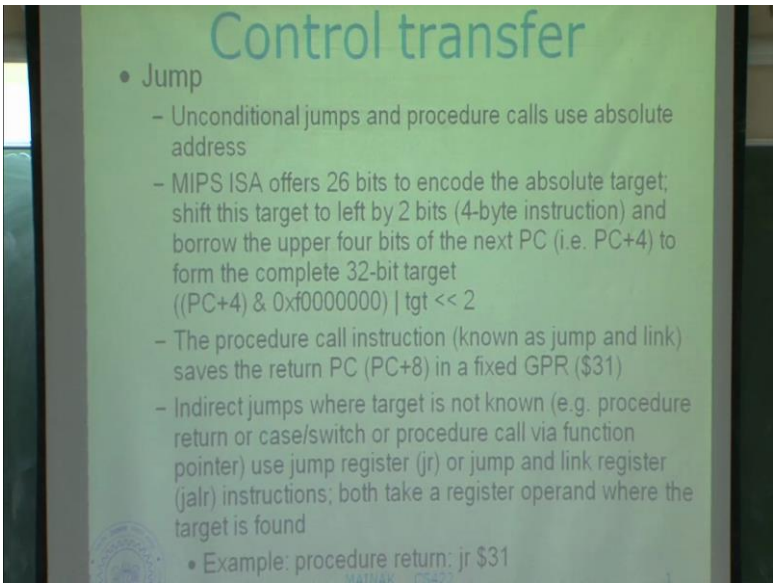
**Computer Architecture**
**Prof. Mainak Chaudhuri**
**Department of Computer Science & Engineering**
**Indian Institute of Technology, Kanpur**

**Lecture - 7**
**Case study with MIPS-I**

So, we were discussing (Refer Time: 00:20), hopefully, just to quickly remind you discuss these instructions, we talked about floating point instructions. We also looked at the memory operations, there are only two types source and also we talked about how can you synthesize direct addresses and talked about the special instruction load word left and load word right, similarly you have load word left and load word right.

(Refer Slide Time: 00:57)

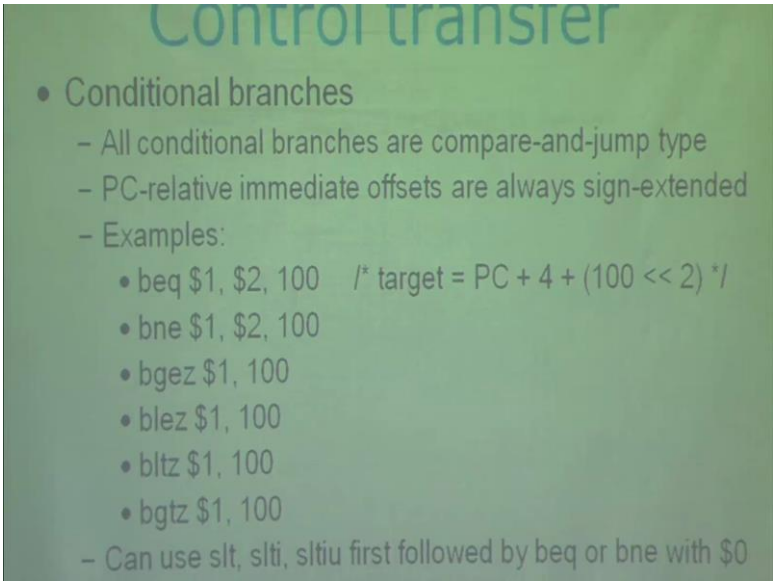



So, we started talking about control transfer instructions, we looked at jump instructions and there are two types of them, one is unconditional jump and procedure calls where the term is computed like this, the target.

This comes from your instruction these are 26 bits and your upper 4 bits comes from the next PC and for indirect jumps where the target is not known, you usually use j r that is jump register or jump and link register or jalr instructions. Both take a register operand

where the target is found, so like a procedure return in routine to use j r with 31 as your orbit, so dollar 31 result for, so any questions so far? So, there is a sheet of attendance circulating somewhere, so please sign if you are registered in the books, so what else you have in control transfer? So, of course we have conditional branches right.

(Refer Slide Time: 02:16)




So, here as we discussed earlier that there are two parts, one is your condition; that means, usually support that alright if the tradition transform to be true. So, all conditional branches are comparing and jump type. So, in which we will use these two things that is sort like we comparing one instruction and there you jump in next instruction it is not like that it is actually compare and jump. You always use PC relative immediate offsets and these offsets are always sign extended, and the reason obvious because we want to go forward or backward, it should be both unless you do sign extension, you cannot.

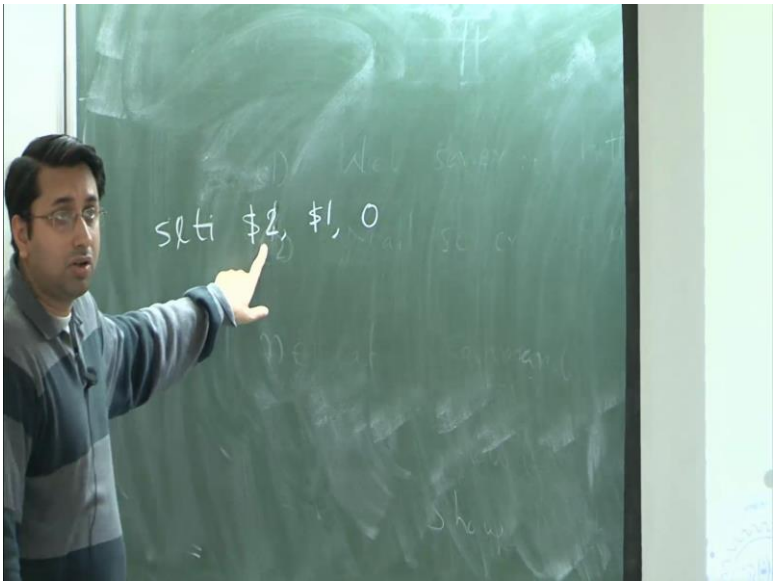
So, here are the examples this is not the example list of conditional branches, if you want you should look your dot c file that beq stands for branch equal. So, here what it means is that if dollar 1 is equal to dollar 2, then target is this PC plus 4 plus 100 shifted by 2.

So, what is that here we are mentioning the offset ignoring the last two bits because we mention the last 2 bits are always zero. So, there is no need to specify this and we do not

add it to PC, but to PC plus 4 because the same reason we used PC plus 8, does anybody remember this one? My return at this was p c plus 8 procedure PC plus 4, so we talked about this way, in fact in this side, so just remember that PC plus 4 not PC BNE. Similarly, the opposite of that branch if not equal b g e z branch on greater than or equal to 0.

So, dollar 1 if it is greater than or equal to 0, then you should go to this particular target branch less than equal to 0 branch less than 0 branch greater than 0 right. So, notice that only in these two cases you will be comparing two registers in all other cases one register gets compared against 0. However, if you want you could actually synthesize these using two instructions, for example if you want let us say one example. Suppose, if you want if you did not have this instructions right that no, not this one, let us suppose this one, b l t z branch less than 0. What if you could do is you first should compare you first would executive a set less than immediate instruction, where your instruction would be essentialized dollar 1.

(Refer Slide Time: 05:19)




Let us make it dollar one dollar 0, essentially what will do is it will set dollar 2 depending on whether dollar 1 is less than 0 or not. Then, you could actually use b e q or b n e to figure out whether the comparison passed out. So, which actually fuse these two,

so compiler more freedom in choosing instructions all branch and jump instructions have a d a slot. So, right now I would not be able to explain why this is so, but, what this means is that the instruction right after a branch of jump instruction is always executed.

So, you have a branch instruction, you compute the target suppose the condition is that you should told that out, but before going there, we will execute the next instruction. So, it is as if the effect to the branch is delayed by 1 and so on, that is why the name called slot, is this concept clear to everybody? So, we will find out why this was required shall we that this is what example were how your pipe line structure influences what compiler sees because here now compiler also will have to know this that o the instruction after the branch is always been executed.

So, the compiler job would be to fill up that particular instruction with something useful see instruction right after the jump or branch is always executed. Now, that explains why you should add into PC plus 4 uses as if the branch instruction was actual at PC plus 4 not at PC. So, the procedure regard address should be PC plus 8 because a procedure is a PC procedure call. Then, PC plus 4 is always executed, you should come back PC plus 8 what is the procedure completes, any question?

Student: Sir other instruction PC get.

Sorry.

Student: Instruction apart two branches.

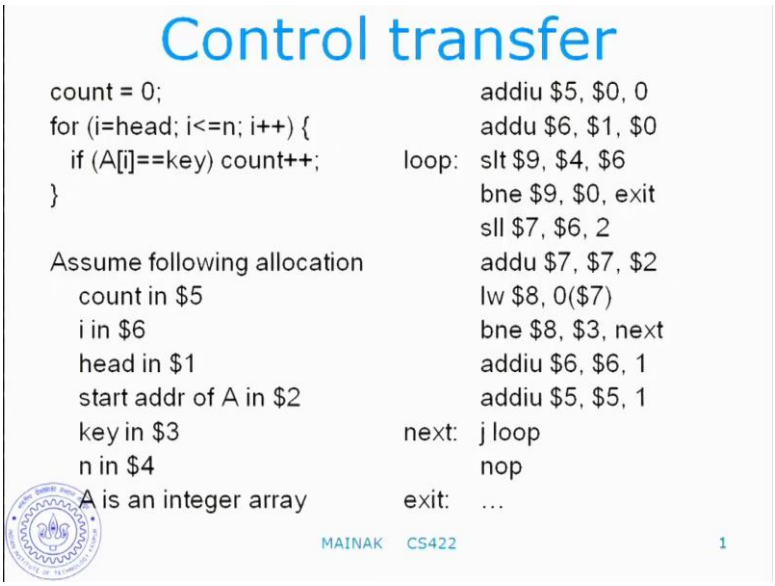
Yeah

Student: PC get subjected.

Well, we are we are not even thinking about because we do not have any motion of looking at program explain what the instruction should be? How long it takes? It does not happen, execute instruction 0 time the updated PC everything happen will bring in

time will try to implement the instruction sector that is why the time will come into picture any other question.

(Refer Slide Time: 08:24)




So, it is an example let us try to work it. So, here we are doing a search essentially building a histogram. So, count is initialized to 0 and I have done over which starts at a head index head and we done over t to each end looking for particular key I will count how many times the key activates that itself very basic part of building a histogram. So, just for the sake of the example I will show you the assembly program, so you follow what really happens.

So, let us assume the following allocation that the compiler is done suppose count is in register point alright the loop index i is in register 6 head is register one the start address of a is in dollar 2 key is in dollar 3, is in dollar 4 and a is an integer array. So, compiler has already allocated these things and just setup these particular scenarios now where now let us see how the loop executes. So, first what is this doing can anybody explain the first line dollar finds count initializing to 0.

So, dollar 0 is to 0, so it is just adding 0 to 0 to get 0. So, that is initializing dollar5 if what is this doing dollar 1 is head and dollar 6 is hand exactly. So, it is doing this i equal

to head and moving here to I the inside the loop it first checked first compiled dollar 4 and dollar 6 and if dollar 4 is less than dollar 6, we set dollar 9 in 1. So, let us see what is this dollar 4 is n and dollar 6 is I. So, what is it doing and if that is, so then it checks if dollar 9 is not equal to dollar 0 that exists. So, why it would be not equal it should be one in that case dollar nine right if dollar 9 is one; that means, we have already reached the loop bound.

We should exit and jump exit and do something alright less than the based on less than equal i less than or equal to n dollar 4 is 10, no it is just liberating this comparison that is it is just liberating this particular comparison. So, that is why you get this b n e this clear to everybody, so next what will do we. So, remembered that this instruction is a delay slot of this. So, it will always be executed on this case it does not really matter because this should be taken exactly once.

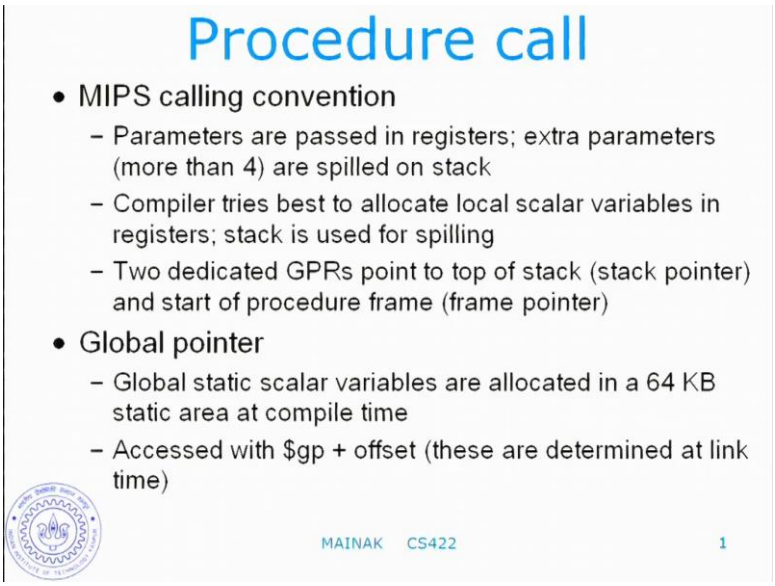
Otherwise, in all other cases you will actually just continue, so dollar 6 gets shifted that by 2 positions, so what is dollar 6? Since I shifts left by two puts in dollar 7 so that sequence the dollar 6 multiply by 4. So, we try to understand what it is doing let us see how it uses dots, then we will understand it adds dollar 7 to dollar 2 and puts it in dollar 7 dollar 2 is what base of 8, so what is dollar 7? Now, can anybody guess, so then it does a load loads from dollar 7 into dollar 8. So, now, we have dollar 8, any checks if dollar 8 is not equal to dollar, 3 what is dollar 3? Dollar 3's key if it is not equal then it goes to next where it just takes me back alright is essentially spot there is no a spot.

So, it just goes back to the next operation remember that. So, it will be executed here it actually increments i. So, i plus that is a very useful way of using the delay spot because it is, otherwise you will be actually wasting another instruction. So, you go back here otherwise you continue and you increment count here by one because if you here; that means, you actually got the gone the key right and then anyway you come here go back. Jump loop that is an unconditional jump which takes you here j means it is an unconditional jump any question on this not that is no operation, so can anybody guess why is it here exactly right.

So, this is a delay slop of jump right and compiler do not found anything to fill in this

slot. So, it just looks a nop slot instruction. So, this is why the compiler becomes important the compiler should be able to find something useful to slot. Otherwise, in the end of this case compiler finds anything, so it has to put a nop. So, you need not to execute the instruction here this might change the correct execution any other question here.

(Refer slide Time: 15:03)



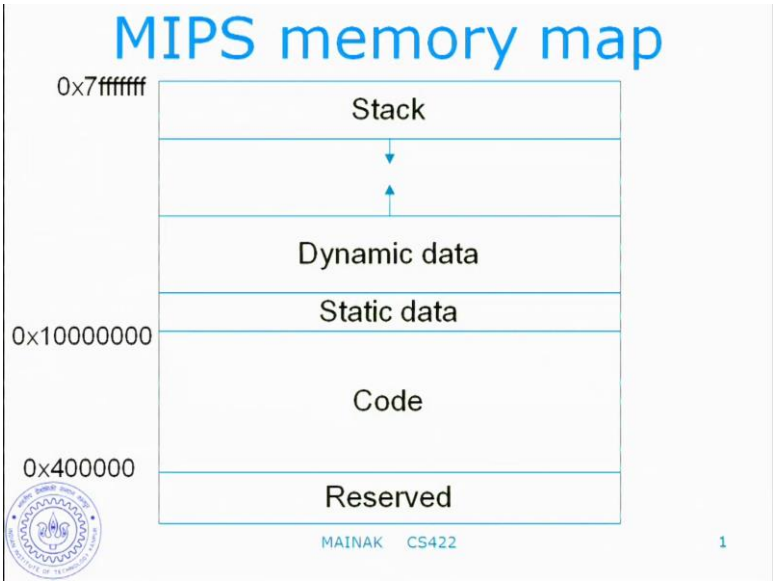

Procedure call, so every instruction set comes with the calling convention that is what really happens on the call specifically tells you how the parameters should be passed number one and how the stack should be prepared for the procedure. So, in this case in which parameters are passed in registers. So, if you have four parameters you can pass all of them to registers because which results four arguments registers. Of course, you cannot procedures that synchronize more than 4 times if you have 4 parameters, they are spilled on the sack passes to the sack compiler tries best to allocate local scalar variables in registers stack is used for spilling.

So, in a procedure if you have local scalar variables it is the compiler will try to put them to the register, but of course if there are two dedicated general purpose registers point to top of the stack or stack pointer and start of the procedure frame called a frame pointer.

So, procedure frame is of the stack pointer, so the frame pointer points to the start of, so will see one example. So, there is something called a global pointer where global static scalar variables are allocated. So, this is a 64 kilobyte static area at compile time then the global pointer points to stack of that region alright and it is always accessed with dollar g p plus some offset. These are determined at link time because a link are actually establishes exactly where the global segment will resigned, so these offsets gets determined only at the mean time.

Is it clear everybody? We have a stack point register we have frame point register, we have global pointer register, and we have four argument register and local scalar variables to registers, otherwise stack.

(Refer Slide Time: 17:18)




So, look like you have the address grows in this direction we can see that right this is 0 in the highest address this portion is reserved for register right this is usually called the test review where your code lies. So, this is where your program will l be loaded in memory then comes the static data. So, static data means the global data and whatever local variables you have sorry this is just the global the global gate that is a global pointer data.

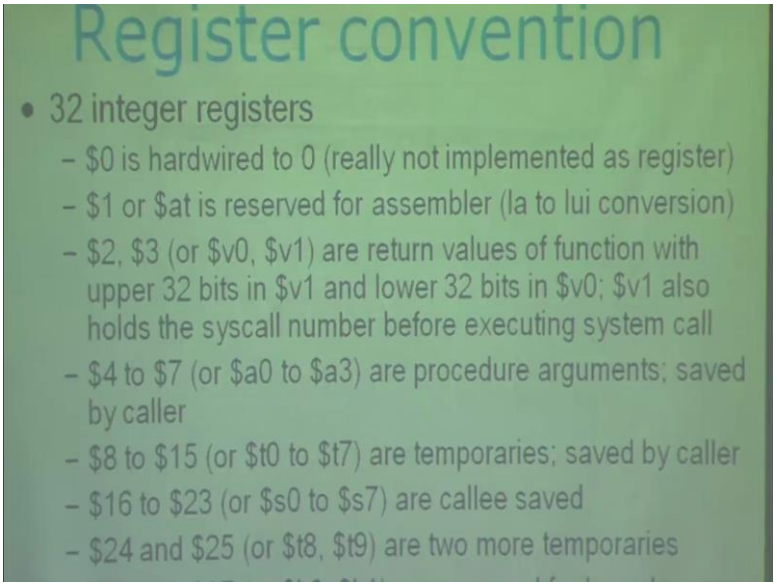
They are dynamic data this is gowned up and stack which goes down I repeat may be what was happened if the stack. So, which is why that examines what happens when they is it clear to everybody. So, user see we can calculate actually if we calculate that particular number it will come to about two gigabyte. So, these reserved region although I say that this is. So, what it really contains it basically the interrupts hangers.

So, often this is also called a whole theme is the user space access this region you have to go through to the operating system to access this region. So, here your what reserved is basically the interrupt. So, upper two g b is the cordial space that is where you actually Scordial code resides cordial code cordial data all these things. So, that the upper 2 g b the lower 2 g b is user space at just show that on processor reset execution start at that particular address that you have to decide the program gets loaded with that particular noise and that is were in stack.

(Refer Slide Time: 19:36)

A stack grows downward while heap grows upward, so if you actually take this constant and try to find out why it actually decides will finally, decide you in upper 2 g b variant. So, it is really in the kernel space that is why we actually start from kernel space kernel takes up your control and finally, hands over to whatever program you run. So, where you boot the issue the program has to break something right that is why the first instruction we must execute and that is that address this is the first instruction. Usually, you will have the boot stack load of them. So, this is the alright from its not off course not for every processor, so why these are the instruct convention?

(Refer Slide Time: 20:36)




You talked about this four argument registers, so here is the four register convention from its, so there is no convention specified from. So, dollar is hardwired to 0 its actually not implemented on the register whenever you ask for dollar 0, the hardware will actually provide for all 0 constant, dollar 1 or dollar at is reserved for assembler. So, often you will find that well in this course although which if you ever get to use a compiler and if you try to recession that compiler code you find it has seven lemonics called la is actually on bridge instruction.

So, it is often used to load constraint absolute address that is what it stands for load address. So, the assembled job is to convert those to do whatever sequence of

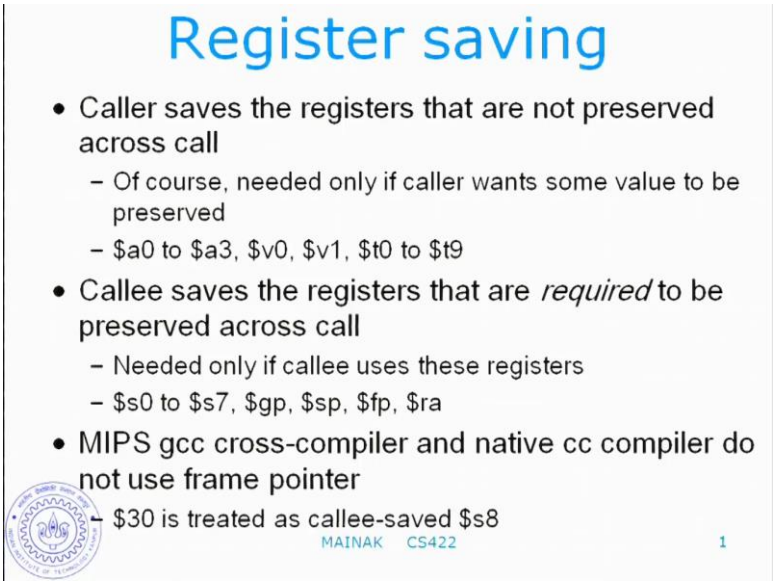
instructions that we discussed last time. Dollar 2 two dollar 3 or sometimes called dollar 0 and dollar V 1 are written values of function with upper 32 bits in dollar v 1 and lower 32 bits in dollar v 0. So, we can recover 64 bits you done 64 value dollar V 1 also holds the system call number it could execute your system call, so dollar V 1 has a dual points. So, in one case dollar V z e dollar V 1 together will specify the withdrawn value it is a 64 bit value if it is a 42 bit value.

Then, only the dollar v 0 is n alright and dollar v 1 used to hold the system call number before jumping to the operating system execute the system call dollar 4, dollar 7 are sometimes written as dollar 0 to dollar a dollar 3 procedure. Arguments said by the caller to be obvious they have to be said whether caller because caller is actually the reset the arguments. So, before saying the arguments if you say the argument registers and set them dollar 8 to dollar 15 or called t 0 to t 7 are temporaries saved by the caller.

So, these are the callee or the procedures is free to use this registers that is what saved by the caller dollar sixteen to dollar 23 or dollar a 0 to a seven are callee saved registers. So, what that means, is that caller can actually keep the values in this registers and change target forget about them without even about them because it knows that if the callee wants to use this registers it will be said. They actually happens on stack its save on the stack and copy on the stack 24, 25 or t 8 and t 9 are two more used for the same purpose 26, 27 or k 0 k 1 are reserved for kernel. Here, the kernel k 1 requires in register which can use those 28 is a global pointer 29 is a stack pointer, 30 is the frame pointer and 31 is return address that is your complete register map purpose alright here any question.

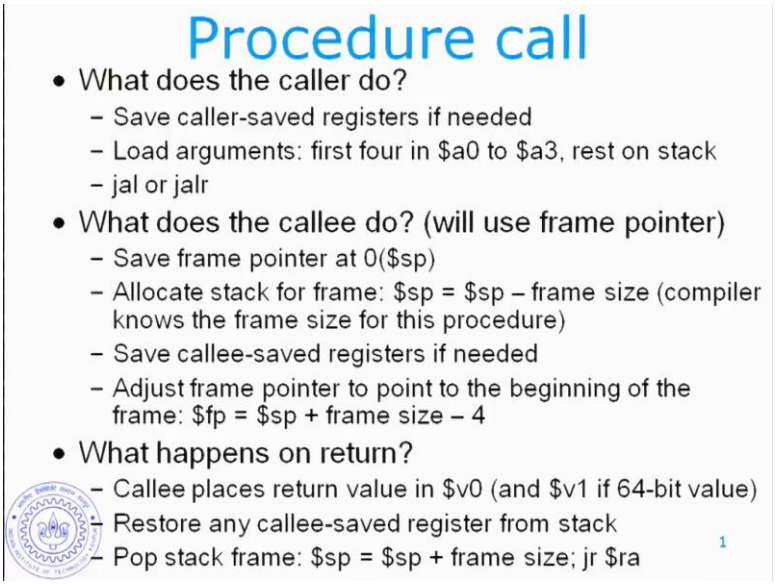
So, this is very useful because for example, in operating system gets the system call it knows where to look for to get the system call number that is the order that purpose are not mentioned here convention actually. So, the limit on register setting caller save the registers that are not reserved the calls off course there will only caller once of value to be provoked and these are the ones that the call received to this mention callee saves the registers that are required to be preserved. They are needed only if callee uses this registers, so these are the callee said registers, so these are of extreme importance.

(Refer Slide Time:24:50)




If you do not save this you able to mess up the execution, so the global pointer later be restored on the procedure calls stack alright because which your tested calls indirectly should must be 0 the MIPS gcc cross compiler, native cc compiler do not use frame pointer. So, that is dollar 30 is treated as callee saved s 8, so third point is usually not use question.

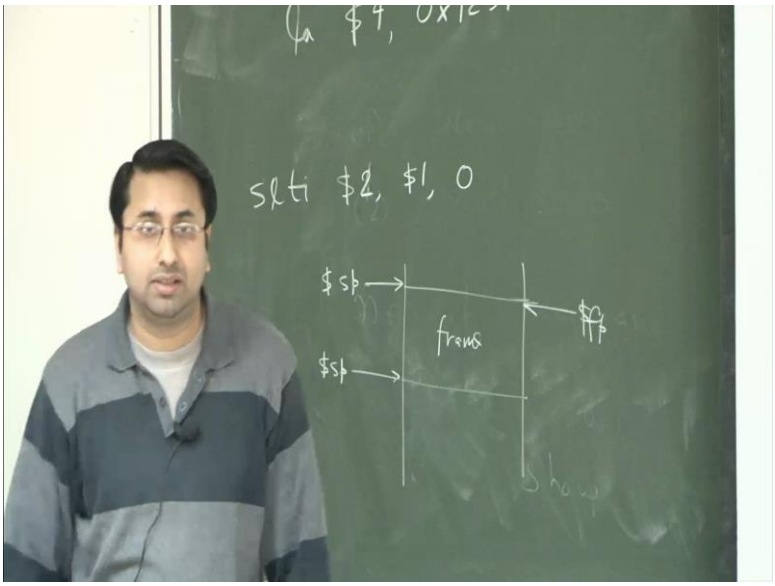
(Refer Slide Time: 25:27)



Let us see how access a procedure call, so four arguments can be passed in four registers the remaining wants to move on the stack, so what does the caller do? It saves the caller saved registers if needed it loads the arguments first four in dollar a 0 to dollar a 3 rest on stack and then call jal or jalr. So, jal is a direct procedure call jalr is a indirect procedure call alright jalr what does the callee do? So, in this prescription, we will use the frame pointer because it is easier to explain it will first save the frame pointer at the top of stack that is 0 dollar sp alright allocates stack for frame.

So, if you essentially what you do is that remember the stack pointer down right. So, if you bring the stack pointer down by frame size now entially what will do is we will use. So, stack pointer was here right if you delete down here. So, use this area for the procedure that is the procedure frame. So, compiler knows the frame size for this procedure, so it can actually calculate what save the callee saved registers if needed before you start doing anything adjust the frame pointer to point the beginning of the frame. So, what is the beginning of the frame this stack pointer at the frame pointer subtracts 4 why subtract 4.

(Refer Slide Time: 27:05)

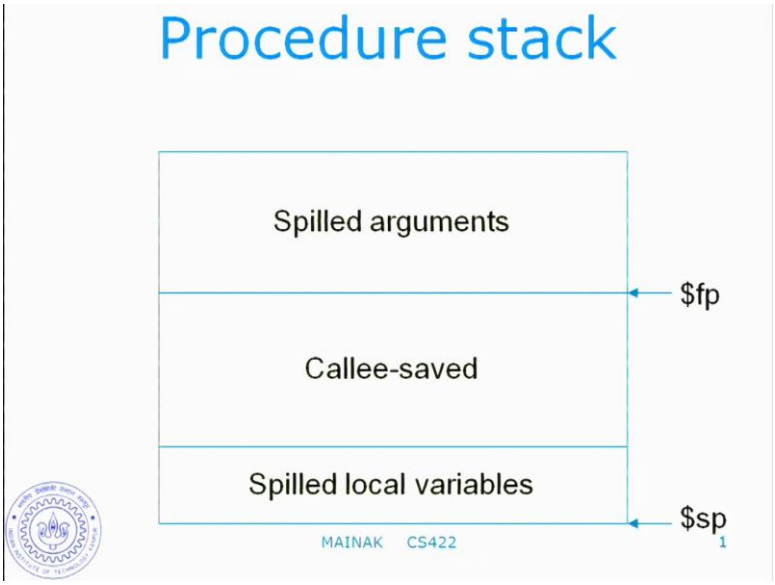



So, our stack pointer was here right this is my frame. So, I bring the stack pointer down here right and data want the frame pointer to point to here, so thus I can use this.

So, what I am doing is I am leaving of the top four bytes why is that there at that location finish point? Exactly, I just saved the frame pointer there right at 0 dollar s p I cannot overwrite there. So, that is why I have to leave the four bytes at the top is it clear to everybody, what happens? On return by here, somebody was mentioning about return address keep in mind that return address is not on the stack using the register it leaves; that means, dollar thirty one dedicated to alright. So, will you do a jump and link part of this instruction is to put the return address into dollar 31 alright what happens in return callee places the return value in dollar v 0 and dollar v 1.

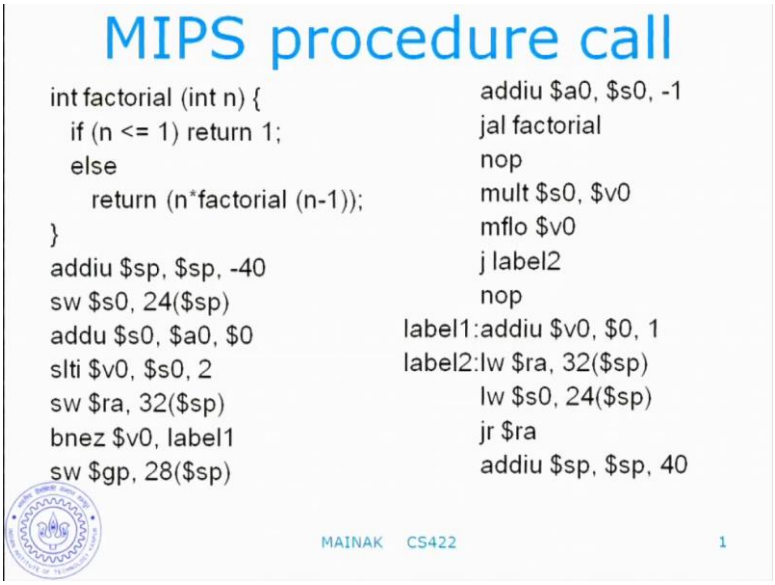
Exactly, that is the very good question, so the answer is here callee saved registers have dollar r a. So, if the callee makes one more function call; that means, it is going to verify dollar i in that received dollar. So, we will put it on the stack and then put the new. So, yes you are right if you are wasted calls the return address will actually direct step up into alright. So, what happens on return callee places return value in dollar v 0 and dollar v 1 if it is a 64 bit value restores any callee saved register form stack pop stack frame. So, s p plus s p becomes s p plus frame size. So, then execute j r dollar r eight dollar r eight contains return address. So, this instruction will modify the program counter to be equal to the contact of dollar i that is it ok, any question on procedure call?

(Refer Slide Time: 30:10)



So, this is what it looks like I just showed it to here. So, these are your arguments coming from the caller. So, this is after doing all these adjustments stack pointer was pointing here was brought down here after subtracting the frame size. So, frame size a frame contains callee saved registers and spilled local variables the compiler will have the responsibility calculate in this example how much is needed. Usually, the compiler actually overestimates this you will find in examples that will show you and then if you bring out the frame pointer here the procedure returns the stack pointer will brought out again here. So, that the caller sees exactly the way the stack pointer was going towards callee during the procedure call.

(Refer Slide Time: 31:07)




So, here is an example factorial with the recursion, so this is the c code right any question on the c mod say greater less than equal to 1 that is when you start and folding the recursion. Otherwise, will return into n factorial into n minus 1, so here is the code, so let us try to understand about, so the first one, So, these what does not happen if they require. So, this is facing directly. So, I essentially compiler and these, so we do not have any frame pointer because which component does not use frame pointer for first one adjust your stack pointer. So, it calculates that as requires fourteen bytes for this particular procedure then what do I do I store dollar a 0 into twenty four s p.

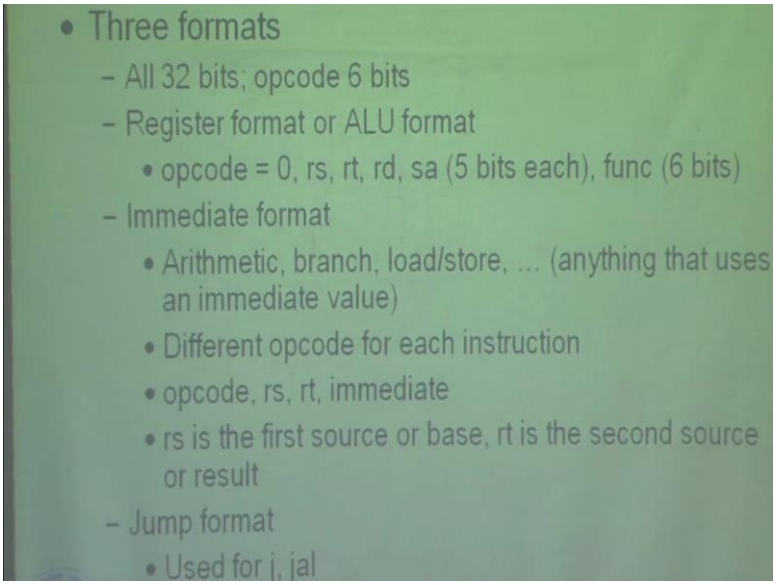
So, we will see if what it contains currently that the whole intention of doing this is that s 0 is a callee saved register and the function wants to use s 0 for some purpose that is why it is saving this register on the stack. Then, it moves a 0 to s 0 right this particular instruction does that only it adds 0 to a 0 a 0 is the what is a 0 what is a 0 n right. So, dollar a 0 should contain n. So, I am just moving n to dollar a 0 then I compare s 0 to two set less than immediate if dollar s 0 less than 2, which is exactly same as doing this n less than 2. So, I store the value in dollar v 0 alright dollar v 0 is a important value you remember that.

So, a less than equal to if any if n is less than 2, then the return value is 1, then if not and here well I have not yet compared, I have not yet checked anything. So, now, I come here I put return address on 32 dollar sp, I save it on the stack, and then I check what happen to v 0 if it is not equal to 0. That means, this comparison will actually and I jump to label 1. So, what do I do in label one will actually this was not really needed it just moves 1 to dollar v 0 and then forms from stack forms s 0 from stack returns. Here, it adjust the stack pointed the delay slot of the return alright is this part is to everybody from here to here to here to here.

So, actually in this path this is not needed, but, we are just preparing for this particular call that is what you will write is this clear to everybody? This is the genesis of this synthesis of this particular statement right this is this one alright also notice that this is going to be in the delay slot. So, it also saves global pointer on the stack it is again preparing for the call actually. So, notice that the compiler actually prepares itself for the common case that is in most case there will be a point it messes that actually. Then, it does decrements s 0 prepares this argument minus 1 alright and then calls factorial jump and link factorial nothing to put the delay slot it puts some more the return will be here.

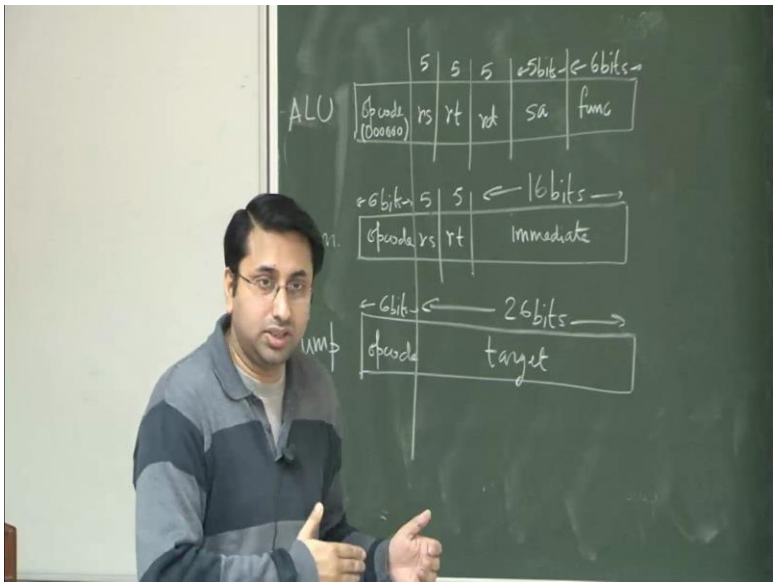
The return will come back here do the multiplication s 0 with v 0 v 0 is the return value of factorial multiplies that with s 0 s 0 means n of this point. So, it computes these for moves the value from low to v 0. So, since I have a return turn of integer knows that the thirty two bit value does not even clear about of multiplication and then jumps to label to this is essentially preparing to determine, so that is your command any question.

(Refer Slide Time: 35:43)




Last slide where done, so in summary we have three formats of bit instruction.

(Refer Slide Time: 35:15)




All the instructions are 32 bits in size op code is always 6 bits the op code exhibits upward and this is 6 bits. So, how many instructions do I have? So, what are the three formats the first format is called the register format or au format.

So, let us suppose this is my a l u format. So, this space will be always 0 alright it dedicates the lower 6 bits to specify a function. So, you can add 64 different ALU functions. There, you have to specify your source and target registers, so, you have r s r t ad and s a. So, you have that s a is the shift amount, so remember that you have this shift left logical shift right logical shift and send instruction where you use mention an on stand shift amount right the upper 6 byte register are 32 bits it is a 32 bits is a right shift amount cannot be more than 32.

In fact, it has almost 31 senior to 31 that is why it is 5 bits the shift amount and then I have the destination register that is r d which is also 5 bits and then I have two source r s and r t that is the ALU format included. So, what is the thriller for the decoder decoder looks at the opcode sees all 0 s in instruction looks at functions knows what to do extract the sources with the computation could result in register rt. These are all 5 bits because which sets 32 into general registers there is only the integer instruction point by the way any question on the ALU format.

So, this are the register instruction format because its op codes are real registers the immediate format. So, this one takes an immediate constant that is part of the instruction. So, this can be arithmetic instructions like your add immediate right or your or any immediate it can be branch instructions were we use a immediate fill for specifies the branch offset it can be load store where it specifies the displacement in immediate fill. So, anything that uses the immediate value because it opcode for each instruction, so format that you know it is not like this.

So, the format is that you have to opcode here which is 6 bits then you have r s and r t which exactly saves positions five bits each. So, how many bits are left? 16 bits it is a immediate. So, the decoders looks up the opcode and lowers what this instruction is like it could be a add instruction it could be a branch instruction it could be a load instruction and so on. It takes the immediate accordingly either 0 extends or size extends and whatever r s is the first source or base and r t is the second source or result, so what does this mean? So, in this case is when, so in most cases this immediate will be one operand right and we will require one more source operand and that will come from r s that is the first source or if it is a load instruction r s is the base address.

It is a store instruction r s is also the base address r t is the second source can you think about the instruction where r t would be the second source an instruction that has immediate value. So, I just mention the immediate value always should already be an operand for this instruction then r s is the first source one more source. So, in most common case this r t will be the result like for example, if you have an at I instruction you add that immediate with the content of r s put the result in r t in what case you have the second source in r t.

So, off course that to tell you that in such instruction there cannot be any such is there no place to store it actually exhaust d r s is the source and r t is the source. I do not have any other register for this source what is such an instruction what instructions do not have register value to be included. Do not mix up multiplication with this multiplication is a different thing multiplication does not have the immediate value some instructions that uses the immediate value, but does not have a hazard that goes to the registers jump format is the last one. We haven't yet gone there, but yes procedure calls are also in the last format conditional branches exactly, thank you very much conditional branches.

For example, if you take b e q it will be comparing values in r s and r t and using the immediate spot the branch target offset. So, this is the instruction where the outcome is not getting back to any register when the outcome is used to update anything else is everybody is clear about that b e q and b a. These are the two branch instructions that it consume both the source instruction, there is no target. Any other instruction that would not have any outcome feedback to any register what about what will they do I will tell you that the store operation falls into this category, can you tell me what it will do? Actually, how will calculate the address that itself already there in the slide r s is the base alright v n that is what you think r t will be the offset how many of you think.

Very good thank you, so here address will be r s plus sign extended immediate and r t is the value that is inside the store by the way we are not really storing r t. Although I am repeatedly saying storing r t where I am storing the value they register r t right r t below than the 20. So, I told 20 register stored that value alright that format is called the jump format. So, this is immediate format, so this is used for direct jumps like j and jel these are the only two instructions that fall in this category.

Here, you have a opcode 6 bits and a 26 bit target see you can jump across 64 many instructions. So, that is your bits, so now, you can calculate how many instructions the integer side has right this is 65plus 64 right. So, you have one 29 instructions actually you have these are not use. So, I just want to re re synthesize the fact that gives the risk. So, you have the set of instruction as now as you can do pretty much everything more it is not like you should provide very comparative instruction one million instructions. You support the most common instruction commonly used instructions the remaining one, you should able to synchronize using this.

So, you are saying that if you have the constant more than 16 bits that what we discussed last time right how to synchronize larger constants. So, should I start now what do you think yeah only five minutes left. So, there is a sheet of attendance circular around please sign before you leave. So, next time talk about binary used for in our home works you understand, so that is what we do in the next class.

Thank you.