

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

**Module No # 02**

**Lecture No. # 08**

Welcome, to the lecture 8th, of the course on, “High Performance Computing”. We are in the process of trying to, understand both, how function calls and the returns are implemented, as well as, how the MIPS one machine language instructions can be used to describe various things. So, the function call and the return is our first example.

(Refer Slide Time: 00:34)

**Example: Function Call and Return**

**What must be done on a function call?**

- Pass parameters on stack
- Transfer control to start of function
- Remember return address
  - Where? On a stack (in main memory)
- Save register values on stack
- Allocate space for local variables on stack

**What must be done on a function return?**

- Pass return value (through stack)
- Restore register values from the stack
- Clean up stack
- Transfer control back to return address

523

In the previous lecture, we have worked out the various steps, which must happen as the part of a function call and its part of return. And the fairly obvious ones were there must be a transfer of control, from the “CALLER” to the “CALLEE”. At the point of return, when the return has to be done, there must be also a transfer of control. Then they were various other issues which came up, such as, if parameters have to be passed, then that must be explicitly done; if local variables are present, then space must be allocated for them and so on. If return value to be passed; it must be explicitly done and so on.

So, in this series of steps, we found that there was an important data structure, which would be involved and that was stack, the very same stack that we have talked about during stack allocation of data. We talk then about stack allocation of data explicitly, the local variables and parameters of a functions, which have a life time of: “the call of the function”. In other words, the local variable comes into existence only when the function is called, and it ceases to exist, when the function returns. And we now understand how that happens; the local variable comes into existence, as part of the function call, because there are instructions, which explicitly allocate space for those local variables on the stack.

And those local variables cease to exist, when the function is returned, because there are instructions, which explicitly, free the space that was used, for those local variables from the stack. And therefore, the stack allocation of local variables and parameters that we learn about earlier, we now clearly understand, as something that is implemented either by the programmer, if the programmer is implementing this series or steps or by the compiler, if it is the compiler that causing instructions, which do all these functionality to be included in your program.

(Refer Slide Time: 02:33)

**Implementing a Stack in Memory.**

Example: Growing down (into lower addresses) in memory  
R29 pointing at current top of stack element

0x1234	
0x1235	
0x1236	
0x1237	Stack Element

PushByte: SUBI R29, R29, 1  
SB 0(R29), Rs

PopByte: LB Rd, 0(R29)  
ADDI R29, R29, 1

R29 0x1237

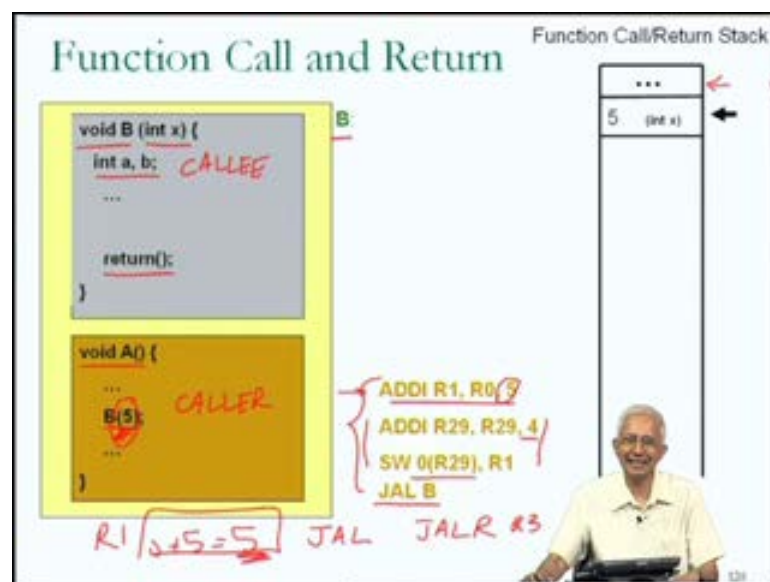
The slide features a diagram of a memory stack with addresses 0x1234, 0x1235, 0x1236, and 0x1237. The stack grows downwards, with the current top element at 0x1237. A red box highlights the current top element. To the right, assembly code for pushing and popping bytes is shown. At the bottom, a small inset image shows a man speaking.

Since, the stack is the important part of this, we look at how to implement the stack in memory and saw that in order to push something, on to the stack, a two-instruction sequence was necessary: a subtract or and add, depending on whether the stack is going

up or down in memory, followed by a store. It would be a store byte, if pushing a byte on to the stack or it would be a store word, if you were pushing a word on to the stack. As far as the pop is concerned, the pop operation, which removes, the top most elements from the stack, is once again a sequence of two instructions: “A LOAD and ADD to update the stack pointer”.

In this particular example and in general we understand that the stack pointer (Refer Slide Time: 03:17) is very important part of the stack and therefore, of the function call and return mechanism, and since it is frequently accessed, it would not be beneficial to implement the stack pointer itself as a variable in memory, but rather to dedicate a general purpose register, for the stack pointer. And If one is reading programs in assembly language, instead of seeing the general purpose register refer to by its actual name R 29, one might for example, find that referred to, by the abbreviation Rsp, or as in the case of the assembly language file, that we saw earlier percent sp. So, the stack pointer and the stack are very important structures as far as the function call is concerned.

(Refer Slide Time: 04:00)



Now, let us go back to our example of the function call. We had a situation where there was some program; there was the function which is being called as function B or void B, takes one parameter, has two local variables a, b; and this is the point of return, and then there is a function A, which calls the function B with a parameter value of 5. So, in the

previous lecture, I had these functions with function A on the top and function B at the bottom. Here, I choose to do the other way around. Just note that we refer to A as the “Caller”, and in other words, the function call takes place inside the caller function and we refer to B as the “Callee”.

So, now we understand the series of steps that has to happen, and let us just run through them, in terms of the MIPS 1 instruction, that could implement that step, and the impact of that execution of those series of instructions, would have on the stack. So, we are going to run through this step by step.

Now, let us suppose that the stack is as shown in the diagram over here (Refer Slide Time: 05:11). Here, I am showing you a slightly different stack, from the one that was in our previous lecture, because here the stack, the stack pointer is pointing at this location and from the orientation of the stack, you would guess that the stack is going to grow in this direction, downward, in other words, in to a higher and higher memory address, which is the opposite of what we had with our, in the slide we saw just few seconds ago with the push byte and the pop byte. So, this is the stack, which is growing into higher and higher memory addresses.

So, just let us look at the first series of instructions, and try to understand what they are doing. We will do this, the other way around. Rather than saying, what series of instructions can implement **this function** this operation, just look at the series of instructions and try to understand on how they might be contributing towards the function call and return. So, the first instruction that you see here is add immediate, this one (Refer Slide Time: 06:09), ADD Immediate R1, R0, 5.

So, what does this instruction do? It is an “Add Immediate Instruction”, and it has an immediate operand of 5, the other operand is R0. R0, you will remember as general-purpose register zero, which always contains the value of zero. Therefore, what this ADD is doing is it is adding 0 to 5, which is equal to 5 (Refer Slide Time: 06:29), and putting that result into the destination register, which is R1. In other words, all that the first instruction is doing is that it is, getting the value 5, into register R1. So, R1 now contains a value 5 and that is all the first instruction is doing.

Now, the second two instructions look vaguely familiar, the ADD Immediate R29, R29, 4, and store word zero R29, R1. This seems to be a series of instructions, which is capable of pushing on to the stack. Remember, as far as pushing is concerned, we expect to see store word or stored byte or store half. So, what exactly is happening over here? Here, the first instructions in the sequence, ADD Immediate R29, R29, 4; R29 is the stack pointer, so this instruction is incrementing the stack pointer by four.

So, R29 plus four is a new value of R29. What is that mean to increment the stack pointer by four? Why is it being incremented by four? In our previous example where we were incrementing the stack pointer or decrementing the stack pointer, we did so by one, because we were pushing or popping one byte on to the stack or from the stack. In this particular example over here, we are dealing with integers and the assumptions seems to be that the size of each integer is four bytes, which is why in order to stored a word on to the stack, I need to push four bytes on to the stack or I need to increment the stack pointer by four bytes. So, this explains why the ADD Immediate R29, R29 is 4 (Refer Slide Time: 07:58). This is causing the stack pointer to get updated by four.

Then at that particular memory location, which is refer to using, “Base Displacement Addressing Mode”, by displacement of zero from R29, is where I store the value which is currently stored in R1. What is R1 contained? R1 contains the value 5. So, the net effect of these three instructions (Refer Slide Time: 08:19) is to push the value 5 on to the stack. This is the stack where at any given point in time, the stack pointer points at the current top of stack. Which is why, I increment the stack pointer first and then I store into that location, so the stack pointer R 29 always points at the current top of stack.

So, the net effect of these three instructions is that my parameter of value 5 gets pushed on to the stack. So the stack pointer, which uses to point, before the sequence of three instructions, the stack point, was pointing over here. After the sequence of three instructions, 5 have been pushed on to the stack, and this is the stack pointer that has been incremented. Therefore, stack pointers pointing at this current top of stack. This is the pushing of the parameter, on to the stack.

Now, you will call this is the passing of the parameter value 5 (Refer Slide Time: 09:18), **and that is only** other than this value 5, it had no special meaning. It just happens to be the value of the parameter in this case. Now, after this the jump transfer of control to the


function, which is being called the Callee, can take place. As I have been, suggesting all this time, the transfer of control, the implementation of the function call, is going to be done using the, “Jump and Link instruction- JAL”.

In this particular case, I will use the two versions of the, Jump and Link instruction, as you will remember, as JAL and JALR (Refer Slide Time: 09:53), and they differ only in that. Jump and Link, takes it is operand in absolute addressing mode, and jump and link register takes it operand out of a register. In our current situation, remembering that, I will use labels to represent memory locations in terms of absolute addressing mode.

In order to refer, to the transfer of control to the function, I use Jump and Link B- JAL B, in other words, execute the Jump and Link instruction, the target address of whatever the address of B is, and B must be referring to the first instruction in the function B. That is what the label B, will be associated with. That in fact, what we have over here, the first instruction associated with the function B has a label of B, associated with it.

(Refer Slide Time: 10:44)

Recall: MIPS 1 JAL instruction			
	Mnemonics	Example	Meaning
Conditional Branch	BEQ, BNE, BGEZ, BLEZ, BLTZ, BGTZ	BLTZ R2, -16	If $R2 < 0$ , $PC \leftarrow PC + 4 - 16$
Jump	J, JR	J target <sub>26</sub>	$PC \leftarrow (PC)_{31:26}    \text{target}_{26}    00$
Jump and Link	JAL, JALR	JALR R2	$R31 \leftarrow PC + 8$ $PC \leftarrow R2$
System call	SYSCALL	SYSCALL	



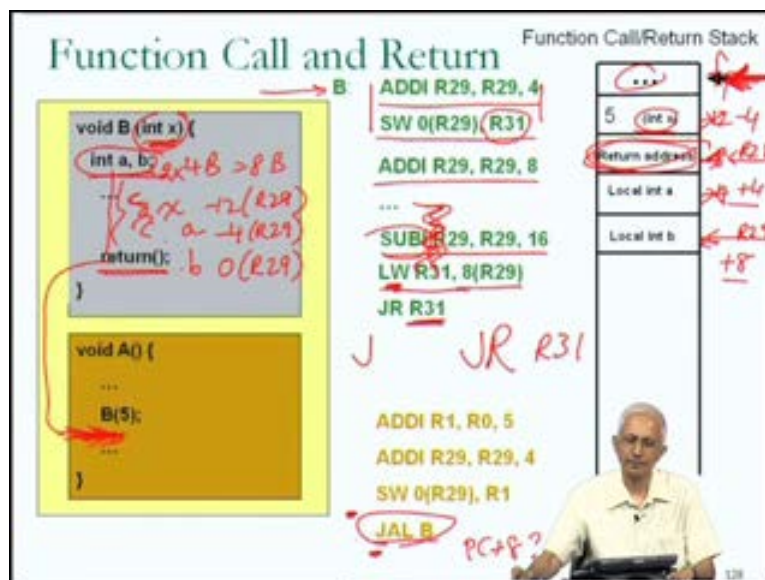
Now, let us just remind our self, what the Jump and link instruction does. Going back to the table of the MIPS 1 Jump and link instructions – MIPS 1 JAL, the jump and link instruction over here is the example Jump and link R R 2 – JAL R R2. But the example we have here, is this Jump and Link to the label B, which is similar. Other than, instead of copying the value of R2 into the program counter, we are going to copy or we are

going to store the address of B, whatever the label associated whatever the address associated with the label B, into the program counter. So, the second instruction or the second operation is causing the controlled of transfer of control to happen. What is the first instruction doing? The first instruction, as we now understand, is helping us in the remembering of the return address.

So, it is remembering PC plus 8 (Refer Slide Time: 11:32), which must have something to do with a return address into register R31. Remember, R31 is an implicit operand, of the Jump and link instruction. R31 is not mentioned inside the instruction, but it is used by the hardware, in implementing the instructions.

Therefore, in these two steps, the second one is the transfer of control for the function which is being called; the first is something to do with a saving of the return address. The return addresses is something after the program counter and remember the program counter, in this particular case, will contain the address of the Jump and Link instruction. Therefore, PC plus 8, is one of the instructions following the Jump and Link instruction.

(Refer Slide Time: 12:11)



So that seems to be... We will talk more about the 8, where it comes from, shortly. But in any event, at this point, we understand that the Jump and Link instruction down here is going to cause control to be transferred to the beginning of function B. Now, inside function B, you will recall that it is important to we remember the written address. By

just keeping it in register R 31, where it was left by the Jump and Link instruction, is not going to be safe. This is because due to subsequent recursive function calls or nested function calls, etc.

Therefore, one of the first things to be done, inside the body of the function B, is to save that return address, as we saw on the stack. How does one save the return address on the stack? Once again, using the instructions, corresponding to push, in this particular case, once again, I increment the stack pointer using ADDI, immediate R29, R29, 4, and then I stored the contents of register R 31.

What does R 31 contain? (Refer Slide Time: 13:13) For the moment, without fully understanding it, we expect that R 31 contains the return address. In other words, **the address where** the address of the instruction to which control is to be transferred, for the return has to happen. That is what R31 contains. Why it is PC plus 8? We do need to understand, but we are postponing that.

So, the first thing that we do that we find inside the function call at the end of inside the function is a sequence of two instructions, which will push the return address on to the stack. As a result, of executing those two instructions, what does the stack look like? The stack looks like this (Refer Slide Time: 13:50).

The stack pointer has gone up by four, and the thing at the top of this stack is the Return address, the address in memory of the point in the instruction, which is well control and is to be transfer on the Return. We are ready to actually, execute the body of the function that is what should follow here. However, prior to doing that, we have to allocate space for the local variables. In this particular example, I am not going to include instructions for saving and restoring of registers.

But if it had to be done, then I would have to save registers prior to the function call, and restore the registers, just at appropriate point in time. Therefore, in this particular example we are not looking at saving and restoring of the registers. However, we do have to allocate space for the local variables, since function B does have two local variables. If function B, did not have any local variables then this would not have been the step that had to be incorporated.



How do I allocate space for integer a, and integer b? In order to this, I must know how much space integer a, and integer b would require. Each of these is an integer variable. We are working under the assumption that each integer variable occupies 4 bytes, the 32 bits (Refer Slide Time: 15:02); therefore, the amount of space that has to be allocated is 8 bytes for the integer variable a, plus integer variable b.

How do I cause that space to be allocated? Basically, I can do this, by just incrementing the stack pointer by eight, which I do, by a single instruction: Add immediate R29, R29, 8, the stack pointer will then be pointing at the location of local integer variable b, and the space above that is the local integer variable a. Above that is the Return address and above that are the parameters, which had been pushed on to the stack.

Which is in fact is the integer variable x as far as the function is concerned. So, at this point the body of the function can be executed. The question that you will ask is, inside the body of the function, is it possible that the variable x is referred to? And the variable a, is referred to? And the variable b is referred to? How are these variables going to be these addresses of these variables are going to be understood? How the compiler is going to generate the address of each of these variables? Looking at this diagram (Refer Slide Time: 16:11), which is on the right side of the screen, it is fairly clear to know how that is going to be done.

Because we can see, right of the back that in order to refer, to the variable b, one just has to refer to the stack pointer with an offset of zero. In other words (Refer Slide Time: 16:27), 0 (R29), inside the body of the function, 0 (R29) is the address of the variable b.

What is the address of variable a? It is going to be stack pointer, but 4 bytes above. Therefore, this is going to be relative to the stack pointer or to the base register R29. A displacement of minus four and that is what the address of a, is going to be.

What about the variable x? So, this I would call this minus four and this as minus eight. I realize that the variable x, the parameter in terms of its local name as x, is going to have associated with an address of minus twelve R 29 (Refer Slide Time: 17:11). Therefore, inside the body of the function if it has any need to refer x, it will be done so as minus twelve; displacement of minus 12 from R 29. We can recall R 29 as the stack pointer, which during the execution of the body of the function is pointing at this location, and

just why using this simple picture of what the stack looks like, the compiler can compute the address of any local variable or any parameter.

Now, I see a lot of local variables and lot of parameters. It could be the case that these have a very large negative offsets, and the displacements and therefore, the compiler has the option of doing something little bit different. In general, the variation would go along the lines of, while a stack pointer is pointing at the top of the stack, the compiler could actually associate with some intermediate location in memory, in another register, let us say here as R28 (Refer Slide Time: 18:06). For example, let us suppose that R 28 is made to point at the return address location.

The advantage of doing this is that as far as the local variables are concerned, and if I am addressing these local variables as displacements, from R 28, rather than from R 29, then the local variable a, will have a displacement of plus 4, and the local variable b will have a displacement of plus 8.

What is about the parameter x? It will have a displacement of minus four (Refer Slide Time: 18:33) relative to R28. So, under this convention, all of the parameters, and there could be a large number of parameters, would have negative displacements from this pointer and all local variables would have positive displacements from this pointer.

One could refer to this pointer by some other name; obviously, cannot be referred to as a stack pointer, since by definition, the stack pointer always points at the top of the stack. But one could refer it by some other name, and the one name which is often used for it, is referred to as the “frame pointer”. So that is something, which the compiler could do. But in this particular example it suffices to just refer to local variable “b”, as zero; displacement of zero from R 29, whereas, “a”, as minus four; displacement of minus four from R 29 and so on.

So, the compilation of the body of the function, will be done by the compiler, based on whatever the different statements are, and uses of the variables x, a, and b can be therefore dealt with, appropriately. The next interesting point in the execution of our sequence is when we reach the return (Refer Slide Time: 19:36).

What has to be done at the point of return? One thing, which we saw had to be done, was the cleaning up of this stack. We are sort of assuming that we have finished executing the body of the Function. We are about to Return transfer control back to the point of Call inside the Caller A, and we just want to clean up the stack. We want to get the stack pointer back to, what it should like as far as Return to “A”, is concerned.

How should the stack pointer look like, at the point in time when it has Return to A? The answer is in the picture, which shows that it should be the same as it was, when we started this example. In other words, the stack pointer should point at that the box label: dot-dot-dot, write at the top (Refer Slide Time: 20:19). Currently this stack pointer is pointing at the local integer b. I need to make a point at the location, way up there; that is going to be a displacement of minus four, minus eight, minus 12 and minus 16.

In other words, this is done to get the stack point pointing to where I wanted to, in other words, in order to implement the cleanup of the stack operation, I need to subtract sixteen from the stack pointer. So, I have the instruction: SUBI-subtract immediate R29, R29, 16. The basic purpose of this statement is to clean up the stack.

So, the stack point is now pointing back here (Refer Slide Time: 20:55), and essentially these elements, are no longer on the stack. Now, I would remind you that there is no subtract immediate instruction in the MIPS one machine language, but there could well be a subtract immediate instruction inside a MIPS one assembly language. Since it is trivial, for the assembler to translate, the subtract immediate to equivalent add immediate, with a negative of 16 as the parameter as the immediate operand.

At this point we are ready to transfer control back to B. There is no Return value to be passed. All that we have to do is to get the Return address into a place, from which it can be used to do the transfer of control. Now the Return address is still actually available in the memory. Remember we have decremented the stack pointer, at a point at where it is suppose to be subsequent to the Return- to the Caller.

But the Return address is available at plus 8, from the current location of the stack pointer. Therefore, to read the Return address, I can look at plus 8 from what R29 currently points at and what this instruction does is it reads the Return address from 8 of R29 using a displacement of 8 from the current value of R 29 and loads those 4 bytes.

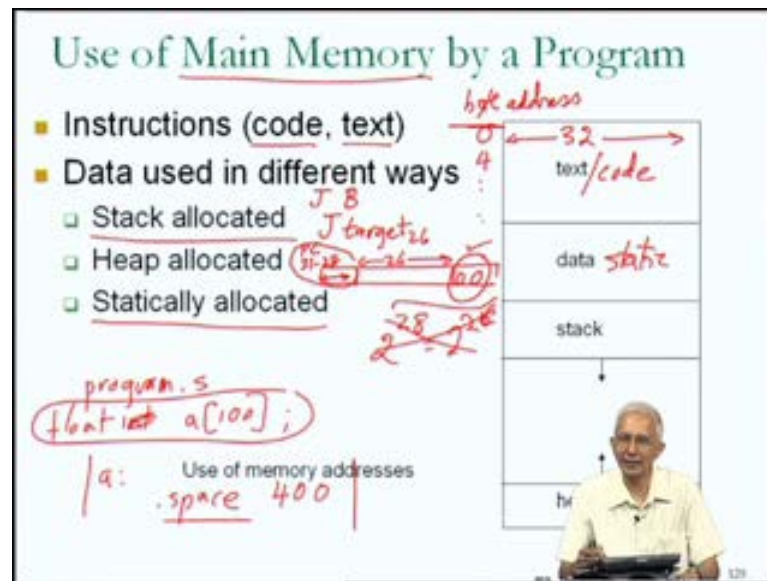
This is a load word instruction into R 31. I am going to use R31 to transfer control back to the correct Return address. How do I actually do the transfer of control back? I can do that using the Jump Register instruction. I will remind you that there are two kinds of unconditional control transfer instructions in a MIPS instruction set. One is a Jump instruction, and the other one is Jump Register instruction. In this particular situation, the Jump Register instruction is more useful because I have the target address available in a register. Jump register will allow me to transfer control back to an address, which is specified in a register. The result of this instruction is that control gets transfer to PC plus 8, which is the value that had been stored inside R 31, by the jump and link instruction.

This implements the complete sequence set which we were looking for. As I said, we have left out some other complexities of some Function Calls like the need for returning a value or for saving or restoring registers, but otherwise, we have pretty much understood what instructions would be involved and how the stack is involved in. What we now see is somewhat complicated Function Call and Return operation.

Now, just going back to previous lectures you will recall that where I had mentioned that in CISC instruction sets, it is possible to have single instructions, which are somewhat complicated. It is also conceivable in some of the CISC instruction sets, the entire functionality that we see over here, could have been implemented, in one or two instructions. Over here, you will notice that the sequence is implemented in more than ten instructions.

Each of those, one or two instructions in the CISC instruction set would have been clearly much more complex, than the simple instructions that we have in our RISC instruction set. The most complicated instruction that we have in this RISC instruction set is the Jump and Link instruction, which does two primitive operations, every other instruction just as one primitive operation, so much, for the Function Call and Return.

(Refer Slide Time: 24:09)



Now, with this in mind, we can go back to picture of from the very first lecture, where we talked about how different variables differ in their lifetimes and how some variables are statically allocated, like some data is statically allocated in memory. Other data is stack allocated in memory, and other data is heap allocated in memory. Now, we can now put all of these thoughts together, into a description of knowing how the main memory is used by a program. When you write a program and you run it on a computer, how the main memory is used. Remember, we use the term “Main Memory”, to refer to that box inside the computer organization block-diagram. Because the word memory is overloaded, the registers themselves are a form of memory and so on. A very important thing, which must be present in main memory, for a program to be executed, is the program itself. From now on talking about the program, we bear in mind that the program itself contains both instructions and data. I will talk about the instructions of the program using words like code or even text (Refer Slide Time: 25:18). You will often find the word text or code use to refer to the region to the instructions of a program.

It is very clear that when a program is executing; its own code or text must be present in memory. In addition to this must be present in memory, different data that is used by the program and some of the data could be stack allocated. Some of it could be heap allocated and some of it could be statically allocated. We have seen how static allocation is done, when we see this and we saw this, when we looked at the example of the file program dot s (Refer Slide Time: 26:00).

You remember that we had in connection with **I am sorry, not integer but** float. In that particular program, there was floating point array of size one hundred, and we saw that the way the static allocation was done in the assembly language program, associated with a label a. There was a space, in other words, an allocation of 400 bytes which is the amount of space occupied by 100 floats. Each float is a four byte. I triple e 754 floating point value (Refer Slide Time: 26:37).

So, we saw how static allocation is done. Corresponding to static allocation we will see space directive, inside the assembly language program. We have also seen how stack allocation is done. We saw that the function parameters and local variables are allocated space when the function is called and the space subsequently reclaimed on function return, and that the stack allocation is done by explicitly, by instructions, which have been included into the program for that purpose by the compiler.

So, the net effect is **(( ))** I want to draw diagram which shows me what all is in memory when the program is executing. The diagram will look something like this. First and most important is probably the text or code of the corresponding, to the program that is the instructions of the program. This would typically occupy the low memory addresses associated with a program, so memory address 0 etcetera.

Now, one thing you should bear in mind in thinking about the MIPS instructions. Each MIPS instruction is of size 4 bytes, I mention that each MIPS. Just like each MIPS register, each MIPS instruction is of size 32 bits or 4 bytes. This was confirmed when we look at the 3 instruction formats: the I format, the r format and the f format in lecture 6.

So, since each MIPS instruction is of size 32 bits; that means, that if the first instruction is that at byte address 0, then the second instruction is going to be a byte address four. in other words **as far as the text segment** as far as the instruction are concerned, each instruction is of size 32 bits and instruction addresses are going to be multiples of four. If the first instruction start at address 0, all subsequent addresses are going to be multiplies of four.

Now, this actually is interesting and answers one question which may have come in to some of your minds when in connection with the jump instruction. When we talked about the jump instruction (Refer Slide Time: 28:42), we will recall that the jumping

instruction has a 26 bit absolute operand, and this is the target address. In our notation, we actually use a label. But in general, when the machine language instruction equivalent, for example, we talked about jump to B, let us say B is the label in the program. But that is the assembly language version, in the machine language version that would be translated into a 26 bit absolute address.

Now, this is only 26 bit address, but the PC is a 32 bit register, which means that the actual target address must be a 32 bit entity. And in the table on the jump instruction we had actually seen that the way that the 26 bit absolute address is translated into a 32 bit address is that the 26 bits, which are target 26, are included along with two least significant zeroes, which adds up to total of 28 bits, four more bits are still needed. The remaining four bits are taken from the more significant bits of the program counter in other words PC bits 28 to 31 (Refer Slide Time: 29:50).

Now, the reason I mention this now, is that we understand where the 0 0 are added as least significant bits, as far as converting the 26 bit target address into the 32 bit actual address. The two zeroes make sense because any MIPS one instruction is going to have least significant bits of 0 0 in terms of its address, given that all MIPS instructions are 32 bits in size, and that they address are multiples of four. A multiple of four has a property that its least significant two bits as 0. Therefore, we understand with the 0 0 are coming from.

That is just fallout of the fact that all MIPS instructions are of size 32 bits and the text of your program is going to be loaded from address 0 (Refer Slide Time: 30:37), which means that all the instructions are going to have addresses, which are multiples of four. What about the other end of the target address generation? Why does it make sense to use the most significant bits from the program counter value? You will recall that our interpretations of using bits, from the program counter value are the program counter is the register which contains the address of the current instruction being executed.

At this point, in time, we are talking about the jump instruction. Therefore, the most significant four bits, we are referring to, are the most significant four bits of the address of the jump instruction. Now by copying those four bits **into the** or by using those four bits in the generation of the target address, what we are essentially saying this is the jump instruction and its target differ in only 28 bits and therefore, the maximum distance from

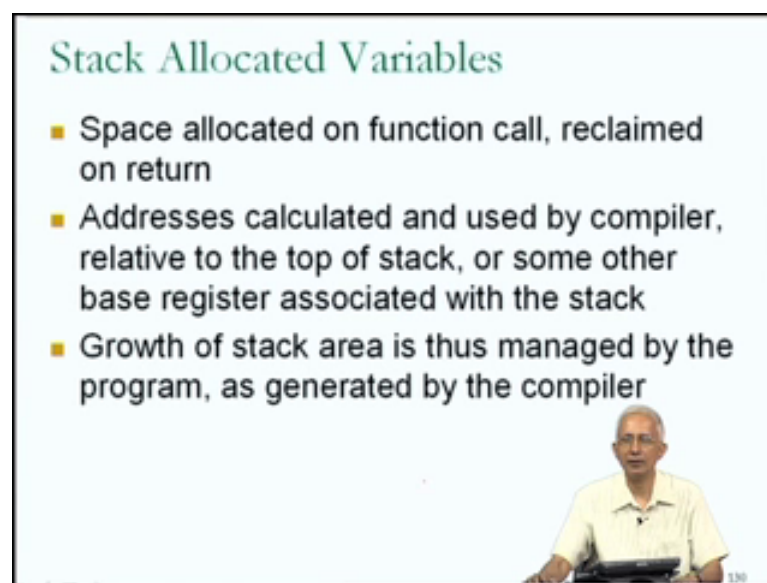
the jump that the target could actually be, is about two power 28 bytes, which is equal to about two power 26 instructions (Refer Slide Time: 31:41).

Remember, that each MIPS instruction occupies four bytes. So, what is this mean? This tells us that as far as the actual distance between the jump instruction and its target, the number of instructions that could lie between them, is could be as much as two power 26. How much is two to the power of 26? You remember, the 2 power 20 is approximately one million.

Therefore, the number that we have over here is significantly more than a million instructions, that is 2 power 6 multiplied by million instructions, which is tens of millions of instructions, which is conceivably, unlikely, to be the case for programs that rewrite will have to be an extremely large program, near about million instructions before this problem kicks in; that the 26 bit target is not adequate to represent the number of the instructions, the distance between the target and the jump instruction.

So, going back to the diagram; the data, the stack and the heap, **the data is stack** this is statically allocated variables. The stack is a stack allocated variables and we understand that the stack will grow and shrink with Function Calls and Returns, and the heap similarly, may have to grow and shrink depending on how dynamically allocated variables are created and freed.

(Refer Slide Time: 33:01)



**Stack Allocated Variables**

- Space allocated on function call, reclaimed on return
- Addresses calculated and used by compiler, relative to the top of stack, or some other base register associated with the stack
- Growth of stack area is thus managed by the program, as generated by the compiler

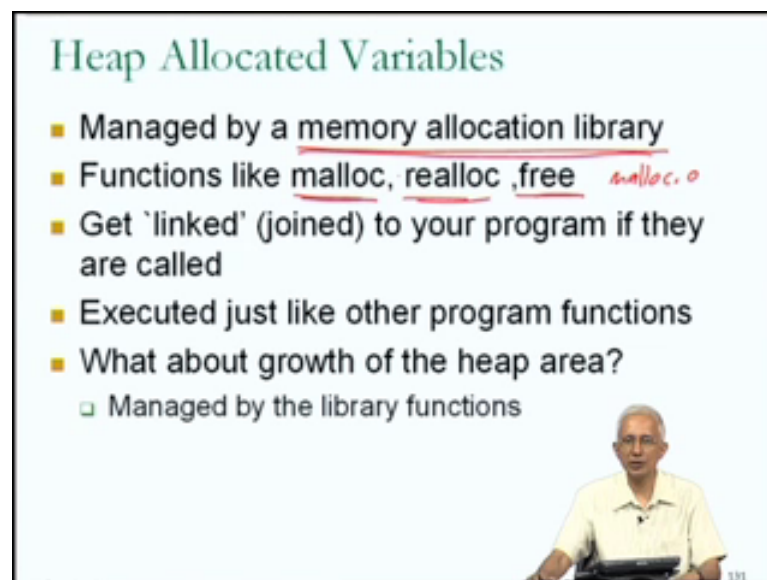
136



Now, as far as, stack allocated variables are concerned, let me just recap. The space for stack allocated variable such as function calls, function parameters, function local variables. This is allocated on function call and reclaimed on return and as we also saw, the variables themselves, the addresses are calculated and use by the compiler, relative either to the top of stack pointer, or some other base register such as the frame pointer that I refer to earlier in this lecture. Therefore, the addresses of these stack allocated variables or computed and used by the compiler, based on the picture that we had of the stack.

The compiler knows what the contents of the stack are, since the compiler is in charge of generating the instructions, which are going to cause those items to be pushed or popped from the stack. So, in some sense, one could summarize stack allocated variables by saying that the growth of the stack, the growth and the shrinkage and growth of the stack area is manage by the program, as generated by the compiler. If you are writing your own assembly language program, it would be on your direct control. If you are writing the C language program, you have no idea that these activities happening, the instructions to do this are generated by the compiler.

(Refer Slide Time: 34:15)



**Heap Allocated Variables**

- Managed by a memory allocation library
- Functions like malloc, realloc, free *malloc.o*
- Get 'linked' (joined) to your program if they are called
- Executed just like other program functions
- What about growth of the heap area?
  - Managed by the library functions

331

Now, what about heap allocated variables? Now, I had mentioned that heap allocated variables are the variables, the pieces of data, whose lifetime starts from an explicit creation point, and the life time ends at an explicit reclamation point or freeing point.

And that this is done in C programs by a memory allocation library. We know that there are Functions like malloc, realloc, and free. Malloc is used to allocate a dynamically dynamic variable.

Now, I am not telling you that these functions are actually members of a memory allocation library. So, just like they were a Mac library, there is memory allocation library. So, some where there is going to be a dot o file (Refer Slide Time: 35:05) which is going to get linked into your program and as a result of which the instructions of the memory allocation library get executed when there is a call from your program to do a malloc.

Once again a series of instructions are going to get executed through which the dynamic allocation of space in memory happens, and the implementation of that allocation is done by the memory allocation library. So, somebody writes this library, the functions of this library, such as malloc, realloc and free, and make some available to you, and you get the compile version inside the object files, such as a malloc dot o. Now, if your program uses such functions then this just like the Mac library, the memory allocation library will get linked to your program and will (( )) those functions will be available when your program executes, just like any other functions.

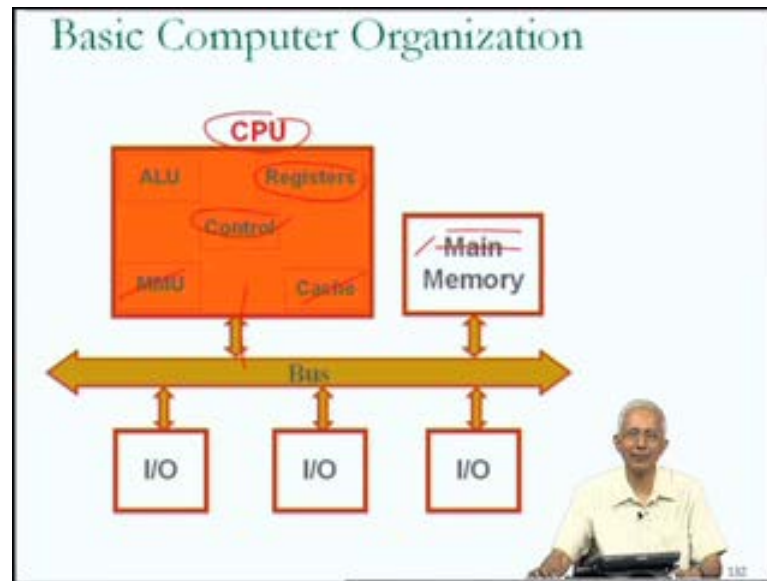
So, these are the functions, just like any other functions. So, just like the function call and return that we talked about, the function call and return operation, that we talked about would happen in connection with the malloc, realloc and free. They are functions in a library, which are called by your program when it does a heap allocation. So, what about the growth of the heap area? What you are being told now is the memory allocation library is returned to allocate space, out of memory on requests for a malloc, realloc, free etcetera, but how is the allocation of that memory actually done.

We saw that the compiler does the allocation of stack allocative space. In the case of heap allocated variables this is obviously, managed by the library functions. They may be situations where it needs more memory to be associated with it, in which case, it would get those additional space allocated to it, and then proceed, and we will be able to talk more about that only after a few more lectures. For the movement, it is clear that for programs is do a lot of dynamic, use lot of dynamic variables. These functions and their implementation could be crucial to the performance of the function, therefore one could

conceivably even think of writing one's own version of these functions if one is uncertain about the quality of the functions that are available by default.

And in a basic [\(\( \)\)](#) on data structures, one might learn simple ideas about how to write Functions of these kinds. Now with this, we have seen a simple example of writing MIPS code, we will see more examples of dealing with MIPS instructions, when the need arises, and where necessary I will revert back to the table of instructions. So, that, the meaning of each instruction and its form that it takes will be quite clear.

(Refer Slide Time: 37:52)



Now, we need to move on to our next important topic of discussion, which is, to actually learn more about... Actually, before I proceed, let me just remind you that at this point, we seen a lot about Main Memory. We have seen a lot about the CPU in the sense of we know now what the different kinds of instructions, and the different kinds of addressing modes that the CPU has to deal with that; and we understand pretty fully what the registers are; that they are special registers like the program counter of the instruction register, and the general purpose registers, and that basically the execution of the instructions is manage by the control. We have not seen anything about the MMU or cache yet, but you will do so, shortly ok.

Now, we are going to now move into a stage where we concentrate on the Functioning of the processor. We want to understand what happens when an instruction is executed. We know what happens when a program is executed. We talked about how when a program is executed the program is present in main memory and the instructions of the program one by one get executed inside the processor. So, we are now at that level where we want to understand. We will understand more about what happens, when a program executes, the basic step in the execution of program, is execution of a single instruction, which is our next target. We want to understand what happens when a single instruction executes.

So, what are the steps in processing a single instruction? Our base assumptions (Refer Slide Time: 39:15) is that to start off with the instruction is present in Main Memory, and

that to be executed, the instruction must be present inside the processor, because it is the processors that has the hardware to execute an instruction. The Main Memory just has a capability of remembering things.

(Refer Slide Time: 39:33)

**Steps in Instruction Processing**

- Fetch** instruction from Main Memory to CPU
  - Get instruction whose address is in PC from memory into IR
  - Increment PC  $104 + 4$
- Decode** the instruction
  - Understand instruction, addressing modes, etc
  - Calculate memory addresses and fetch operands
- Execute** the required operation
  - Do the required operation
- Write** the result of the instruction

Handwritten notes on the slide include:  $PC\ 1004$ ,  $104 + 4$ ,  $d(Rb)$ ,  $-4(R29)$ ,  $1000 - 4 = 996$ , and  $ADD\ R1, R2, R3$ .

Therefore, as far as the steps in instruction processing are concerned, the first step must be to get a copy of the instruction from main memory in to the processor, and a word which is typically used to describe that operation is the word, “Fetch”, so one talk about fetching an instruction from main memory in to the CPU. So, that is the first step.

Now, once the instruction has been fetched from the CPU into the memory, it can be processed. Because the instruction will be present inside the instruction register, which is one of the special purpose registers, inside the CPU. And therefore, the control hardware can examine that instruction and start processing it. But, how does it process the instruction? We seen that the instruction will be in a well defined format and so, it is possible that the first instruction is jump register instruction in which case we know that it is in that, say the **I format**. So, how does the processor hardware process the instruction from this point on?

Now, the first thing that the processor hardware must do is, given that the instruction is present in the instruction register, it must understand the instruction, and typically the

name that is given for this step of processing of the instruction is to use the word, Decode (Refer Slide Time: 40:44).

You will remember that the instruction is present in memory in the binary form, and some of the bits of the instruction are contained information, about what the instruction is suppose to do, other bits in the instruction contain information about what the different operands are, in the case of the MIPS one R format, the information about what the operation of the instruction is contained in some of the most significant bits which was the Opcode field, as well as, some of the least significant bits, which was the functions field.

Therefore, all of that information together will be necessary to decode the instruction. Therefore, while the instruction is sitting in the instruction register, the hardware must examine the appropriate bits of the instruction in order to understand what instruction is suppose to do, using the information about the instruction format. In other words, what bits of the instructions are used for what purpose? Once this has been done, instruction can be executed (Refer Slide Time: 41:38). In other words, the operation that is required of this instruction can be done.

And the term, which is given for that is to talk about the instruction being executed, doing whatever the required operation associated with that instruction is. Once this has been done, the result of the instruction can be written (Refer Slide Time: 41:57). So, this will be refer to as, "Write", typically for many instructions the completion of the instruction will involve writing a value into a destination register. Hence, the names of this particular step as write.

Now, we could look at the sub steps involved in each of these main steps, where there this four means steps: fetch, decode, execute and write. Now, what are the sub steps? As far as fetch is concerned, it is a fairly clear that in order to get the instruction out of memory, the processor must give memory, they must identify to memory, which instruction it is interested in. The processor keeps track of which instruction it is interested in its special purpose register Call the program counter. Therefore, the way to look at the first step of instruction execution is that processor sends the value inside the program counter to the main memory.

The main memory on receiving that information looks up in its memory and returns the instruction at that particular memory location to the processor. The processor then takes the instruction that is got from memory and put it into the instruction register.

So, the steps involved in fetching the instruction from main memory to the processor we can now write think of as the processor sends the program counter value to memory, memory sends back the instruction to the processor, the processor puts it instruction into the instruction register, and then this additional step, increments the program counter. Now, let us just think of the situation where the second step is not there. What would happen? The program counters, unless there was a control transfer instruction program counter, which remain unchanged.

In other words, the same instruction would get fetched over and over again, and executed over and over again, very clearly, since the default mode of execution is for the processor to finish executing one instruction and if it is not control a transfer instruction to then go on to the next instruction.

There is a need to automatically increment the program counter as part of instruction execution and in this particular description of instruction execution, instruction processing, I am talking about incrementing the program counter at the step of incrementing the program counter as happening inside the first step of instruction processing. In other words, I am choosing to describe increment program counter, as an operation that happens during the fetching of in instruction.

So, if the program counter value, which was sent to memory was 100 (Refer Slide Time: 44:34), then after incrementing the program counter, bearing in mind that each instruction is of size 4 bytes, the value should now be 104. Therefore, when we talk about incrementing the program counter, what we actually mean is, incrementing by four since the MIPS one is byte addressable and the size reach instruction is 4 bytes.

So, we fully understand what the first step in processing an instruction is. The step, which we call fetch instruction. What is involved in decoding the instruction? The next step now, as I had mentioned the decoding the instruction involves, the hardware inside the processor examining the instruction, the various bits of the instruction, as it resides in the instruction register with the purpose of understanding what the instruction is of; what

the instruction is in terms of; what the operation is suppose to be; what the different operands are; what addressing modes are used; etcetera.

Now, before the instruction can actually be operated, its operands must be available. I am having shown that as an explicit separate step inside this four step sequence. Therefore, that must be the second basic operation that happens as part of decoding the instruction, getting the operands wherever they may be. Now, in the case of the MIPS one instruction set, we know that the operands could be in registers, in which case they can be of general purpose registers or are the instruction register, in which case they could be immediately be obtained within the processor itself. If on the other hand, the operands are to be obtained from memory, as this is the case with the load and stored instructions. Remember, in the MIPS one instruction set, the only instructions, which take operands out of memory, are the load and store instructions. And the load and store instructions take their operands in a base displacement addressing mode, for which certain calculation has to be done, in order to calculate the effect the address of the memory operand.

Basically, for example, if there was base displacement addressing mode, as we have just seen minus four from R 29. Then in order to find out what this actually means, the hardware has to look at the contents of R 29, R 29 might contain the value one thousand and then it has to add to this displacement minus four. Therefore, calculates the value of the operand is 996. So, that is why there is an explicit step which is called calculating the memory address (Refer Slide Time: 47:01) if the certain addressing modes, there is a need to do some arithmetic, in order to find out what the memory address is.

In the case of the MIPS one instruction set, this is the case for the base displacement addressing mode, for more complicated instruction sets, you will recall the addressing modes, where different kinds of arithmetic may have to be done, such as the index addressing mode, where the contents of two registers have to be added, or the post, or pre auto increment addressing modes, where a constant may have to be added to a register as a side effect of the operand calculation of the memory address. But in general, in order to get the operands to the instruction, it is possible, that some calculation has to be done first, as in our MIPS one example of base displacement addressing mode.



So, I am including two basic operations, as far as, fetch is concerned. That is getting the instruction from memory into the instruction register, and incrementing the program counter. I am including two basic operations, as far as, decode the instruction is concerned; understanding the instruction, as well as calculating memory operands and fetching operands. What is involved in executing the required operation? Well this will depend on what the operation is, you will know that if it is an add instruction, then the add functionality in ALU will have to be activated. If it is a multiply instruction, the multiply will have to be activated.

If it is a control transfer instruction, something else will have to be done. If it is a load or store instruction, memory will have to be involved in doing the required operation. So, the step of doing the required operation will depend lot on what the instruction is and on case by case basis, the hardware would have to do the appropriate, initiate the appropriate activity. For the net effect will be that the instruction will get executed or **whatever the instruction had** whatever the intent of the instruction was will have to be achieved.

Finally, once the instruction, that operation has been completed, the results of the execution of the instruction can be committed, can be written. For example, if the instruction was ADD R1, R2, and R3. Then as part of execute the value inside R2 and the value inside R3 would have been added, yielding a result, which is what will be stored into R1, in the write the result of the instruction stage. So, we need to stress this out a little bit more.

(Refer Slide Time: 49:28)

**Steps in Instruction Execution**

- **Fetch instruction from memory to processor**
  - $IR = \text{Memory}[PC]$ ; Increment PC  $PC = PC + 4$
- **Decode instruction and get its operands**
  - Decode: Operands from registers/memory to ALU
- **Execute the operation**  $PC - 4$ 
  - Trigger appropriate functional hardware
  - If load/store, send access request to memory
- **Write back the result**
  - To destination register/memory

So, the steps in instruction execution we can now talk about as follows: fetching the instruction from memory to processor, decoding the instruction, and getting its operands, executing the operation, and writing back the result, which we explicitly write down as, fetching the instruction involves two operations: one is sending the program counter value to memory, memory will respond by looking up with the current contents of that memory location or sending a back to the processor, which stores that instruction into the instruction register.

In addition the processor will increment the program counter, which is same as far as the MIPS one instruction set is concerned, the same as  $PC = PC + 4$  (Refer Slide Time: 50:09). Decoding the instruction getting its operands, involves decoding, calculating the address of the operands, getting the operands either from registers or from memory into the ALU. They are required inside the ALU, since it is within the ALU that the instruction functionality is implemented in hardware. What is the executing the operation involved? It involves triggering the appropriate functional hardware and if it is load or store instruction, it involves triggering an operation at memory, in the main memory.

Finally, writing back the result, we typically involve updating destination register or memory. So, this is just a slightly more systematic description of the operations as we have described them. I do want to comment on just one thing in before closing this

lecture, and that is, the fact that we have assumed that the incrementing of the program counter by four as what happen in the case of the MIPS one instruction set, happens as part of the first step in instruction processing, is interesting. Because this now gives us a little bit of an idea, about the description of some of the instructions, in terms of the meaning of the instruction, in the tables that we saw.

And if you look back at the meaning of some of the instructions, where they was some doubt has to why PC was not used, but rather PC plus 4 was used, or P C plus 8 was used. You now understand that at the time that the instruction is being executed, if one wants to refer to the address of the instruction, one should understand that subsequent to incrementing the program counter, the address of the instruction is self will be P C minus 4. Because the program counter would have been incremented by four, before the instruction itself is executed, and therefore, the address of the instruction which is currently being executed will actually be P C minus 4.

And therefore, in looking back at the table that I am referring to, the table of meanings or the different instructions, in this slide, a little bit more idea about what is actually happening will arise. Of course, we are still not too sure about the P C plus 8, as far as, the jump and link instruction is concerned. You will recall that in the jump and link instruction at the value PC plus 8 get stored into R 31, which is still a little bit of a mystery.

So, we will have to wait for little while before we understand this. So, at this point we understand the steps involved in instruction execution, and in the next lecture, we will proceed to look at how those steps would actually get implemented in terms of very simple hardware, and the implications of that very simple hardware on the amount of time that it would take to execute any one instruction of a processor, which implements something like the MIPS one instruction set.

Thank you.