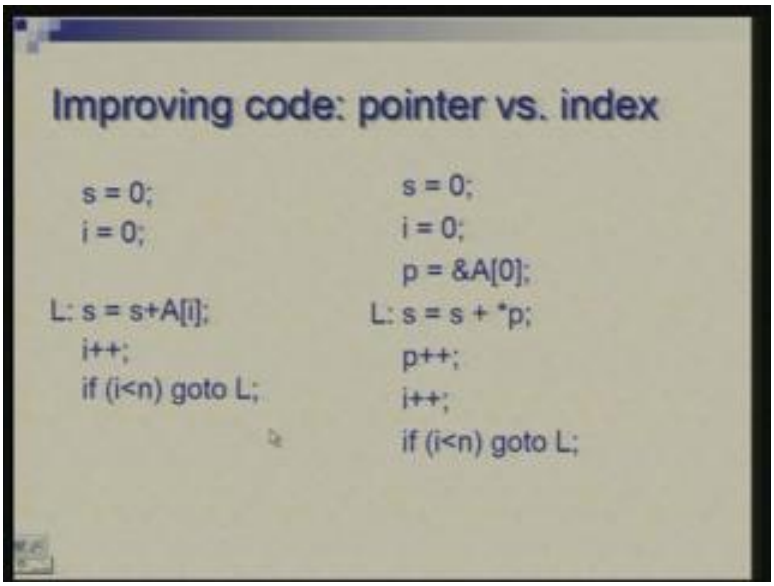
How does the overall program look like?
We have some assembler directives which define the data area and the code area. So dot data is to be used when you want to define some data area. So we are saying that A is a label here, there is another assembler directive dot space and there is space which you are trying to leave of 400 byte. What we are saying is that in the data area leave space for 400 bytes. I am assuming here that n equal to 100 we have array of size 100 then there is dot text which indicates beginning of the code or the program. So you will initialize these two registers which we assume… we assume that s1 contains the starting address of the array so here we are using instruction la which is load address into this register and A is the address we are loading. So, as a symbolic language programmer or assembly language programmer you need not worry about what address actually A is; you are leaving it for the assembler to figure out. If you do want to bother with the data directive you can also specify an address here but here we have chosen not to.

And the assembler will assign some address in some default data area. So A for us is label but internally A is an address and that address is getting loaded into s1. Assuming that A is an address which could be of arbitrary size 32-bits at most this instruction is actually a pseudo instruction which will get split into these two real instructions and this is what we had learnt in the last lecture that loading a large constant into a register requires two steps: the first step loads the upper half and the next step loads the lower half. So the two halves of the address I am denoting by A upper and A lower. A upper is loaded by lui instruction into upper half of s1 and then this or instruction simply superposes the lower half into the right portion of the register.

Our second initialization I wanted was (Refer Slide Time: 14:08) that register s2 contains n so I am saying load immediate s2, 100 and 100 is a small constant so a single instruction can get loaded; you OR 0 and 100 and put in s2. So with this preparation I am ready to go. You need some code to input the data fill up the array for which you are going to perform the sum then there is code for computation this is what we have seen

already and there will be some code for the output. Once you have done the summation you have to output the result. So I am skipping the details of input and output code we are discussing that in tutorials.

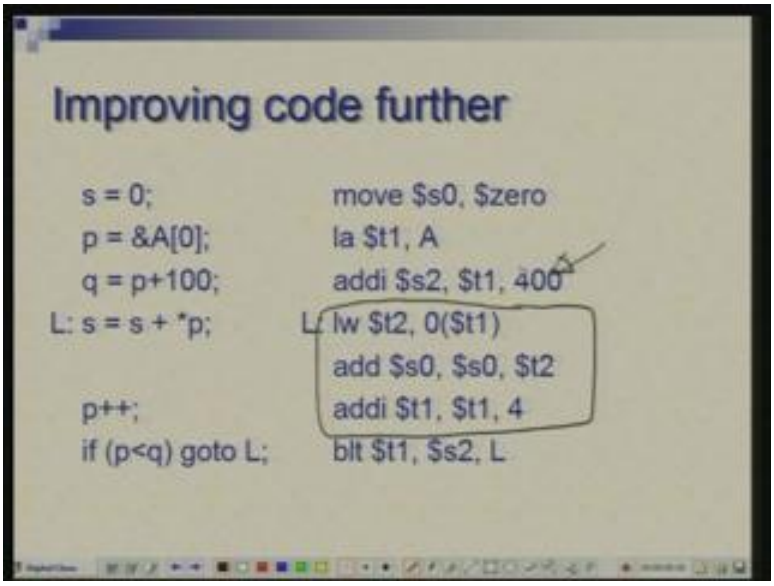
(Refer Slide Time: 14:54)



Now we have written some code which works but can we improve it?
Here we would notice that using pointers or addresses directly rather than index we can have faster code or improved code. So, on the left side here you see the same code which we have and there is some modification made here. I am introducing a pointer p; this is still in high level language I will show you how we do that in assembly language. So p is a pointer which to begin with is pointing to starting address of the array and when you are performing sum instead of indexing the array A we are dereferencing the pointer. So instead of saying s gets s plus A[i] you have s gets s plus star p. Then, in addition to incrementing the index we are incrementing the pointer and the rest is same. These are the two extra statements we have introduced: one is initialization of pointer; updation of pointer, the summation gets replaced by this.

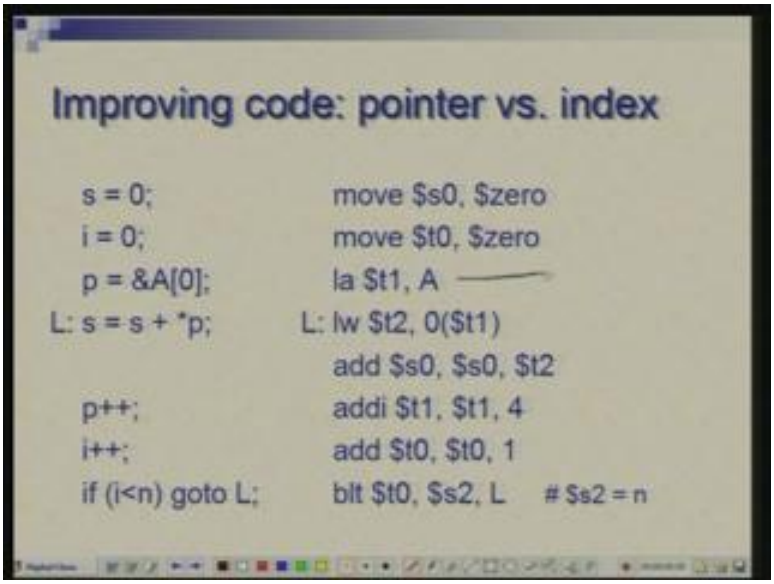
So although it may appear that I am introducing more statements but when you translate this summation into assembly you will find that you can do with lesser number of instructions; that is where the gain is occurring. And initially there is one extra statement but in general we are more worried about making the loop efficient because that is the one which is done several times. In this case it is just hundred times but it could be thousand, it could be million.

(Refer Slide Time: 17:28)



Here is what we have indicated and these are the instructions. Here is main body of the loop where you would see that the number of instructions has reduced (Refer Slide Time: 17:31). This is the initialization of the pointer which I have introduced I wanted to look at this first here.

(Refer Slide Time: 18:17)

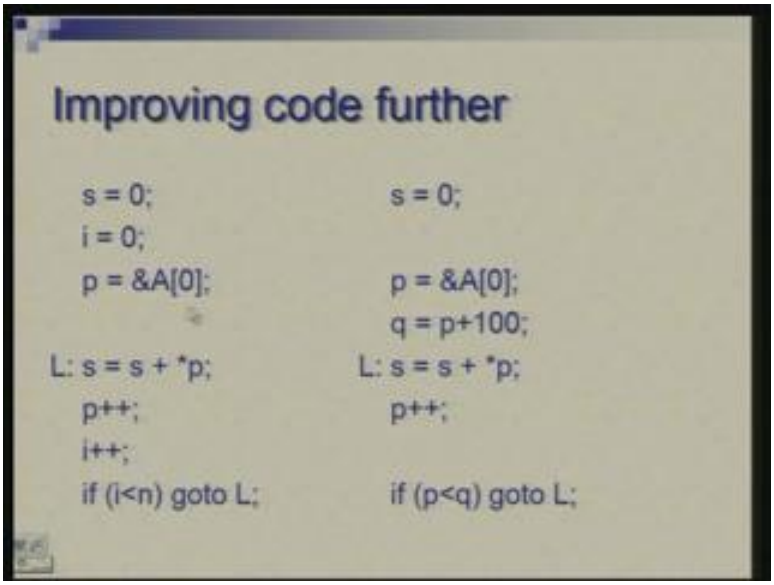


So this is the initialization of the pointer (Refer Slide Time: 18:21) I am saying load address $ t1, A. We have seen how such a statement gets expressed in terms of real instructions and the loop has become simpler, this is simply loading the word and performing the addition because address is already ready. In the previous case we had to

multiply the index by 4, add it to the starting address of the array and then get the data from the memory. So those two initial address preparation steps have been skipped here; you can straight away load the data from the memory and from the addition and that is where the improvement is, this is an extra statement which has come (Refer Slide Time: 19:07) although we have cut down something we have added but what we have cut down is much more and what we have added is not that much but the remaining part is the same.

Is there any question about this?
So having done that we find that there is more opportunity for improving the code. You would notice what is happening here is that we are working with a pointer and an index both are there. We are using index simply to control the termination of the loop. Question is can we use the pointer itself to do that. Instead of maintaining an index which we are not using to access array we have the pointer and at the end of the loop we can check pointer if it has reached the final value. If you do that then i can be eliminated altogether.

(Refer Slide Time: 20:40)



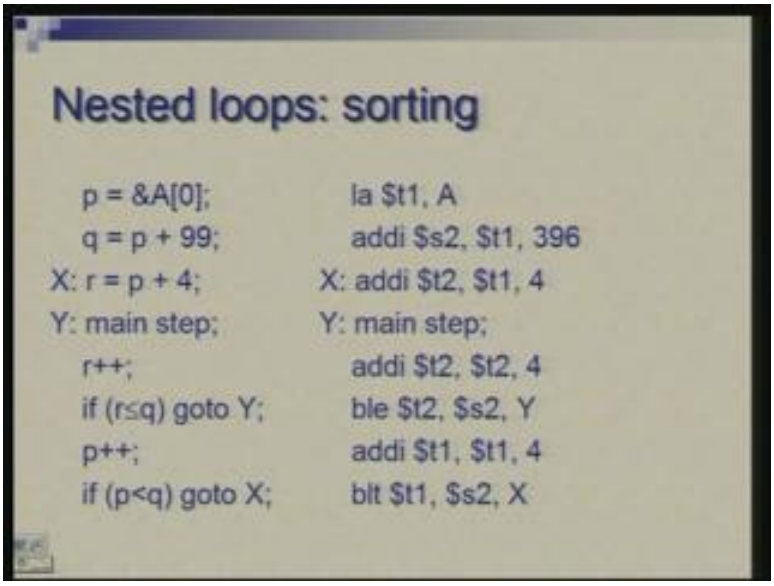
Here is an attempt to improve the code further. You have i equal to 0 initialization and i plus plus and this comparison and these all change (Refer Slide Time: 20:51). First of all we have removed i equal to 0, we have added a statement which computes the final value of the pointer. We say that q is p plus 100 and instead of comparing i with n we can compare p with q so i plus plus is also removed and this comparison changes.

Once again we have introduced a statement here but we have removed there so there is not much of difference and the loop is even shorter. So we go back to assembly and see how this is done. This is a statement, this is the instruction which is preparing the final address the final value of the pointer and we are keeping that in s2. So s2 is equivalent to our q now although in high level language I said p plus 100 but I must be careful to add 400 here. It is 400 which is added here in place of 100 and s2 now contains the starting

address of A plus 400 which is now the final address. So rest is same and the comparison here is between p and q. So this is our final code after having all these improvements done.

Having understood how a loop is programmed let us take up something little more complicated. What if there are two nested loops and you typically require that in sorting.

(Refer Slide Time: 23:18)



So, here is a simple sorting algorithm; this is not definitely an efficient one; what it does is simply iterates over the array, finds out the smallest one, places here at the first position and looks at the remaining, goes over those, finds the next lowest and so on. there are two loops here; once again I have not put for structures I have used comparison then jumps so you would notice that there is one loop here which is beginning at Y and this is contained in another loop which is beginning at X so that is the outer loop.

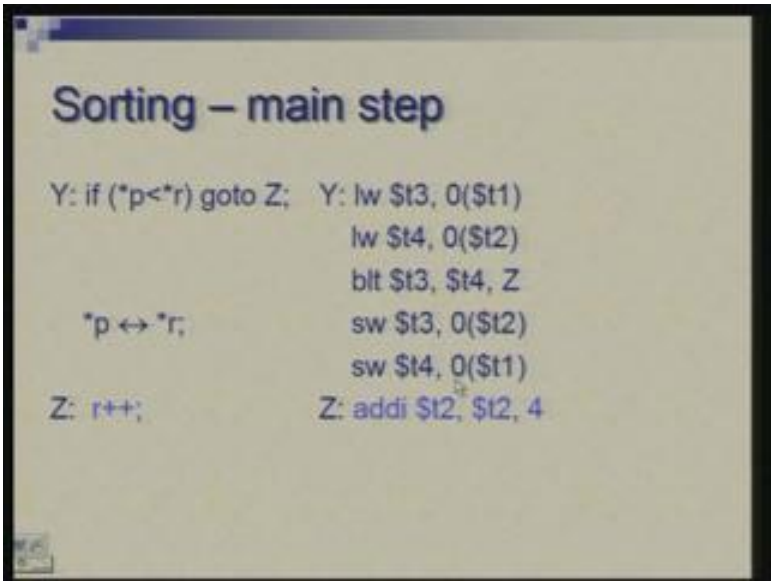
The inner loop makes a pass over the array; the main step which I have not detailed here I will look at it separately so main step basically looks at one element of the array which you are currently scanning and compares it with the element placed in the key position where you are trying to bring it; it compares and if necessary interchange is done. This key position in the first outer loop first iteration of the loop is the first position then becomes the second position and so on. So what I am doing here is with anchor at the first position you scan rest of the array then anchor with the second position scan the whole array and so on. This is a comparison and interchange step.

What you need to focus here what you need to worry about here is how these indices are changing or in this case basically I am working with pointers. So p is one pointer. to begin with it is pointing to initial position of the array, q is again pointing to the end value so I am doing p plus 99 so it is not pointing to the final value at which you will terminate it is pointing to the last element. r would be the second pointer and q will not

change in the process; it is p and r which will change; r will change in the inner loop and p will change in the inner loop so r gets initialized to p plus I think this should have been p plus 1; in assembly I will say p plus 4 but here please correct it to p plus 1.

Now r is initialized to this and in the inner loop it is updated. We are basically comparing elements pointed by p and r and if necessary interchanging and this inne r loop simply does that; p is not changed, r is updated, comparison is done and you go through this. Once you come out of this then you update p and repeat this process (Refer Slide Time: 26:37). The r starts with a position which is next to that. So once again it will translate very simply, the pointer initialization is load address so t1 is carrying our p value, the next statement prepares q for comparison purpose, s2 contains t1 plus 396, t2 is initialized to t1 plus 4, t2 is r, then update r, do a less than equal to comparison here between r; t2 is r, s2 is q, update p which is in t1, do a less comparison t1 and s2 which is p and q. This is how the overall flow of control is and now what remains is to elaborate on the main step.

(Refer Slide Time: 27:44)



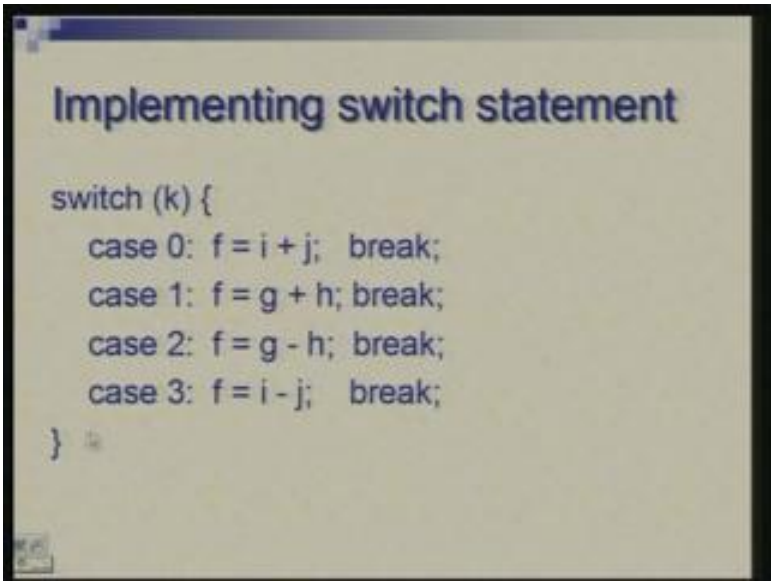
Basically we are comparing limits pointed by p and r. If p element or element pointed by p is less than element pointed by r we skip this interchange step; this is interchange (Refer Slide Time: 28:01) I am just showing it in abstract sense that star p and star r are interchanged this would require three assignments going through temporary and so on but I am not going to go into those details but how we do interchange in assembly that we had seen in the earlier lecture.

Therefore, all you need to do is load the values pointed by p and r into t3 and t4, do this comparison less than comparison I am once again using blt statement which is pseudo instruction; if I need to interchange all I need to do is simply store the values in a cross fashion so t3 and t4 the order was t1 and t2 and here it becomes t2 and t1 so you just store them in an interchanged manner. And, Z is a label which I am just putting to the

statement which is following it. So if you go back after the main step this was r plus plus statement or add immediate t2 t4 so basically that statement it is not that you are putting it once more but I am just indicating that Z is a label which is there just to make sure that you come out of this part.

So once you understand how comparison is done and how branch and jump instructions are used the loops can be implemented in a very straightforward manner. Another control structure which you often encounter is that you have some value which could lie in certain range and for different values you need to do different actions. Unlike Boolean comparison where you have true and false outcome here you have some expression which you evaluate and the result falls in certain range.

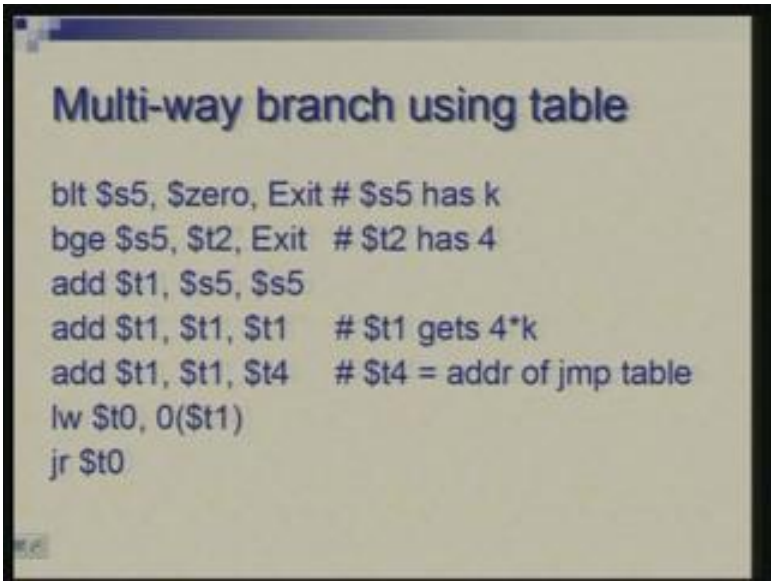
(Refer Slide Time: 30:24)



As an example suppose you have a value k which can be either 0 1 2 or 3 and small pieces of code you have for each case you will capture it by switch statement and after each you want to break that means after performing f equal to i plus j you dont want to go through all of them you want to skip and come to the end you put a break statement. Of course the last one is not necessary but just to make it look uniform I have put it there.

So how do we do that?
One option of course is that you can make repeated comparisons. You compare k with 0 check if it is equal to 0 or not do this action then compare with 1, compare with 2, compare with 3 with the range just being 4 it is not all that difficult but if the switch statement had lots of cases then this will become very tedious and messy so there is a more elegant way of doing it by doing a multi-way branch by a single instruction. A new instruction which is jr jump on register content which will be used here.

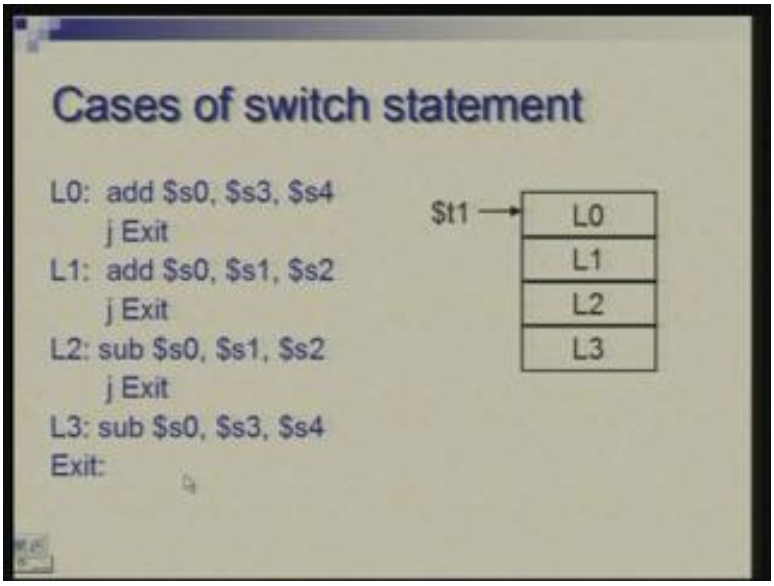
(Refer Slide Time: 31:22)



So we will do this with the help of a table. What we have is that in a table we have all the starting addresses of different cases and then we simply look at the value of k use it to pick up the right address from the table and jump to that. So let us see how it is done. Initially I am making sure that the value k falls within the range 0 to 3 if it is outside then I want to skip because my table will not have entries corresponding to it. So, assuming that s5 contains k first it is compared with 0, if k is less than 0 then I skip; second comparison is made against 4 so if this is greater than or equal to 4 again I exit so label exit would be somewhere at the end here so once that is done then I need to prepare an index for the table from where I am going to pick up a destination address.

So basically I need to obtain four times k; each address in the table is going to occupy 4 bytes or one word so here you will see that there is a different way of performing a multiplication by 4. I am simply doubling the number twice and that gives you four times its value. So add t1 to itself the result goes to t1. there should be one more add statement like this (Refer Slide Time: 33:37): I am sorry there is…… you do it twice, this is doubling t1 no I am sorry, yeah okay okay we are starting with s5, first we have doubled s5 got into t1 then double it further got into t1 again so we have 4k then t4 is assumed to have address of the jump table the starting address the way we prepared address for the array same thing is done so starting address is added and now the final address is in t1. So this content of memory location address by this are loaded into t0 and then I say jr t0. So this exercise is to jump to an address which is contained in a table and which address from the table we pick up is determined by k.

(Refer Slide Time: 34:34)



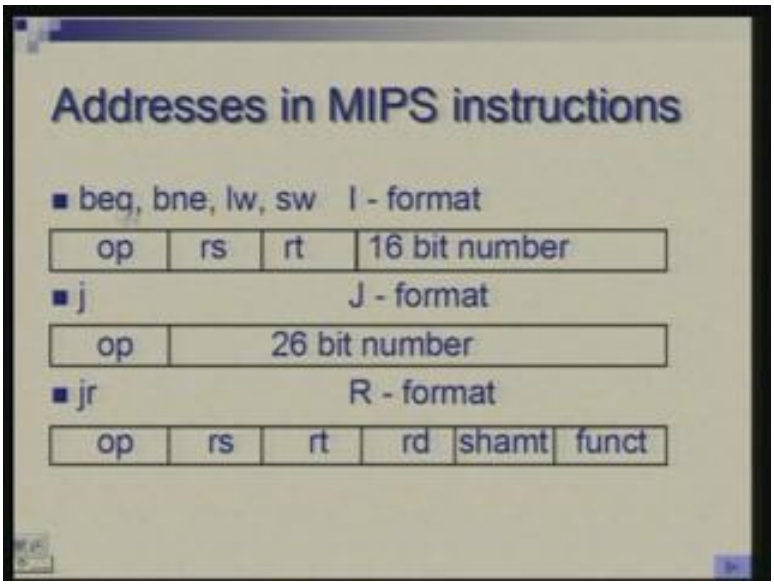
Hence, the four cases we had are now listed here. They are at label L0 L1 L2 L3 and the table is shown here L0 L1 L2 L3. These are the four addresses kept in this table and register t1 finally is made to point to the appropriate address in that table and we are jumping to that. So the statement we had in the first case (Refer Slide Time: 35:11) now we will recall this. We have i plus j in the first case so there is an addition here and another addition there and then two subtractions. You will find a straightforward code doing that. s3 and s4 are added and put in s0 this is the other addition, subtraction and another subtraction. So whatever the code is whatever is the body of the case can be put here; it could be one statement or it could be more statements and finally you have jumped to the exit which is corresponding to the break we have.

Now this looks nice for a simple case but problems could often be more complicated. If you are talking of values which are not contiguous; here we had 0 1 2 3 all four values were relevant but if you have non-contiguous values then the table could be sparse you would waste the space and if it is too sparse it may still be better to do repeated comparisons but if the range is compact that means you are interested in almost all the values of the range then the table approach will be fine.

In very specific cases when these four labels are let us say equally spaced if they are uniformly spaced or they can be…… or these bodies or cases can be padded up with blanks to make them uniformly spaced then you do not really need that table. So, suppose these addresses are 1000 let us say 1020, 1040 and 1060 they are uniformly spaced. Then instead of table what I need to do is I take the starting address 1000 plus twenty times k so I need to simply compute this and then do jr with that. Instead of looking up to these four addresses in the table this is fine this is a general approach if the addresses are arbitrary. But if they are uniformly spaced like this then I can simply compute the address by taking a base address plus some constant time k so that sometimes that is much more efficient.

Now let us get back to this range of instructions we have talked of and see how address is being actually specified in these.

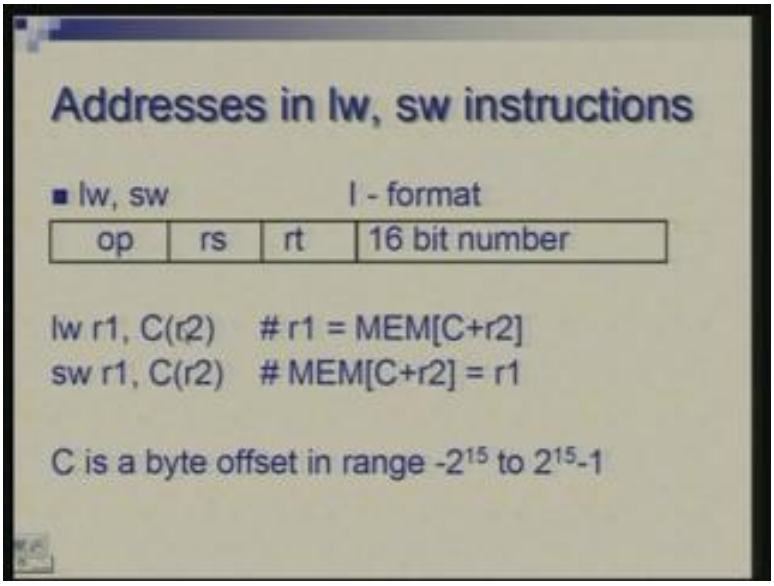
(Refer Slide Time: 38:42)



We have so far seen branch instructions eq ne comparison; I will not look at blt bge and so on because they are ultimately built using beq bne. Then we have load and store instruction which all these used I-format where a 16-bit number was specified and that was used in computing the address; j instruction had different format J-format where there is a 26-bit number specifying the address and today we have seen a new instruction jr which is actually in R-format. So although R-format provides fields for three registers we require only one here others are left unused but the format is R and the remaining these three fields are unused.

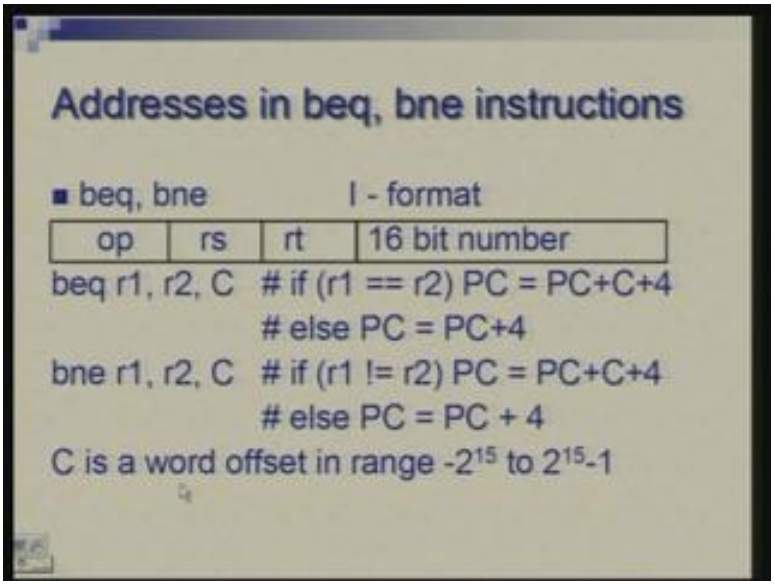
Therefore now in these three cases how do we specify how do we really get the address given an instruction and we will even look at these two and lw sw separately. So first let us begin with load and store.

(Refer Slide Time: 39:49)



So, imagine an instruction of this type load word r1, C(r, 2) where C is the constant part and r2 is the register this is also called index register or base register. So the exact effect of this is that the memory location which is being addressed is obtained by [C plus r2] these two are added and the thing to be noted here is that C is an offset which could be positive or negative and as a 16-bit number the value of this offset lies in this range 2 power minus 15 to 2 power 15 minus 1 this is in 2s complement form; if you are familiar with 2s complement notation you will immediately recognize this but if you are not we are going to talk of this representation detail later on. So, at the moment you can just notice that the offset could be positive or negative in certain range and the address which is being specified here is a byte address.

(Refer Slide Time: 41:15)

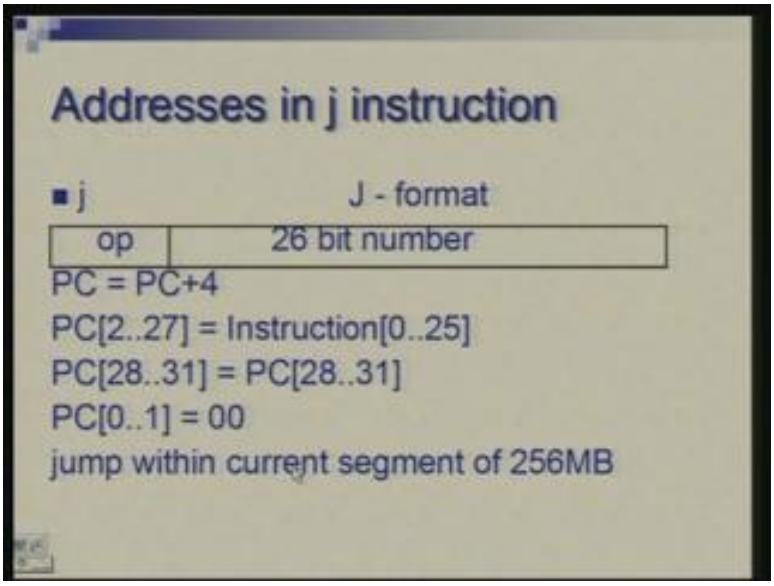


In beq, bne we will actually have we are worried about word address whereas data could be in terms of bytes or words or double word or half word all those varieties may exist the instruction in this architecture are all one word instructions. Later on we are going to see processors where instructions could also be of varying size but here we have simplicity that all instructions are 4 bytes or one word. Therefore the instructions are always located at aligned boundaries; the instruction addresses are always multiples of 4 and that fact is made use of, so the address which is provided here is actually an offset again positive or negative with respect to the current position in the program.

In a program if you are jumping four instructions ahead then you will give a positive offset corresponding to that; if you are jumping back like in a loop we had you will have a negative offset. When you are using assembly language you simply write a label and the label of destination could be after the current instruction it could be before the current instruction; assembler will automatically generate a corresponding offset either a positive or negative. The offset here is in terms of word it is not a byte offset it is a word offset because we know that addresses for instructions are always multiples of 4.

So the meaning of this instruction exactly is shown here beq r1, r2, C compares r1, r2 and PC which is the program counter that is the register specifying address of the current instruction it becomes PC plus C but strictly speaking notice that there is a plus 4 also. So the offset is not with respect to the current instruction it is with respect to the following instruction. There is a subtle reason which is related to the hardware implementation of this that why we are keeping PC plus C plus 4 and why not simply PC plus C. It is for some hardware convenience which we will see later. For the moment you can just notice that depending upon the outcome of comparison the next value is either PC plus 4 or PC plus C plus 4; it should actually be PC plus 4 C plus 4 because C is a word offset. The next instruction follows similarly.

(Refer Slide Time: 44:03)

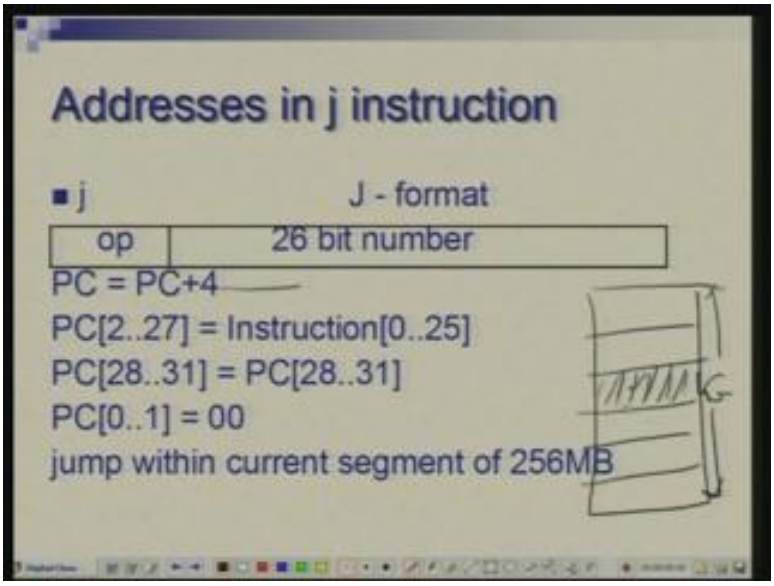


What happens in jump instruction?
In jump instruction we are specifying a 26-bit number as destination address but we need a full-fledged 32-bit address finally before we can proceed further. So the effect of this instruction is as follows that once again PC is incremented to PC plus 4 although we are not going to the next instruction but this happens automatically this happens sort of….. in terms of hardware PC gets PC plus 4 happens without even bothering whether it is a jump instruction or a branch instruction or something else and after that we see how rest of the address is computed. So these 26 bits of the instruction which I am saying instruction 0 dot dot 25 so these are the twenty six LSBs of the instruction corresponding to this field they are placed in position 2 to 27 of the PC; bits 28 to 31 of PC are unchanged and bits 0 to 1 are 0 and they remain 0.

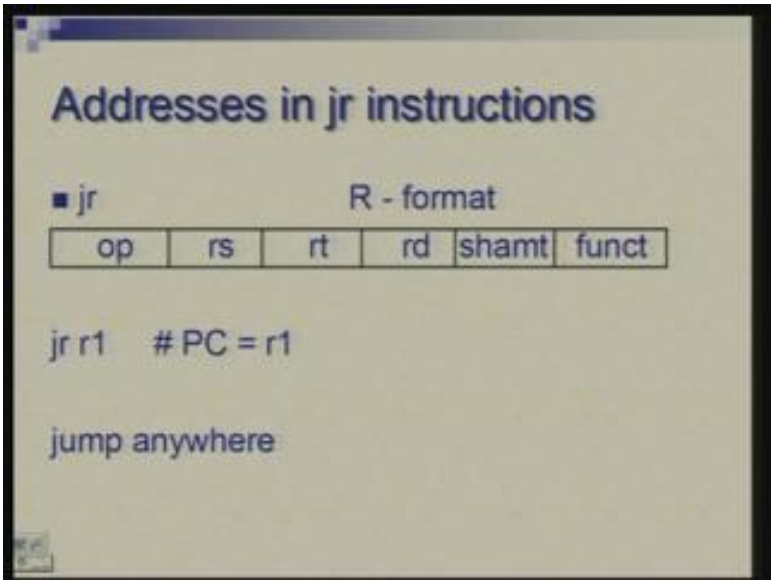
Now what it means is that using j instruction you cannot jump anywhere in the memory space; you can specify range which is defined by 26 bits and this 26-bit number is taken as a word address. So effectively in terms of bytes it is effectively 28 bits. So your range of jump is within the current segment of size given by 2 raised to the power 28 or 256 MB. So, if you imagine that the memory is divided into segments with 32-bits you can get 4GB of memory 2 raised to the power 32 is 4GB imagine that 4GB space is divided into segments of size 256 MB. So your jump is occurring within the current segment. Let me illustrate that.

Suppose this is 4GB (Refer Slide Time: 46:59) and each of the segments is 256 MB and if your current instruction is here you can jump anywhere in this region. of course there will be an exception case if the jump statement was the last statement of a segment then jump will occur in the next segment because of this initial value of PC. PC is initially incremented by 4 and then you are figuring out where to jump. So typically you will be somewhere within the segment and jump will occur within the segment. If you are at the last point at the last word of the segment then the jump will occur in the next segment.

(Refer Slide Time: 47:43)

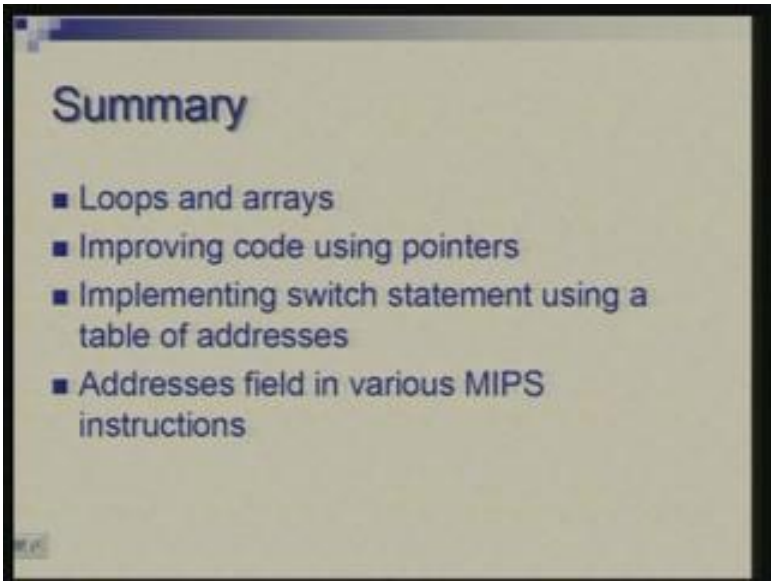


(Refer Slide Time: 48:09)



Lastly we will look at jr which is very simple you are specifying a register which is a 32-bit register so all that you are doing is transferring the contents of that register into PC. So here you are specifying a full-fledged 32-bit address and this can be used to jump anywhere. So most of the time j instruction suffices when you are simply jumping to a constant address but if you want to jump to a far off segment for some reason you will have to use jr again. We have seen jr is used for a different purpose; we have seen how jr is used for a switch statement but jr can also be used for carrying out jump when the destination is far off.

(Refer Slide Time: 49:05)



Finally let me summarize about what we have learned today. I began with examples of some loops, how we can use simple compare and branch statements to carry out simple loops, nested loops. We have seen that when arrays are being accessed sequentially in a loop there are opportunities for improving the code. You can do more computation before the loop begins so that the loop is a tight loop, it is an efficient loop, the body is more compact.

Now what is the importance of this improvement?
As I mentioned earlier you are like likely not to program in assembly language in practice. So what is the point in bothering about these small nitty-gritty improvements these are important from the compiler point of view. So the way we try to improve the code that understanding is required for compiler writers because they have to generate code which is efficient and these transformations, these improvements should be kept in mind by a code generator of the compiler so that efficient code is generated.

So given the kind of example we took in the beginning actually compiler can automatically generate the final improved code from the inefficient program we wrote initially. That means you might write using index but compiler can always convert it to the final form. These are systematic transformations which can be used to do that. We also saw how we can implement a switch statement using a table and jr instruction and finally we have seen in a very subtle manner that how various instructions address the memory. There are different ways in which address is computed in different instructions and we have gone through those details. I will stop at this point.