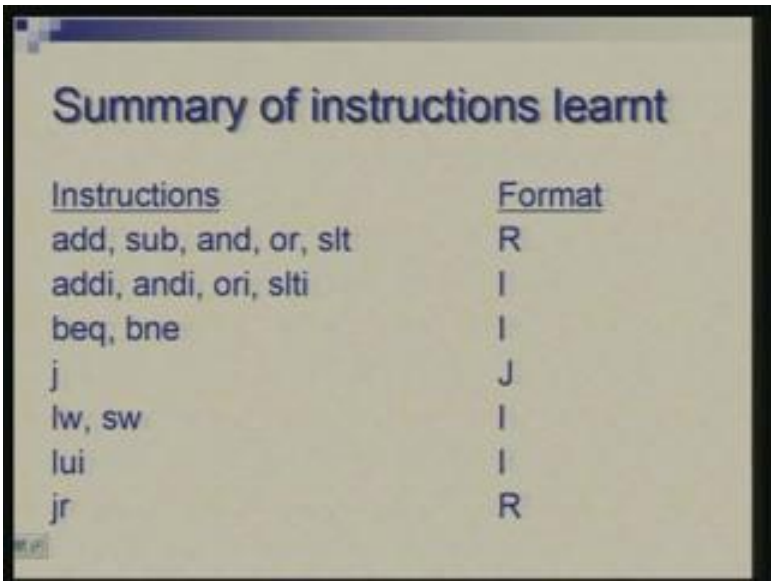
Computer Architecture
Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 5
Instruction Set Architecture - 3

Today we are going to look at a very important programming language abstraction namely function or procedure. This is very important in the sense that you can build program in a hierarchical manner, in a top down or bottom up fashion and without this constructing large program would be impossible. The set of instructions we have already learnt is summarized here.

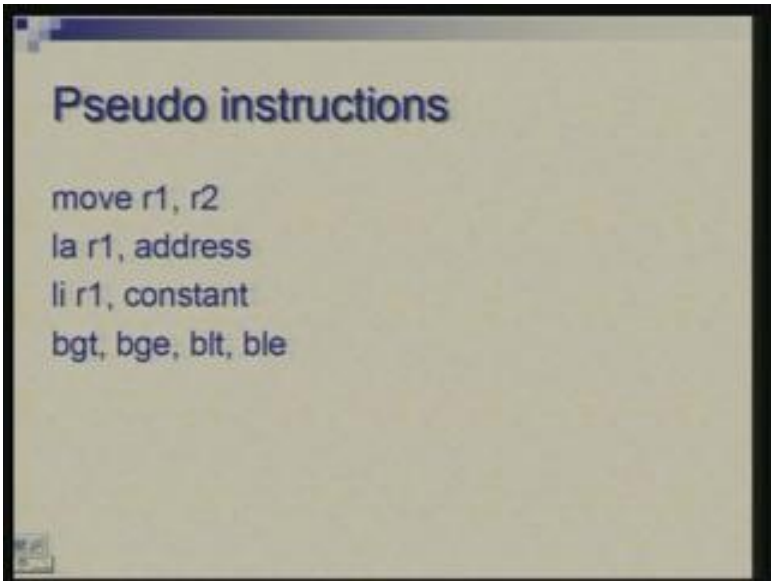
(Refer Slide Time: 01:17)



This includes arithmetic instructions and logical instructions which you see in the first two rows when both the operands are registers or one is a register or one is in the form of a constant, comparison and branch, unconditional jump, load store, load upper immediate and jump with register containing the address. So we use this as a means to jump to an arbitrary location and also as a mechanism to carry out multiway branch.

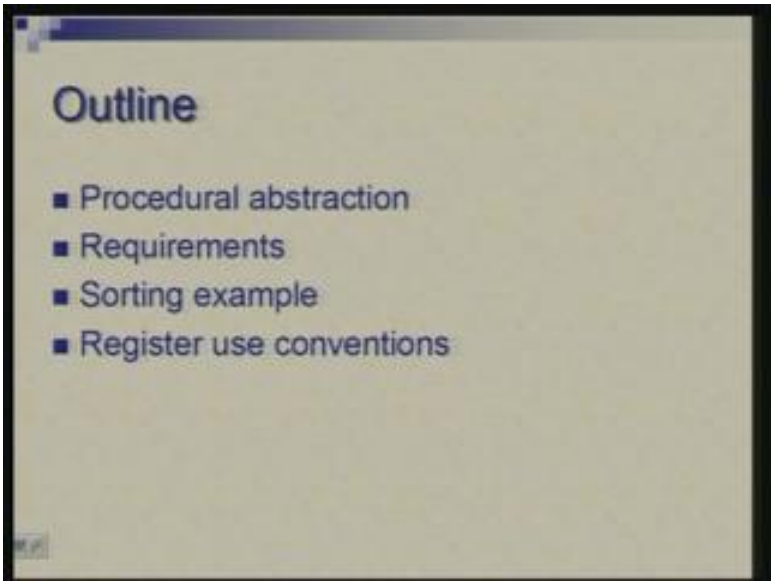
As you went along we also felt the need of some pseudo instructions and some of the pseudo instructions we have defined are shown here. Move is something which is frequently used, simply moving a data from one register to other register. Essentially a copy of value is made; load address, load immediate which loads a constant into a register and some variations of branch. These instructions are implemented by one or more real instructions.

(Refer Slide Time: 02:23)



Some of these we have seen how they get expanded, some of these we will discuss in tutorials.

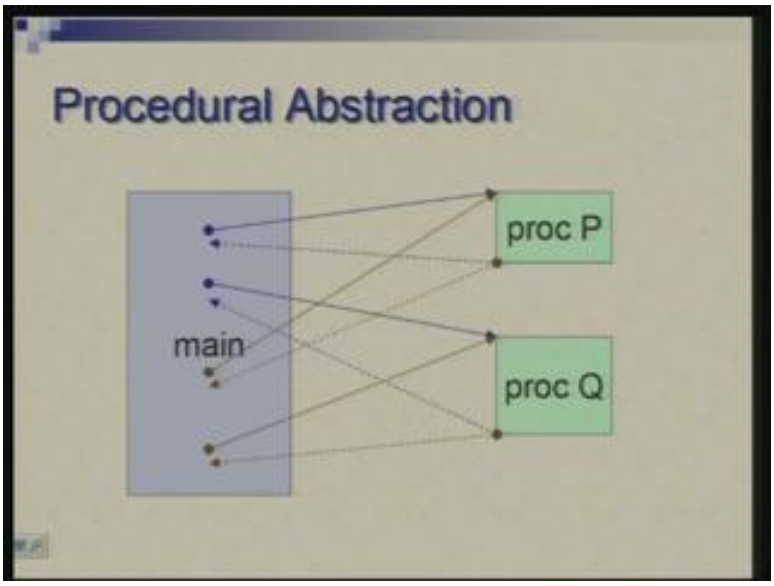
(Refer Slide Time: 02:39)



So today will talk about what actually is involved in procedural abstraction. To implement a procedure what all we require, what activities or what function the instruction has to support. We will illustrate all this with an example. I will continue with that sorting example which we had built up in the last class and try to do in the form of functions there. And finally we will find that there are registers which are although

general purpose but there are certain conventions which have to be followed in order to develop a program smoothly.

(Refer Slide Time: 03:18)



So what actually we mean by procedural abstraction?
Essentially procedural abstraction means that there is a piece of code which you can write once and use it one or more times thinking of that as a single statement. It could be an arbitrary piece of code which does the computation which has a well-defined well-identified meaning and this becomes your basic operation either a single operation or a single statement which can be used with the same ease and convenience as you do for the basic operations.

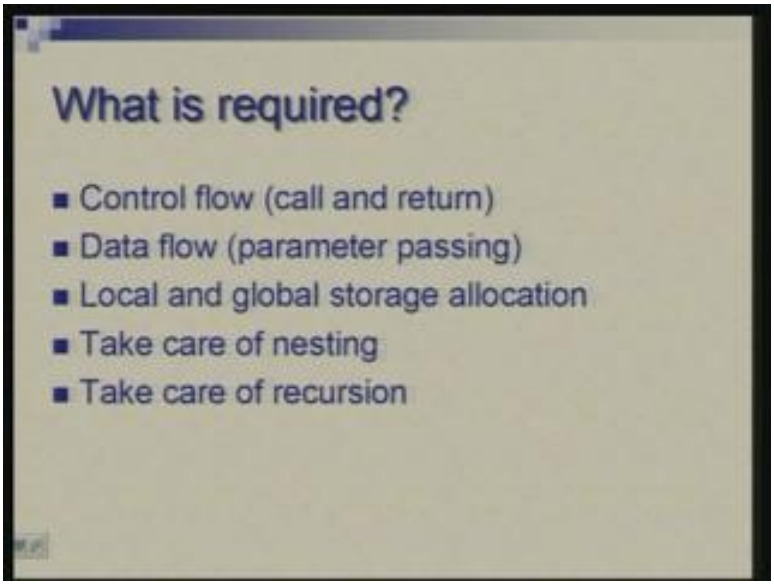
So here it shows an example that there is a main program, there are two procedures P and Q and P is being called here, this computation is performed then there is a return which is made. There is another point here where Q is being called and then there is a return. Similarly P and Q are being called again and there is a return. So the number of times we use could be arbitrary and to implement this we require several things. Several things have to be kept in mind. First of all there is control linkage; you have to worry about flow of control. That means from main program you should be able to call that means transfer the control in such a manner that when the procedure ends the control returns back to the point where you made a call. So there is a linkage which is required. It is not simply a matter of using a jump statement a one way and the jump statement another way. You need to know where you came from so that return can be appropriately made because call may occur from several different points and return has to be made accordingly so that is a key part here.

Secondly, every time you invoke a procedure it may work on different set of data. So there is a concept of parameters of procedures and when a procedure is called some data flows into the procedure parameters are passed and when computation is over the result

flow back to the calling program so the parameters which carry values into it and those which carry value back to the caller.

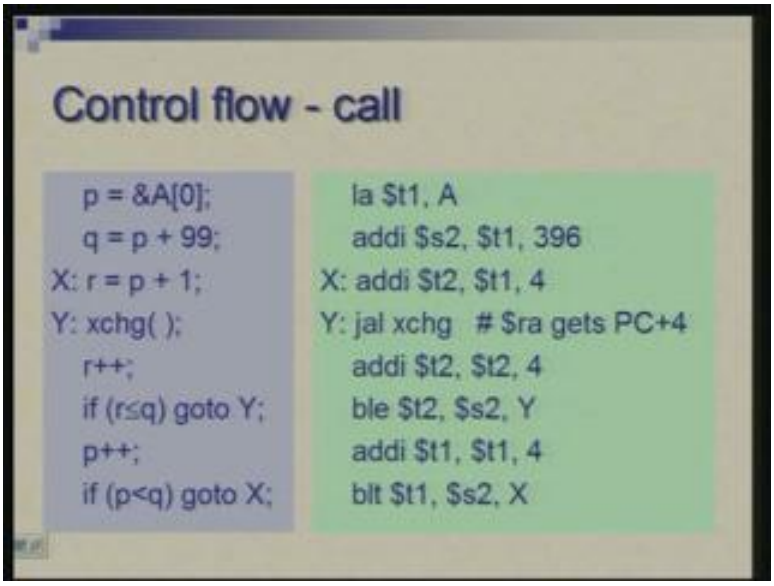
Apart from the parameters which are decided by the caller or the results which are consumed by the caller there are also often local storage declarations which you may have inside a procedure or a function. And if you call a function multiple times then it is considered to be a fresh allocation of storage so how do we handle that; at the same time a procedure may make an access to the data which is defined outside so there could be global data, there could be local data so both needs to be accessible.

(Refer Slide Time: 6:21)



What I depicted in the previous picture was a very simple case where main program and there is a function or procedure which is called but there could be nesting. In the previous case P could have called Q and Q could be called by the main Q could also be called by P and the matter gets complicated further if there is a recursion, there could be a direct recursion and indirect recursion that means P could have called itself or P calls Q and Q calls P so there could be direct or indirect recursion and all these issues of control flow, data flow, organizing local and global storage become more complex when you have to take into account the need for nesting and need for recursion.

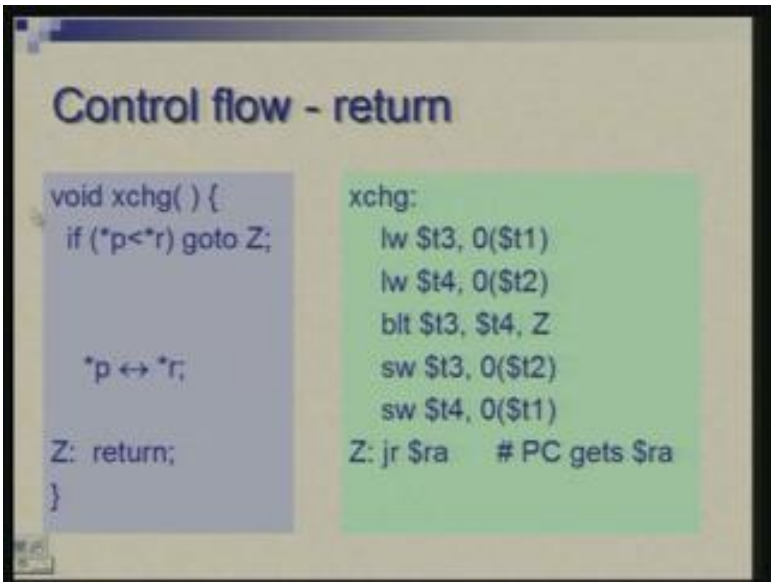
(Refer Slide Time: 07:17)



So let us take the first thing first, how do you organize the flow of control. I am taking here the same example of a very simple minded sorting program where we simply had double loop and the main operation inside the loop was comparison and interchange so that comparison and interchange supposed to be defined as a function or a procedure which I am calling as xchg exchange and now it is basically this exchange with some other overhead which is enveloped in two nested loops.

Now what happens at the assembly level, rest of it is same the only change is here. Earlier what I have done here was that although I have not shown in the same screen but this comparison and exchange was basically a set of instructions some seven eight instructions which were placed here. Now we do not place instructions here we put a call instruction or which is called jal jump and link jal stands for jump and link and I am treating xchg as a label. So somewhere there is an instruction with this xchg as a label attached (Refer Slide Time: 8:43) and the effect of jal instruction is to transfer the control much in the same way as jump instruction does but it does one more additional thing it saves the current address of instruction into a special register; well, it is special in terms of functionality but it is one of those thirty two registers which we symbolically denote by ra which stands for return address. So the effect of this is that $ra or the return address register gets the value of PC plus 4. So PC is pertaining to this instruction, ra will now contain address of the instruction which is following. This is the point where you have to return after completing the procedure. So this address would be ready in ra register and when you are done with the procedure you can use that address and link back. So rest of it is not changed and here you can see how call actually has been established.

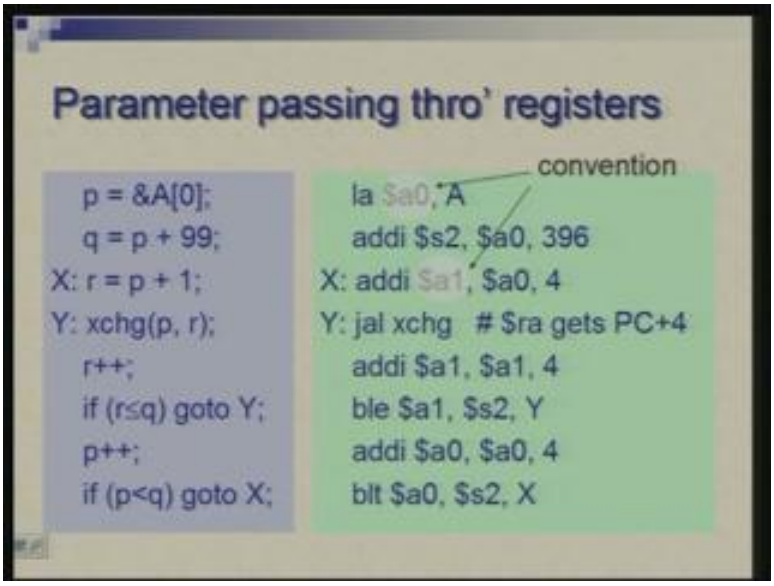
(Refer Slide Time: 9:55)



Now let us see what happens at the other end. What I call as main step earlier I have encapsulated in the form of a function call. We are not returning any values so our return value is void there is no parameter being passed, it is simply looking at global values and doing something with it. So I have just added a return statement here. And in terms of MIPS language this is the same piece of code I have put this as a label (Refer Slide Time: 10:29) and at the end I say jr $ra it is same jr instruction which takes contents of a register and uses that as the destination or target address.

Since jl had stored return address in this register you can simply do jr and get back. These are the two instructions which actually provide control flow and linkage of caller and the callee.

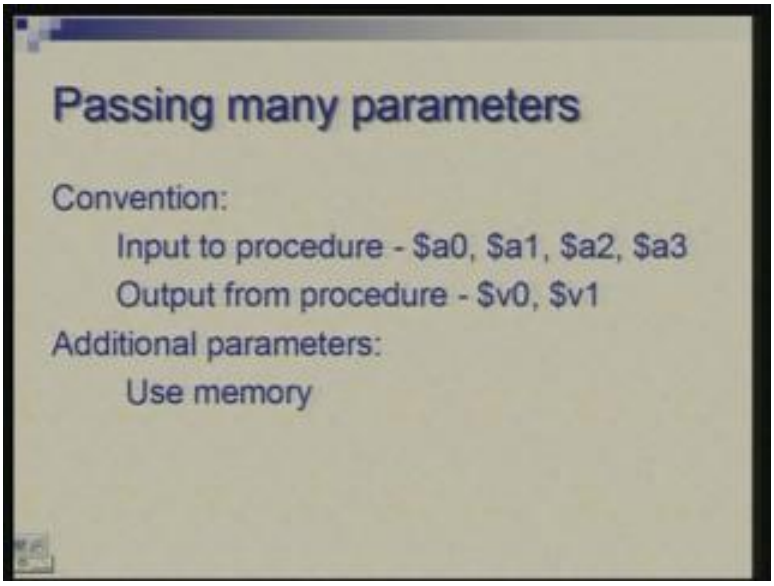
(Refer Slide Time: 11:08)



Now let us look at the question of passing parameters. We have seen how control is let. Now we see how to take care of data. Now, in this case what I have done is from the previous picture the only change is that I am making p and r as arguments, the parameters. So the function need not look at the values which were with the main but it is explicitly passed on p and r the two pointers.

Now the way it is done, the simplest method is what is shown here is use some specific registers which are designated for passing parameters. So the values which have to go into the procedure are loaded into these specific registers. In this case you can see dollar a0 and dollar a1 these are the two registers which are part of a set of register from where I can convey the parameters. So all I have done is I have changed the register which I was using there which I was using arbitrarily but now I am just making sure that p is passed in a0 and r is passed in a1 and rest of it rest of the program has been accordingly modified. So there is no extra statement it is just that I am careful about which registers I have to use for this purpose and typically I will avoid using them for something else.

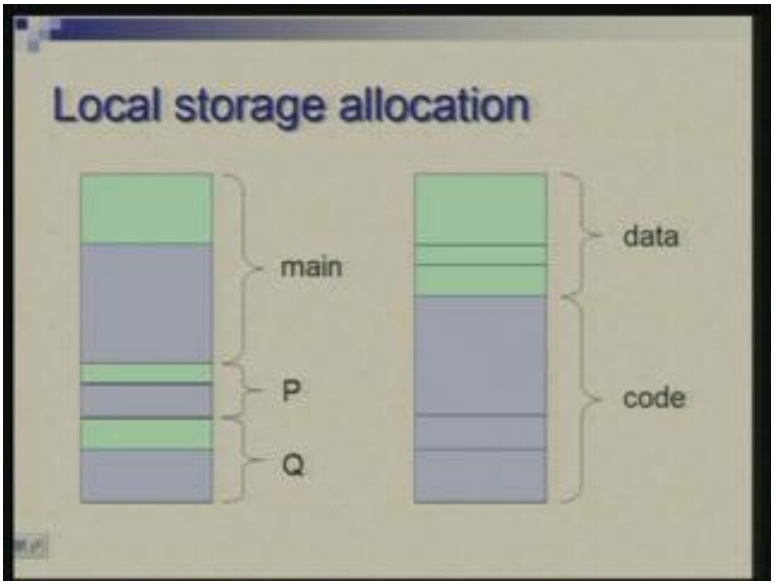
(Refer Slide Time: 12:54)



What happens if you have large number of parameters?
The convention is that if you have up to four parameters there are four registers designated for it which are labeled as a0, a1, a2, a3 and actually that will cover lots of common cases. Similarly, the values being returned or the output from the procedure or function is through two registers v0 and v1 so once again this would suffice for many common cases.

What happens when the number of values going in or coming out is more than 4 and 2 respectively?
In such a case we have to resort to memory. So any additional parameters which you have you can place them in specific memory locations and the function is expected to load them from there, work with them and the results can be partly returned through registers if they are more they can be returned to memory locations. That is a simple extension of what we have seen through registers.

(Refer Slide Time: 14:05)

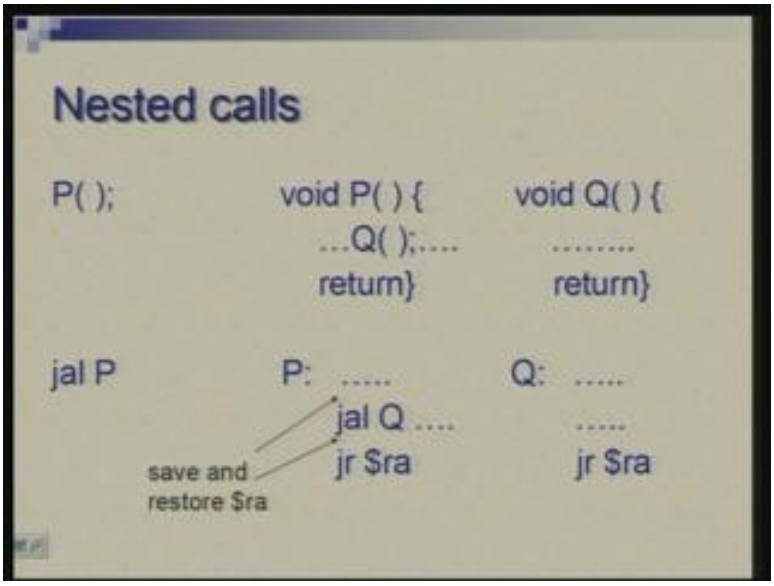


The next issue was that of defining local storage. Suppose within the procedure, within the function, there is an array declaration or there are structures which are defined so what do you do; you can organize each of these functions or procedures with its own storage area that is the data area and the area where its code is kept.

So, for example, if you look at this picture this (Refer Slide Time: 14:39) is the main, this green one is the data area, this is the code area; then there is P green is the data area code area for Q data and code. I will just place them one after another in contiguous location. It is not necessary that data has to be before code but any convention you can follow. All that I am trying to say here is that each of these functions or procedures has its data and code together.

Another alternative could be that you have an overall data area where you keep data of each function or procedure and then there is overall code area where you have code part of each function or procedure. Any of the convention can be followed and of course a compiler would follow always a specific convention and produce the code accordingly. So it will process all the procedures, look at their code part, look at their data part and do the storage assignment accordingly.

(Refer Slide Time: 15:49)

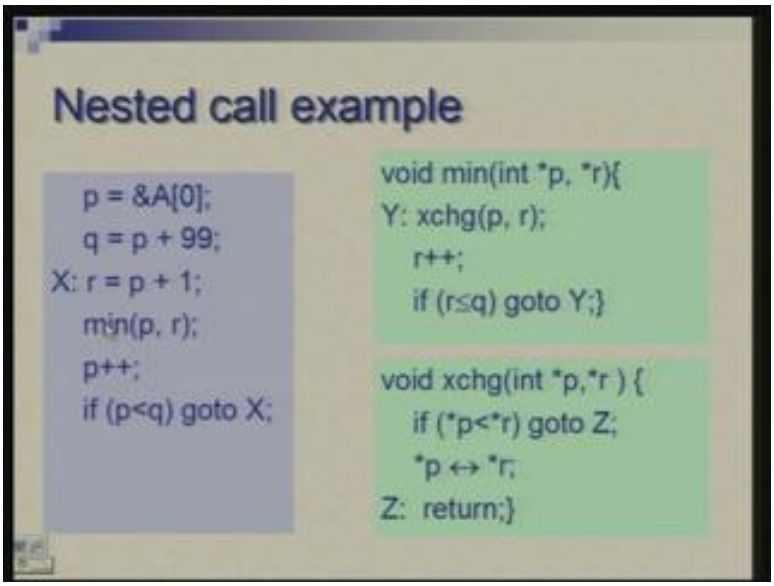


Now let us move to the case of nested calls that means a function can call another function. I am showing here an example where this is the main (Refer Slide Time: 16:07) at some point you are calling function P. this is P and there is some computation here somewhere you are calling Q and this is Q so this is the return point of Q, this is the return point of P how will they appear in assembly you have jal P for this call so that will bring the control here and at some point you say jal Q the control gets transferred here (Refer Slide Time: 16:45) here you expect that jr $ra brings you back here go further you expect that jr $ra brings you back here that is how you want the control to be linked.

But now what will happen?
When first call occurs the address of the instruction here which is after this jal gets stored in ra. But when you reach this point and issue another jal instruction then the old address which was there in ra gets replaced by the address of the instruction following this. So there is no way you can return to the main. And here this call will occur correctly, this return will occur correctly but after this point again when you say return, the program will get back to this point so it might get into an endless loop here and the solution is very simple that is you must take the precaution that the return address which was made available here should be saved somewhere and before you do jr it should be restored.

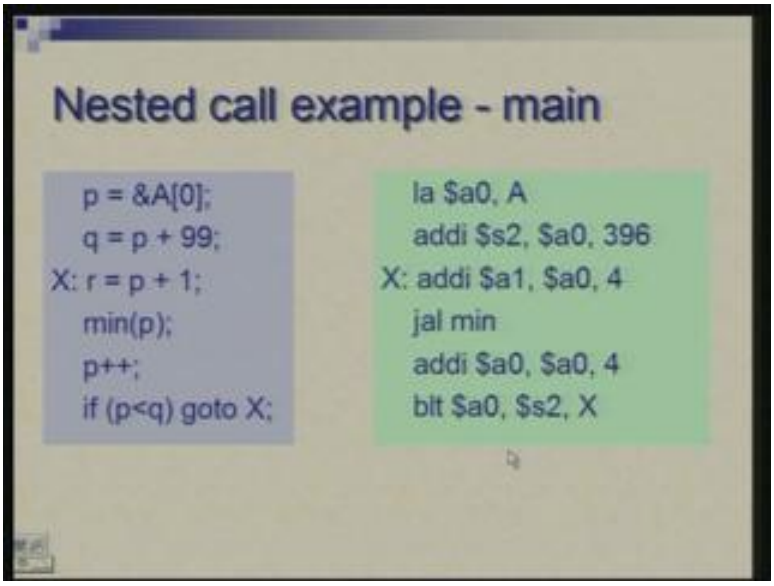
Now, this procedure (Refer Slide Time: 18:03) would have some local storage to that you can add one more location where it preserves its return address. So now in between many calls can occur, Q can call something else and so on but as long as P takes care of where it has to return the job would be done you will not make a mistake. So Q can take care of saving its own so what you could do is the first thing when you enter the procedure you save the contents of ra into some memory location and just before call the last thing you do before returning sorry not before call just before return the last thing you do is from that memory location load into ra. Once you have done that in between ra is free, you can make any calls, you may or may not make a call and there could be any nesting of calls.

(Refer Slide Time: 19:17)



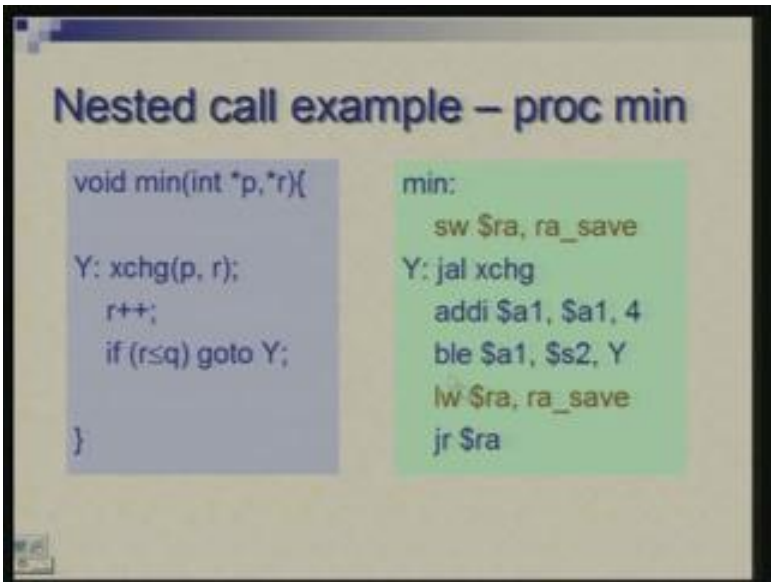
Here is an example. What I have done is that in that sorting case, sorting problem, the inner loop has been redefined as another function. We will recall that it was trying to find a minimum and put it at the right place a minimum of certain number of elements. So it is being passed as two parameters p which is the pointer to place where minimum value has to be kept and r is the pointer to the area in array from where you need to start scanning. So r is made p plus 1 and then you make a call. So, what min will do is it will go through that loop, scan up the array from r onwards and bring the minimum value back to location pointed by p. So, rest is same and now we are left with a single loop here. There is single loop call to min, min has that inner loop basically, it performs that exchange condition conditional exchange compare and exchange, update r and keep repeating. This is as it is (Refer Slide Time: 20:42) this I have not changed, this is the inner most activity which you do, compare and exchange. Let us see now how these will be done, each of these.

(Refer Slide Time: 20:53)



So the main body we have simply jal to min as a replacement for this and we are making sure that the two parameters…… there is a mistake here…. there two parameters p and r so I am making sure that the two parameters are in a0 and a1 so that both caller and the callee understand where the values are to be looked at and this (Refer Slide Time: 21:27) is the call to min procedure.

(Refer Slide Time: 21:36)

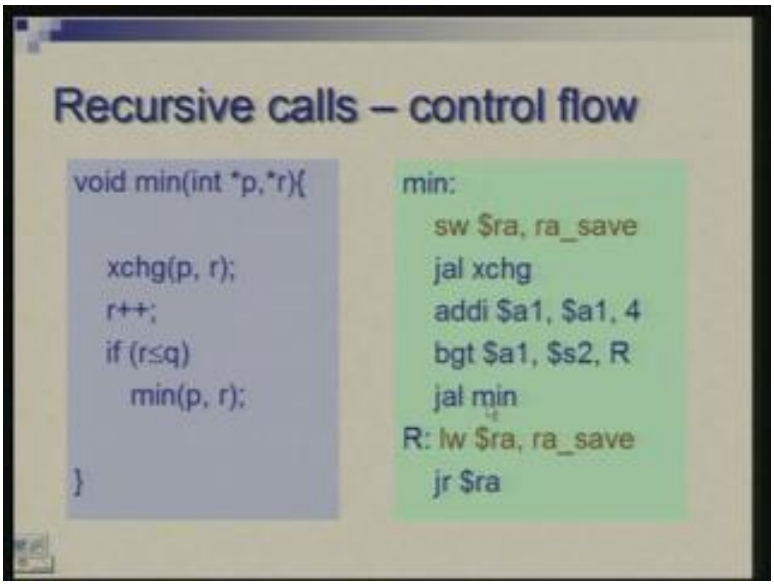


This is the same min procedure and this is how we have implementation. I have added these two instructions here (Refer Slide Time: 21:44) apart from that it is a simple loop. This is the same loop, makes a call to exchange, updates r, compares and goes back. So

this is a simple loop here that basically forms the body of this function; all I have done is I have actually padded up with save and load here. So here I am saving ra value into this location and I am loading it to this location.

Now please notice here that I have used this load and store somewhat like a pseudo instruction. I am not worrying about whether this address call ra save is a small constant or large constant or how it is to be handled. I am just leaving it as it is. The assembler will translate this into possibly two instructions or two or more instructions which will prepare the address into a register and then with the suitable offset it will use sw lw instruction. So again elaboration of this we will see separately. But all that we need to understand here is that ra is getting saved somewhere and from the same location ra is being restored.

(Refer Slide Time: 23:13)



Now let us move a step further and go for what is called recursive call which means a procedure directly or indirectly that calls itself. There is a cycle which is formed and we have basically changed this loop which was inside min into recursive calls. It is not necessarily improving the program but just for illustration I have rewritten as a recursive call here. So, if r is less than equal to q instead of saying jump to exchange I am saying make a call to this function and meanwhile the value of parameter has changed, r has changed so you will keep on calling this and return when you find that this is false; when r exceeds final value then you start return and when r is unchanged so you will keep on finding that it is false and all chain of return will happen.

Now let us see its translation in MIPS. I took care of saving this and restoring this. Earlier I was making this comparison and looping back. So instead of looping back I am making a call to min again. Now the parameters I am maintaining in a0 and a1 so I do not need to

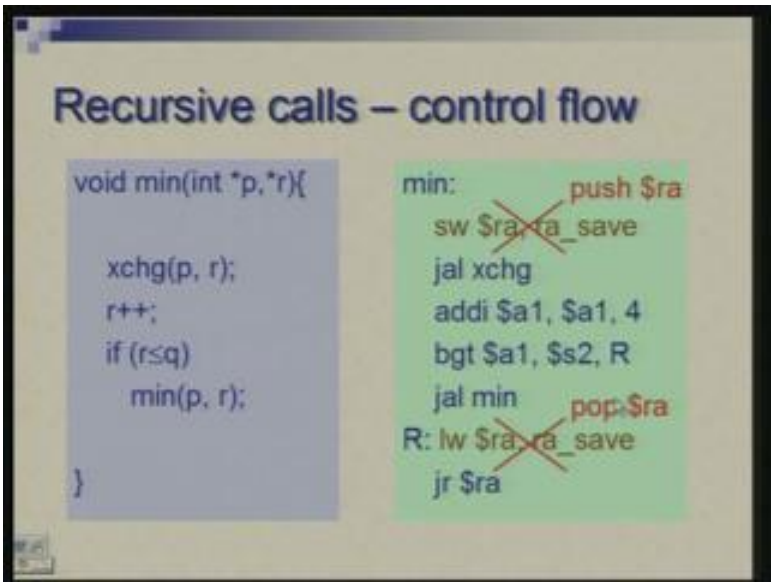
do anything else I simply make a call and I am hoping that the control will repeatedly enter this and appropriately exit.

Now what will happen if I do this; where do you see the problem?
Yes, every time I am saving ra value into same fixed memory location. So, first time the value which gets stored is a value corresponding to call which came from outside; subsequent calls are getting generated inside. So, subsequent returns are to this point. As you see here there is a call (Refer Slide Time: 25:51) and the return takes place here so every time now return will take place here because the original entry point from outside has been lost. So the solution of this is that I should not lose any value which is saved and in natural structure natural data structure where this value can be saved is a stack which is a last in first out structure because the order in which calls occur and the order in which returns take place are in a last in first out manner and therefore as I enter into function from wherever call is occurring the return address gets pushed in a stack and just before returning I pop the latest one from the stack and use it for jr instruction.

Therefore, irrespective of how calls are occurring; whether it is nesting of calls to different procedures or there is a recursion to the same procedure directly or indirectly you can simply keep on pushing the return addresses into a stack. As soon as you enter a function push the return address into stack and just before exiting take it off from the stack and return.

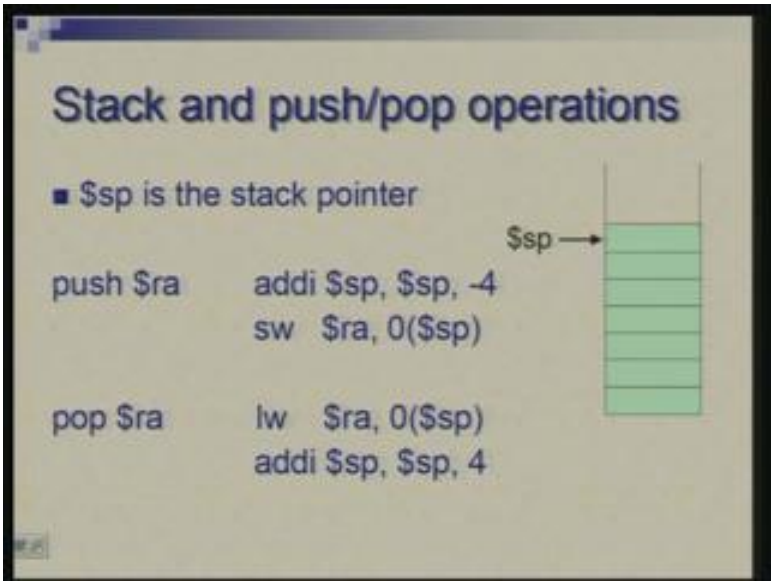
Now the question is how do you do this?
There are no direct instructions in MIPS which available for pushing and popping.

(Refer Slide Time: 27:25)



The stack is created basically by using special register called stack pointer. So, once again it is one of the thirty two registers which is used to implement a stack.

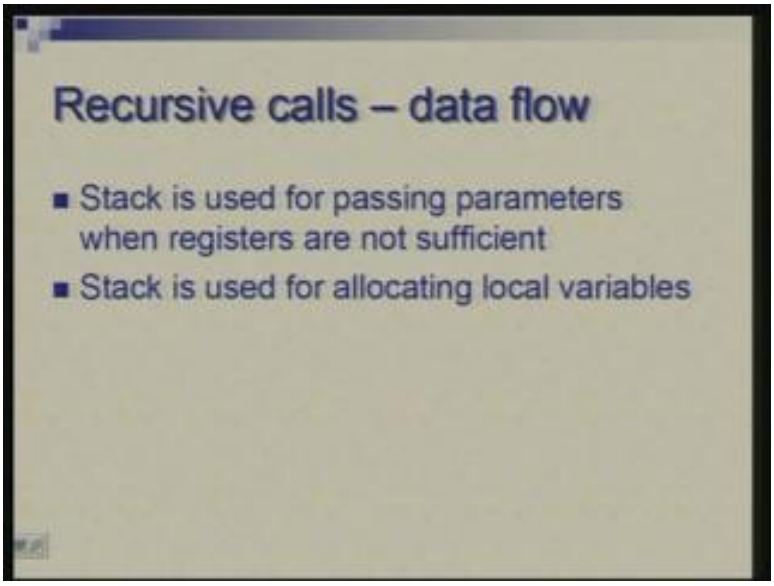
(Refer Slide Time: 27:38)



Pictorially let us say this is a stack and conceptually you can make stack grow towards reducing addresses or increasing addresses. I am imagining that let us say address 0 of memory is at the top and maximum address is at the bottom so somewhere I define bottom of the stack and start building stack towards lower addresses towards 0 and this register sp will always point to top of the stack (Refer Slide Time: 28:12). So, for push what I need is to decrement the stack pointer to create space for putting in data and then store the value you want to put in the stack.

Hence, add immediate sp, sp minus 4 then store word ra at sp. Now one more thing I like to notice, this is just a side observation here that, in add immediate also the constants can be positive or negative and actually it is because of this reason that probably I had listed subtract immediate as an instruction but actually there is no subtract immediate instruction; you add immediate with a negative constant which is nothing but subtraction. So, here you are subtracting 4 from stack pointer and pop is just the opposite so you pick up value from the stack as pointed by the stack pointer, put it in the desired register and update the stack pointer increase it by 4 so that the value is no longer considered to be the part of the stack. These are the instructions which you would use to save and restore the values of return address.

(Refer Slide Time: 29:42)

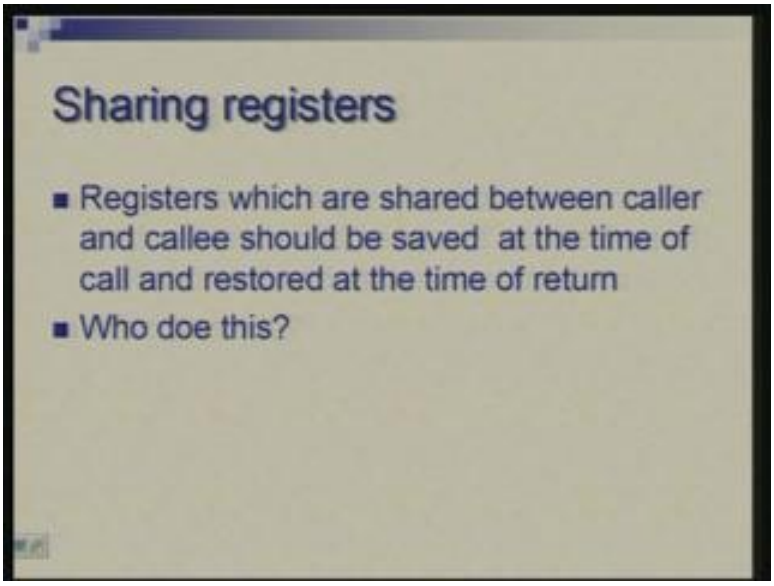


So this has taken care of the control flow in a recursive environment. We are ensuring that information about return does not get lost, what about data? When you are passing parameters in a recursive call….. in the previous example (Refer Slide Time: 30:13) the situation was simple that after the call when I return back I do not need those values. So basically every time a call is made the fresh values are passed on and what happened to old value I do not have to worry about. But you could have a situation where there is some code after recursive call also. You passed on values p and r but you are still working with them you need to do more operations. So, in that case you cannot afford to lose old values and the solution again is to use a stack.

Therefore, the parameters could be passed through registers if situation is simple but if you need parameters to be available even after the nested calls old parameters you need to keep them in the stack so you can save the parameters you can pass the parameters through stack. Also, recall that I mentioned that if number of parameters is more than 4 you use memory locations. Now that question also gets answered as to which memory locations. So additional parameters you can simply push in the stack. So before you call you load the parameter in the stack and when you are inside the function you can take it off the stack. Stack is also used for allocating the local data.

Therefore, now imagine that you have a recursive call to a function which has its own local data. So, it means for every occurrence, for every instance of the function a new array has to be declared. Suppose there is a local array so with every new call a new array has to be declared and they cannot be all located at a fixed address. Once again the natural place for them is stack. So, on the top of the stack when you enter a function you can create you can earmark space which corresponds to a local data so you can create local arrays or all local structures at the top of the stack and before you exit you can clean up that area because it is no longer required.

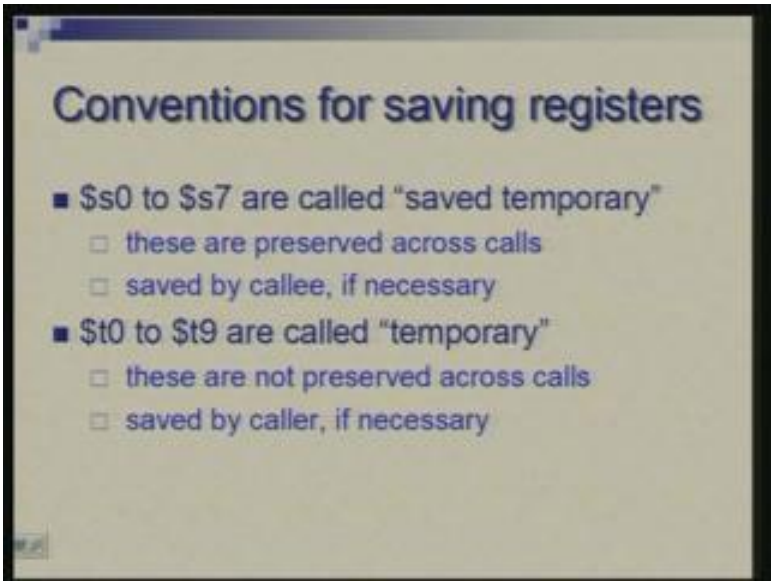
(Refer Slide Time: 32:38)



One more problem one has to pay attention to when talking of procedure is when you are writing a program you assume availability of certain number of registers and you may freely use them but when a call occurs the control is getting transferred to a function which will again require registers for its own computation and when the control comes back to the main the main program will continue with the computation which was done partially. So some intermediate results may be available in the registers and if you are not careful you may conflict. Imagine a situation that the main program is written by one person one programmer and the function is written by another programmer. So either they write in sequence; once 'a' finishes he tells that I have used these registers you go and use other registers or they could be a fixed division. So there is a convention which can be followed which ease the task and if necessary you save the registers.

Let us say you are writing the calling program, you have done partial computation some results are in the registers you need to make a call and come back and continue. So the values which you need to preserve you have to take care that you save them. So, whether caller should save or callee should save; again a convention helps in this area and this question this conflict will not be there otherwise you would write programs, two people will write a program there will be a conflict one would destroy the value of the other and the fingers will point at each other.

(Refer Slide Time: 34:34)

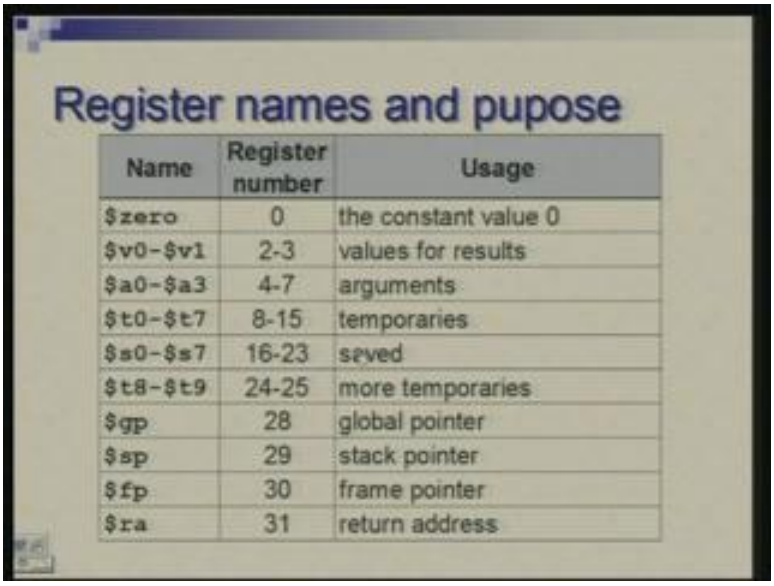


So the convention which is followed here in MIPS is given here as registers s0 s1 s2 etc these are called saved temporary registers. The caller can assume that it is safe to leave values in these registers and everyone has to ensure that values of these are preserved across calls. So, if some partial result are left in s registers you can be safe if everyone is working, everyone is complying with the convention then you can assume that values will be retained and if callee feels the necessity of using registers it will be made in a transparent manner; it will save the values before using, save the callers' value which were left in these registers before using make use and then restore those values.

So, as far as caller is concerned the caller will stick to that assumption that values in these is not disturbed; if it is disturbed it is done in a manner that you do not come to know of. Similarly, registers t0 to t9 are called simply temporary registers where values are not expected to be preserved. So callee has no responsibility of leaving these untampered and on the other hand if caller requires these values to be saved if it needs more value than this eight to be saved across calls it can put in these but then there is no guarantee. So, if there were values in this caller is expected to save them somewhere safely then make a call and when you come back recover these values. So these are not preserved across calls and they are saved by caller if necessary.

So these conventions define who has what responsibility in terms of tampering with the values or touching the values and saving in case if it is required.

(Refer Slide Time: 37:00)



So now all put together we have been talking of lots of register names and there is some convention, in some cases there is hardware constraint so here is the summary of all the registers which we have talked about. So, starting with 0 this is ensured by hardware that the value is constant 0 in this. this is one thing you missed out here (Refer Slide Time: 37:33) $at is register number 1 it was assembler temporary which is used for expanding pseudo instructions.

When expanding pseudo instructions requires a temporary calculation for example preparation of address or storing the comparison result then assembler uses $at or register 1 and the programmer is expected not to use that because if you use it and at the same time you are also using some pseudo instruction you can run into problems.

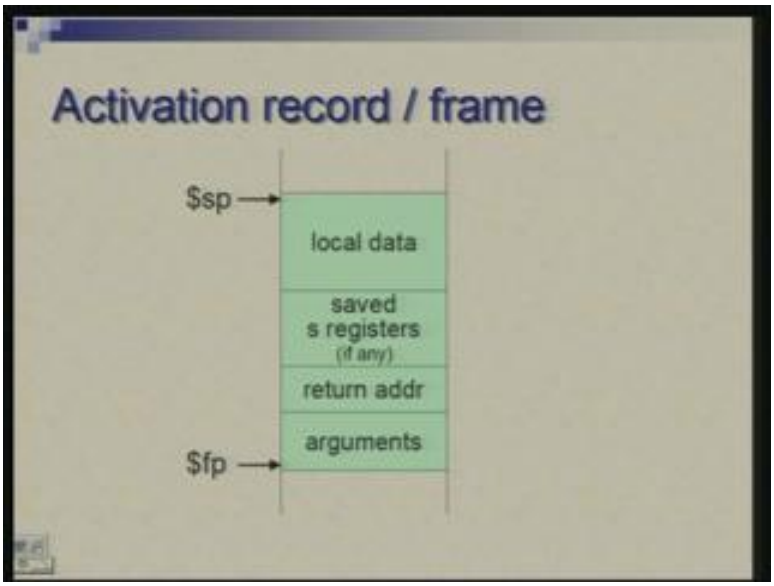
Then the next two are for parameter passing; this v0 and v1 are basically number 2 and 3 and they are used for returning values; a0 to a3 are passing values into the procedures they are numbered 4 to 7. Then we have t0 to t7 these are temporaries; s0 to s7 are saved temporaries; for some reason t8 t9 are not contiguous to these then there is a gap there are some registers which are used by kernel again, they are reserved and then we have global pointers.

Therefore, if you have global data which is shared by many functions, many procedures then that could be in some contiguous area and a register gp could be made to point to that and with suitable offset to that register you can access various components of that data; sp is the stack pointer which I talked of; ra is the return address and fp is frame pointer for which I will describe its use shortly so these are the names which you can conveniently use, internally they are $zero, $two, $four etc and that is the usage
so hardware-wise it is ensured that register 0 has certain value and jal instruction assumes that value is to be put in ra; the others are by convention. So the fact that we are using

these for parameter passing is only a matter of convention the hardware does not know this.

Similarly, the convention about these about t's and s is again a convention and hardware does not understand this. Stack pointer is again the convention we are using this particular register for pointing to stack; anyone can be used once again it is a convention. With ra there is a role of the hardware in the sense that when you execute jal instruction the return address is put in this particular register. The return instruction the jr is a general instruction; it is not specifically for return and we have seen other usages of it. Finally I like to show you what is called an activation record or a frame.

(Refer Slide Time: 40:52)



We have talked about putting so many things in the stack. Once you come to recursive calls then basically everything the solutions for everything I mentioned was stack. So all this information in the stack is organized in a particular manner which is again a convention and different systems follow different convention. So what is typically followed is shown here.

Every time a function or procedure is called you create an activation record on top of the stack and when you return you clear that of. So, as nested calls occur you build these activation records in the stack or the frame in the stack and typically the stack pointer will point to top of the activation record and other pointer which I mentioned other register fp is called frame pointer which points to the beginning; it could point to the first location here or may be the last location of the previous activation record again the convention could slightly vary.

Therefore, I have tried to put almost everything which I discussed earlier. the arguments which are being passed on would be one part of this record the return address which is to be saved is saved here, any s registers you need to save they get saved here, local data is

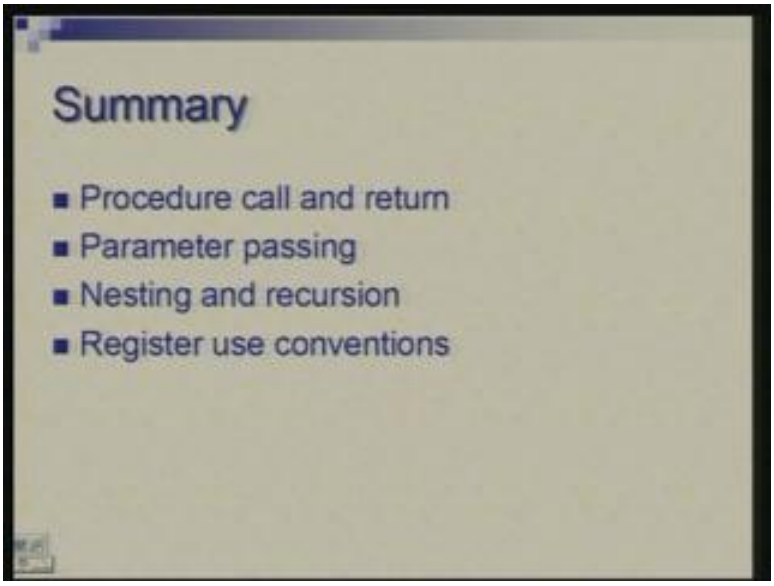
allocated here so this whole thing is the frame. So a function basically works with this, this is what you see as local data area and apart from this it may make reference to some global data and that will be accessible through gp pointer. So, through gp, fp and sp access is made to all the data within a function.

Now the question would be why we are having two pointer sp and fp?
One might imagine that suppose you have a pointer either here or either to the bottom or to the top everything could be referenced in terms of some suitable offset from this point or that point. But there is a difficulty in that particularly when the size of this changes dynamically. If within the function you are doing any dynamic storage allocation; we have not seen at assembly level how you do that but suppose the program is sophisticated and dynamically allocation gets done (Refer Slide Time: 43:42) so this moves up. Now this can no longer act as a reference point for accessing the data because we want the offset to be constant. Typically the way you like to access is load word, some constant offset and may be register in the bracket is sp.

So with respect to this pointer the offsets for these are constants. But with respect to this pointers offsets may not be constant. We need a pointer to this top to keep track of how far the stack is filled up so that is required in any case. But another pointer here so that all these can be accessed with the constant offset is required.

Let us summarize let us close at this point. We have seen some of the basic ideas of how we create procedural abstraction in assembly language.

(Refer Slide Time: 44:48)



First issue was that of arranging procedure call and return; then we saw how parameters are passed; we talked of complications which arise because of nesting of calls and also recursion. The solution was to follow lot of conventions and also to use stack for all storage allocation. With that we have seen the basic idea.

I would in the next class take an illustration and show how a complex recursive procedure can be programmed where you need to do activation record creation. I will stop at that.

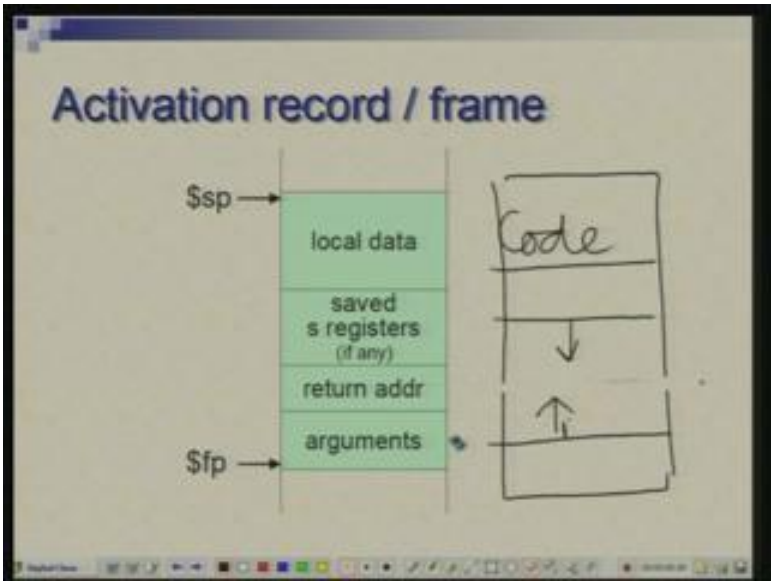
If you have any questions I would answer.
(Student: can you explain frame pointer…….45:43) yeah, frame pointer; the basic idea here is that we have all the data allocated on the stack. If the data was of a size which is fixed when you are writing the program then every piece of data can be accessed with constant offset with respect to the top of the stack. But if the situation is different that means let us say there is some data structure may be you have prepared….. there is a local link list for example where the size may grow or shrink as the function proceeds so all that allocation of the link records will be on top of the stack. Therefore the top of the stack will keep changing.

So, stack pointer can no longer be used with constant offsets to access for example these arguments or the return value or these registers. So we have a pointer to the bottom and with respect to that the offsets of all this part is constant. of course the access to the dynamic part will be changing and it will not be accessed in same manner so for that we would need more complex methods but at least the part which is unchanged whose location or seat is unchanged during the life of the function we can simply access by constant offset to the frame pointer.

There was some question over there?
Actually in aggregate terms the pushes and the pops have to match. You might push different things in PC; for example, you may push arguments, then you may push return address, then you push registers so several instructions may use and stack pointer may be made to grow in many steps. Now, when you need these things you can use them and you may clear off activation record in one go or again you may do it in parts. All you have to ensure that the space which was allocated should be equal to the space which gets deallocated eventually. So these have to be matched. In fact this could be one very very common source of error if your pushes and pops match, there could be serious problems in the program.

(Student: How much memory is allocated to the stack pointer……48:50) how much memory is allocated to the stack pointer?
The memory allocated to the stack pointer…. the stack pointer is just one register so your question should be how much memory is allocated for the stack? There is often no fixed memory may be allocated; you may start at let us say one end of the memory and allow this to grow in one direction.

(Refer Slide Time: 50:45)



Often you may have in your entire memory you may have….. this is a very typical case that you have two areas two data areas which grow in opposite direction: one is called heap and the other is called stack. So, for example if this is stack (Refer Slide Time: 51:02) it grows in this direction, grows and shrinks, other could be heap which grows and shrinks. So heap would be used for random allocations, for malloc and so on and this stack is used for automatic allocations when calls and returns happen.

What you could do is you may make a fixed partition and say that above this is the area for heap and below this is the area for stack but that may constrain; sometime you need more stack area less heap area sometimes you need more heap area less stack area. So a count policy is to not have this line and let them grow freely so that total space is available as long as they do not clash into each other your program works successfully. So if it does then you run out of memory. I hope I have answered this question of how much space you allocate. You may often not allocate a fixed space and leave it to grow or shrink dynamically, thank you.