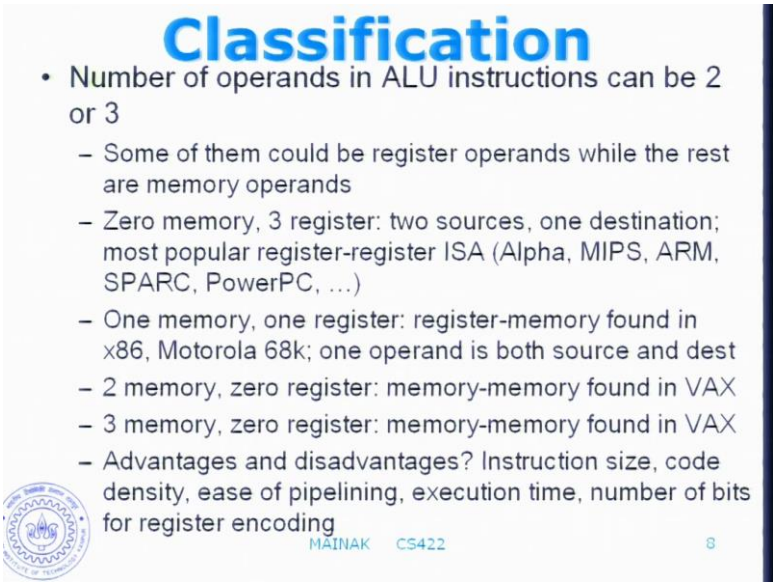
Computer Architecture
Prof. Mainak Chaudhuri
Department Of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 3
Instruction Set architecture

(Refer Slide Time: 00:15)



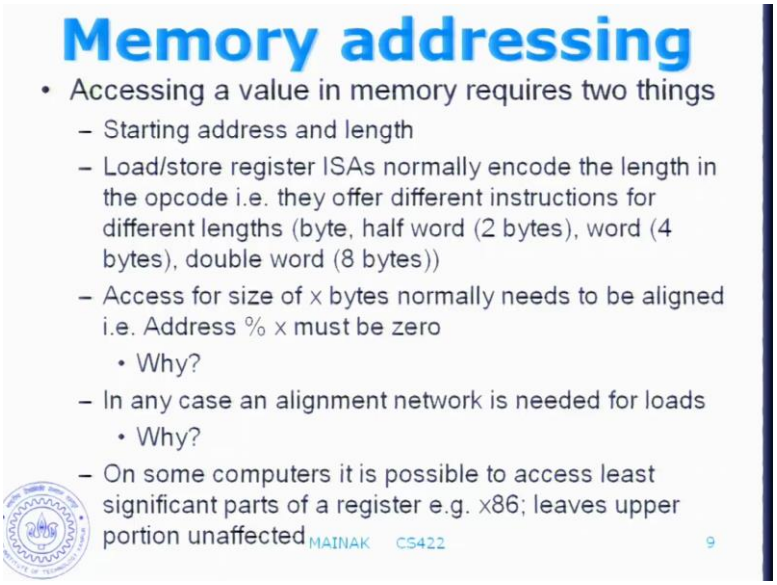
We were discussing instruction set architecture. So, last time we talked about basic definition. So, last time we stopped here, classifying instructions based on number of operands. So, we looked at. See its operands (( )) one memory, one register; 2 memory, 0 register; 3memory, 0 register. And the main point here was that, if you want to have a memory operand, the obvious thing that you have to do is that, you have to encode at least one memory address to the instructions, and a memory address is typically large, depending on the size of address space, that may 34 bits 64 bits. So, that tricks speaks and that makes (( )). Whereas the machine number of registers could be, very small normally small 32 8 16 of that range, number of registers expose to the program.

So, in that case the register address would be just 5bits 4 bits. So, if we have register operands your instructions will going to be smaller in size, or it is could be operands; however, an instruction which memory operand has an advantage that is, if you look at

one instruction, that is. Let suppose that is doing an addition, adding up two memory operands.

Let us take this one. So, adding up a memory operand to point register operand, and putting the real result we let say memory operand. So, it has two memory operands instruction and one register operand. So, here what you actually doing, is you are doing three operations together. You are coding a value from a memory address, in two year processing, some register are internal register, adding that internal register value with this register value. Putting it somewhere inside the processor, again some register that is not expose to programmer, and then storing that register back to memory operand. So, if you want to do that, in a machine which does not have memory operands, we have to assume (( )). So, that would affect your code density. There is a clear (( )) between instruction size and code density, in this classification. And we will further see certain other aspects; for example, if your memory operands usually it is very difficult to pipeline, so we will see why that is so. And if you off course have impact on execution time, because we say it that execution time is a complete parameters, the cycle for instruction, number of instructions and a cycle time. So, we already seen that number instructions, is going to get effected, the code density, depending on what encoding you actually choose. So, number of bits for register encoding. So, as (( )) we will look at these (( )) more.

(Refer Slide Time: 03:41)



So, let us start with memory addressing. So, this is a very important component of every instruction set architecture. By the way somebody asked last time, I will just reiterate that once more, that what comes first, is it the instruction set of the architecture. So, the first thing that you do when designing the machine, is the instruction set architecture that is (( )) instructions, might processor is going to happen. And your processor is essentially an implementation of the instruction set. We will try to implement instruction set, what you get is essentially the process. So, this is what decides; what your processor will do, what your processor will look like. So, as I said last time, if you make a gross mistake, in the defining their instruction set, there is no way you are going to have a good person; however smart designers you may have in your (( )). So, memory addressing is an integral part of your instruction set, which decides how you to address the memory. So, accessing a value in memory requires two things; one is the starting address, of the value let you want to add, and the how many bytes you want to add, length of the byte. So, notes a register instruction set architectures.

So, just remind you what this is, you might have forgotten. This one allows memory operands, only load store instructions, does not allow in memory operands in ALU operands. So, ALU instruction will have one memory operands, one register operands. So, the load instruction will take the value from the memory operand, and put the value

in the register operand. A store instruction would similarly take the value from the register operand and store the value back to the memory operand. So, these architectures normally encode the lengths in the opcode. So, what this means is that, they offer different instructions for different lengths. For example, if you want to access a byte in memory, for that they have the separate instructions. If you want to access a half word, half word is two bytes. So, these particular types a taking directly from MIPS; varied from person to person.

So, here I define half word would be two bytes. If you want to access half word, you will have another instruction. If you want to access a word which is four bytes to the one more other instruction. If you want an access double word or eight bytes, line of one more instruction. So, tied becomes implicit in the opcode itself. So, I will not (( )) the data, how many bytes you want to access. and access called x bytes normally needs to be; that is, if you are trying to access that is the x bytes starting at address, an address is must be the x must be the 0; why. Any idea why should be. We solved an aligned access, and if this condition does not hold that are called the nonaligned access. So, in most processors, at least in some processors you will see that, this is one required, the address module x must be 0, which means if I want to do a load word for example, I can only start at an address which is multiple of four, I cannot started an arbitrary bets. Any clue, why they should be. I am not saying that these are the requirement for every processor. In fact, one of your most popular processors do not require this, but this simplifies matters a lot, why is that, any idea.

Student: Reduces the size of the memory.

Did you say size of the memory.

If p 2 the exact 4, then we need not consider the last (( )) to decrease the instruction. I see, but that would be the. She has saying that you gets the power of two ten, I can decrease the size of the address in the instruction; that is a good observation, but we will see that we can solve that problem otherwise, some other way; there are other ways to solve this.

Student: Memory is in multiples of 2.

Memory is in multiples of 2.

Student: Suppose a power 10.

Is that part of multiple of 2, multiple of 6 will also. I am sorry 6 will also multiple of 2

Student: Power of 2. So, it was supposed that the end somewhere around, suppose that 1024 bytes is the memory, if you are accessing the 1021, and if you are trying to access 4 bytes, in that case it may fail. It means if we give a restriction that only multiples of 4 should be allowed, means when you are accessing.

No I will allow all of these. A load byte instruction can start anywhere; a load half word instruction has to start even address. A load word instruction is start at addresses multiple of 4, double word will at will start.

Student: If we take the case of code, when we should access for starting from1020 like that, if you starting from 1021 then it may fail.

So, what you have suggesting is that, there is something that is power of 2 in memory, which is why, if we assured this, then we will guarantee that, we will never straggle that boundary with all accesses will finish by that boundary. What is that power off, what is it actually; 1024 you said you give your answer what is it.

Student: Size of memory.

Size of memory, why is that important size of memory. I give you 4 giga byte of memory.

Student: That is also to define (( )).

Yes that is the power of 2, but I will have a problem only at the end. This one access may which is why I would here strict that, anybody else. So, he shows he has brought of an issue with part of two. Did he have to do something with part of 2.

Student: In fact, if we read it like 8 bytes and the remaining are 2 or 3, we will be unnecessarily doing that, it states what it would help us that it would break it down to the last chunk of those 3 4 bytes then we will go again block down to 4 2 and 1 simultaneously.

What is this 8 bytes.

Student: So, little if we have a large data to read.

Last means.

Student: So, instead of doing it last, we would have it first, if we have something like 31 bytes.

So, first to do one bite would have thirty bytes to read, and go with.

31 can be decomposed in various other ways

Student: It would help us decompose that, because majority of times during that read of chunks of 8 bytes, but it when cannot read but could reduce it.

That is fine, but has it has been this one.

Student: Because fp read, so if we do not do that then, we will do unnecessarily doing the 8 byte reading, even if we do not have 8 bytes

Can you give an example. I am not following saying I am sorry. Can you give an example concrete values. So, at not enforcing these two actually break your, whatever

you are suggesting.

Student: Something left.

No what you mean by left, that is one I asking end of the word that something or what is it, what is meant by left, nothing beyond that.

Student: they sometime beyond that but we would be unnecessarily reading that.

Left to what, what is that, after 3 bytes what is going to happen. 5 bytes, but you have to define, what is meant by left.

Student: that is the garbage value for us.

Beyond this point, what is this point.

Student: That is the load instruction that we were getting. So read this much data.

How much data? Give me numbers how much data you have to read, I am just trying to get an example from you. Give me some number load 4 bytes 8 bytes, what is it. No what I am not able to understand is, what is it that you mean by saying 5 bytes are left. Left in what. Do you see what I am asking... So, can you articulate that what you have thinking in your head. There must be something, why you are saying this, what is it.

Student: We would have these only instructions, so I thought that (( )).

Anybody else, yes.

Student: Boundary value of the memory block.

Can you define what a memory block is.

Student: Typically what is a block can you give a size.

In kilo bytes, when you request memory some data, how much data do you get back usually. Now, I mean what is it. Who is request.

I understand the boundary that you talking about is the memory block. I am just trying to asking you, what is the typical science of block. What determines memory block.

No that is the file system.

Student: Word size.

It is that right. Is that so? Normally where do you look up this data, when the CPU tries to access data, why does it go first? Is they are something called the cash block size. So, that is what the memory will return, and a cash block size is typically 64 bytes, 32 bytes 128 bytes, 256 bytes, not more than that. So, that is the boundary that we are talking about. If I do not enforce this, the chances are that, and may end up accessing a particular piece of data in instruction, part of which is in one cash block, part of which is in another cash block, which actually requires two memory accessories, with one instruction, which is not a very critical. So, here if you mind that, since we are allowed x take only these values. There is some assumption about block size here, which is the part of 2 and that is why we need to enforce this. So, if you design a new machine where x would take other values, then this particular condition might change. Is it clear to everybody.

So, these are the called aligned access; x 86 does not actually enforced aligned access, Intel processors actually can handle aligned access as well, but we talk about one processes that is MIPS, which actually does not allow aligned access, we have to do all aligned access. And this is compilers job actually to produce address which are aligned. However, in any case, an alignment network is needed for loads. Can somebody decrypt this particular statement. So, I just said that, this particular condition holds. if you have a load instruction of 4 bytes, we know that the address will be at a boundary of 4 bytes, it is a multiple of 4, we know that, but there I say that well, even that I need an alignment network. What is this network doing actually? What is the meaning of an alignment

network? Anybody guess.

Think about a load byte instruction, and remember that the load instruction first look up the cash, and then the cash block size, which is usually bigger than any of these size. And your load byte instructions were, what is the destination? It is the register. We will read that those 8 bits out, byte is the (( )) and put those 8 bits in the register, and what is the typical register size in a machine. 32 bits to 64 bits. So, when we talk about the 32 bit machine, the register size is 32 bits. When we talk about 64 bit machine, 64 bits is the register size. So, a load byte instruction, at the end will go to a register, and that byte will seek in the least significant byte of those 4 byte register. So, now can somebody guess, what this alignment network is. We have a load byte instruction, we have to read the byte from the cash, and put the byte in the least significant byte within the register. I have already told you what it does. Can somebody summaries. What is it's align actually?

Student: Byte is broken up into two parts, because 32 bit is a register size. So (( )) broken into two parts.

Two parts what are these. Do not bring up the word size. Forget about it for now. There is nothing like a word size. I have a load instruction which has to bring 8 bytes from the cash into a register, and if you really want to think about the word size. In this particular case, I do not what you exactly meant by the word size, can you explain then I can tell you. This is going to be this one in this particular mission, that is the maximum. 8 bytes is the word size in this case.

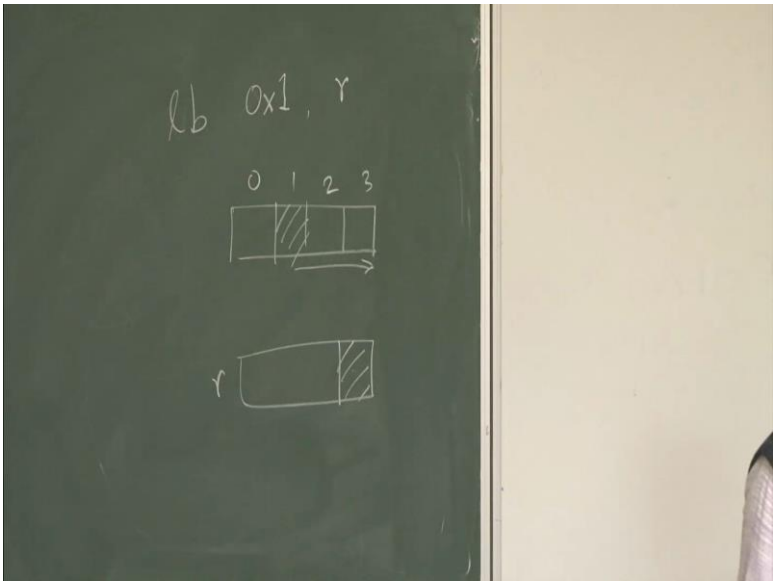
Student: 4 byte size. (( ))

No it is a load byte instruction, no need of byte. So, you will see, if you if you thing about the data paths. I do not know what you are talking about by word size. If you talking about the bus that leads from the cash to the register; that will be 4 bytes, because the register is 32 bits. Any data that cross from the cash to the register will have to be 32 bits. But the maximum that we can read out from the cash is 64 bytes. You have to be, otherwise you cannot execute the double word instructions.

Student: (( )).

No, but in this case I am only reading a byte. It is a load byte instruction. I will give you the answer. You are close. You have almost got it actually. So, these are my data paths which. So, from one go, the cash will provide me 4 bytes. Now from the 4 bytes I have to get the designated byte, and it can be anywhere in this 4 byte, because for a byte load, the condition is only address modulo one, which can be anything. So, suppose that out of these 4 bytes, the required byte 6 in the most significant size. It has to go the least significant size of the register, so need an alignment network; that is exactly holding this. It will rotate the bytes, align into the right position and then copy to the list.

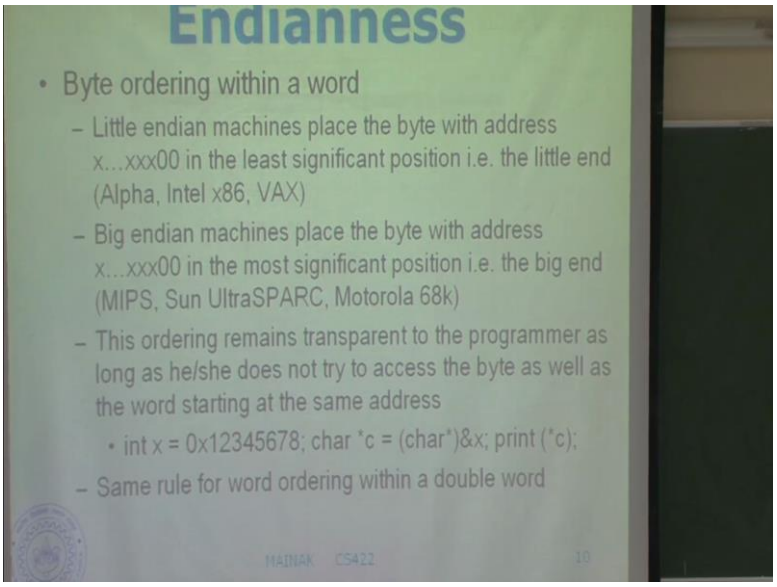
(Refer Slide Time: 22:28)



So, if you want to take concrete example. Suppose I will specify a load byte instruction by L b, or I want to do a load byte from address 0 x 1 bits to register r. So, what will the cash control will do. It will take this address and figure out the 4 bytes solving in this address to be what; 0 1 2 3, I mean this one. I mean this byte and this will go to a register r, and the registers who have it here. So what the alignment of we will do is that, it will rotate it by two slots, and then move it on the paths. Essentially it would shift operation on this side. On some computers, it is possible to access least significant parts of a register; for example, x 86 and leaving the upper portions unaffected. Yes off course yes

it needed for source also. So, this is just an example that I wanted to know. For all memory operations will recovery alignment.

(Refer Slide Time: 24:55)

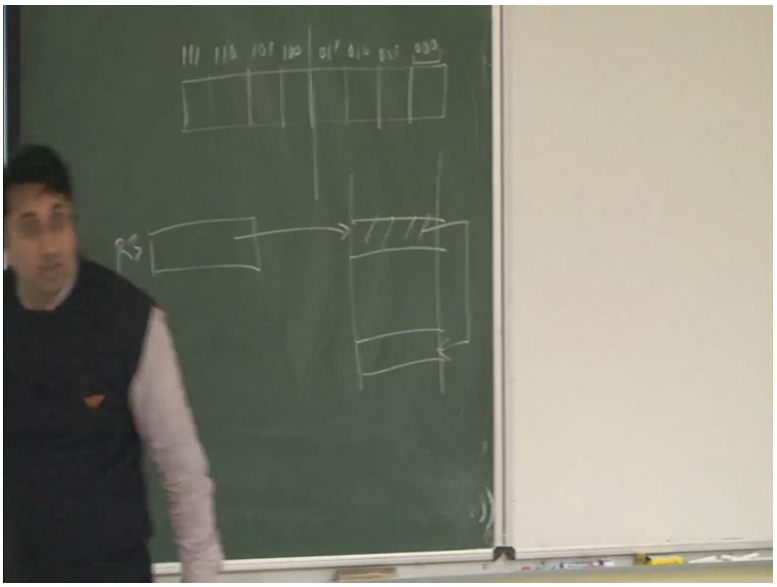


So intimately related to a memory addressing is a concept called endianness. So, this the this (( )) to byte ordering within a word, and one ordering within a double word, both of this. So, here I will talk about byte ordering with in a word. Same rules applying for word ordering within a double ordering. So, little endian machines, place the byte with address at the end 0 0, in the least significant position; that is the little end of the word. So, I can take a word within 4 bytes. So, what is the 4 bytes. So, what will be the address of a word which is aligned. It will have the least significant bits, so will be 0 0 0 1 1 0 and 1 1, for an align word which starts at an address, which is the multiple of 4. So, what is saying is that; the little endian machine, will place the byte the address this in the least significant position. So, this one will actually hold the least significant byte within this word, in a little endian machine. In a big endian machine, is exactly the opposite; the byte sitting here, is actually the most significant byte within the word.

So, here are the example, so alpha not there anymore today, VAX old machine. You have only this one today, and this is the little endian machine. On the big endian sight, MIPS is very much there in the market, sun ultra spark although there not designing any

new processor, but off course you can use them now. Motorola is also again there in the mobile market. So, this ordering remains transparent to the programmer, as long as he or she does not try to access the bytes, as well as the word starting at the same address. So, here is an example; suppose I defined in integral x to be this, is the 32 bit number, and then what I do is I try to distract a byte from it, by saying char star c equal to char star and x, and there print star c, what would you get you know little endian machine. So, a little endian machine what will be the outcome. Yes somebody.

(Refer Slide Time: 26:39)



Student: In little endian it will be 78.

Big endian.

Student: 12.

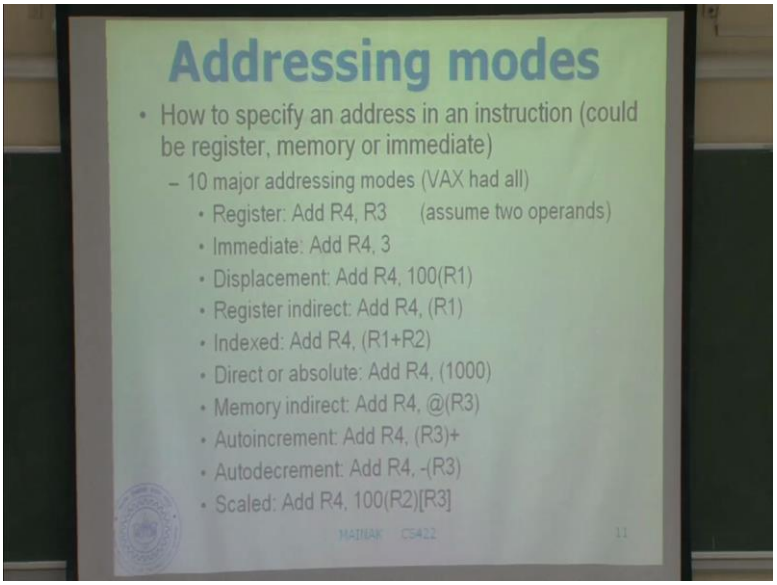
Any other answer. How many of you do not agree with this. How many of you agree with this. What happens to the rest, undecided is it. I hope the undecided folks understand that it cannot something in the middle; is that clear to everybody. It has to be here or there, we talking about the ends, big end of the little end. So, what you a say this correct. So, this is in a little endian machine. So, first of all we have to understand what

he is trying to do. It is trying to extract. So, we will say char star c equal to char star and x, is trying to extract the least significant byte of x, is that clear to everybody, so that is c c. So we will say this, it is trying to extract the least significant byte from x. your question is what is the least significant byte. In little endian machine we have just said that, byte with this address is the least significant bye. So, byte with this address on little endian machine was this one.

On the other hand on a big endian machine, the least significant byte is going to be this one. So, that because it sits on the big end. So, this particular address is on the most significant byte, and since this one is trying to extract the byte, with this particular address; the 0 0, last one actually, what will it get is. In big endian machine you will get this little endian machine will get this; is it clear to everybody, it is very important. And in a double word, we have two words. You can apply the same thing the word ordering there. So, with your double word, the word with 0 0 0. So, in a double word, the align double word, you are going to have two address; that is if you look at the least significant bites in a double word, if we look at the words. So first of all we are looking at the bytes in a word. So, here we are talking about 8 bytes.

This is any one word and this is another word. So, 0 0 0 1 1 0 1 1 and then this is going to be 1 0 0. So, this is also 1 0 1 1 1 0 1 1 1. So, the word with this particular address, will sit on the which side of the double word, can determined by same rules. So, I suggest that you go back and execute this program. These are not a program by the way; you can make it a, convert into a program. Of course, nothing surprisingly will happen into execute on intern machine, you are going to get 78. If you can somehow find spark machine. We have SPARC machines in the department, and execute this one you will this one. No there is no advantage, it just to schools, to design (( )), and whenever you try to make a SPARC machine communicative, then intern machine the word falls apart, because you have to make sure that things are ok.

(Refer Slide Time: 31:02)



So, now over time, the ways of addressing memory have you bought tremendous. So, what I have done, here is the list of things that of come up in several processors, over time, how to address memory. So, these are called addressing modes. So, how do specify an address in an instruction, and it could be specified in a register, it could be specified in memory or it could be an immediate value inside an instruction. So, will see example of each of these three, as we go along the list. So, there 10 major addressing modes, that have been proposed over time, and this machine called VAX at all of them. So, this is called the register addressing mode, you have just 2 operands. So, we all this, actually we have 2 operands. So, in the register addressing mode what you doing is, you are actually not addressing memory.

You are actually just adding to registers, and putting the value in one of these, most probably R4. So, this is called register addressing mode, you are addressing the registers, you are not addressing memory, keep it in mind. Immediate addressing mode, talks about how would you address a particular value, a constant in an instruction; that is called an immediate addressing. Again do not get confused with a memory addressing, these are the memory address, is specifying a constant in the instruction. Are you wanted to add 3 to the value in R4; that is all it says? Displacement addressing mode. This one is actually addressing memory. So, what we are doing is, taking the value in R1, adding 100 to that,

and whatever you get is the memory address. So, often this is called the base register R1, and this is the displacement. Register indirect. So, here essentially same as displacement; displacement is 0. So, it is a special case of displacement address. indexed addressing where you specify your memory address at some of two register values, you add a R1 R2, what you get is basically your memory address, and the name comes from the fact that you can think of R1 as the base address of an array, and R2 is the index into the array.

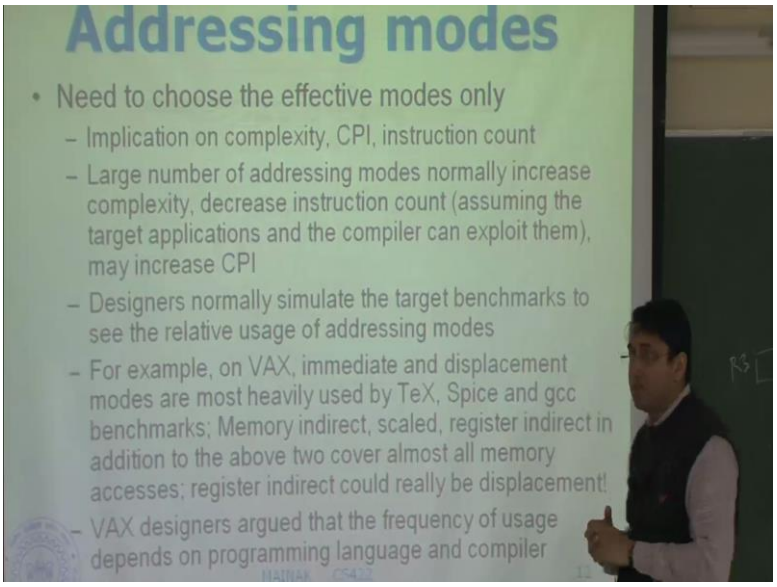
So, what you are doing is, you are actually accessing R1 indexed by R2, is started a R1 add R2 wherever you get is the memory address. Direct or absolute address, you specify the memory address directly. So, notice the difference between this and this. So, way we rotationally we put a parameters is here, specify an address. So, this is a other constant here. This is the constant, but this is an address. Memory indirect, this is probably the most complicated one, what you do is, you take the value in R3, go to the memory location wherever the R3 is pointing to, and whatever the content of that location, is your final address. So, (( )) what it does. This is your R3, it has some value. In using address that point to some memory location. It has some value, you use that as an address it will point to somewhere here, and this is the location that it is referring, this particular one, that is called memory indirect. Auto increment; this one actually two operations, this was first, does the register indirect memory access, and then once this operation is done, it automatically increments the value of R3. What is the application of this instruction, can anybody think of anything.

Exactly right. So, if you want to access an array, you would initialize your R3 to the base of the array, and that is it, you would just put auto increment instruction in that. What will happen what will happen is that automatically R3 will move along. Similarly auto decrement, if you want to start with the end of the array, you can gradually proceed to the head. And finally, you have scaled, this is essentially combination of two things; index, and displacement ok. So, I do not know if you can think of anything else outside this place, how to address memory. Off course we can, could one more add here to get memory indirect, but you know fundamentally everything is pretty much here.

Student: Use of index addressing more.

So, well if you just want to, you can lookups separate instructions for the incrementing R2, if you want to go along and access everything, but as such this is just you know giving you an indexed location into an array.

(Refer Slide Time: 36:47)



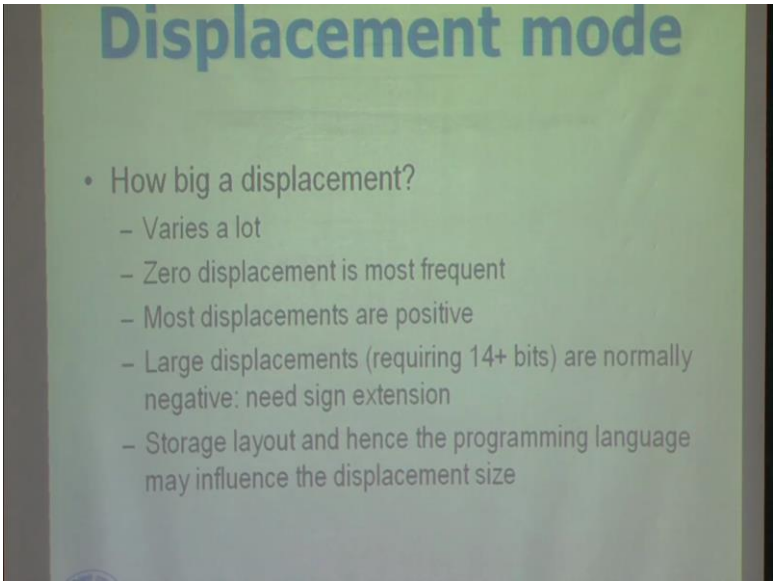
So, although facts had all these addressing modes, it should be obvious that we do not need all these actually. If you analyze programs what will find is that, these addressing modes will appear with different you know frequencies. Some would be very rare, some would be highly frequently. So, you have to choose the effective modes only, because it has implication on complexity, CPI and instruction count. because on one hand, you would like to choose very complicated addressing modes, because that would reduced reinstruction count, because that would encapsulate very complicated memory address in a single instruction like this one. Alright on the other hand implementing these instructions, will be a nightmare, that would probably increase your CPI and also complexity. So, typically what you do is, you take programs, analyze them, and find out which addressing modes are most frequent. You definitely support that, and the rest depends on how much complexity you can afford, or how much that CPI will get affected into implement the remaining address support.

So, large number of addressing modes normally increase complexity and decrease

instruction count, assuming the target applications and the compiler can exploit them; however, they may increase CPI. Designers normally stimulate the target benchmarks, to see the relative usage of the addressing modes. For example, this is the very old example taking for text; on the VAX machine, immediate and displacement modes are most heavily used by TEX spice and GCC benchmarks. Memory indirect scale registers indirect, in addition to the above to cover almost all memory accesses. So, he could actually do away with the remaining wants if that. We have ten things in that place, out of which, in fact, five pretty much covered everything. The register indirect could really be displacement, so the register indirect was. So, as I said, this is the special case of displacement. So, this is the typical analysis that can architect to do, before deciding your addressing modes. So, VAX designers argued that the frequency of usage depends on programming language and compiler, so they did not take any risk.

They said well we will put on time, and probably what they have done in doing so is that, they might have sacrifices CPI for formally occurring programs, to probably speed up some program which is obscure made on may be once in a ten year or something. So, before going along, you have to decide a few things. So, the displacement mode, just to remind you once more what it was. We have somehow encode this particular constant, is a displacement. So at the design time we could decide, how big a displacement can I support, because that will have implication on a instruction size, because this has to go inside the instructions. Instruction has told me that you know you should have a R1, you should have 100, you should have R4, and their proper places in instruction. This usually instruction, when you design the instruction it would have slot for the source register. You will have slot destination register, you will have some slot for the displacement also in this case, and how we get displacement is to decide how we get instructions.

(Refer Slide Time: 40:24)

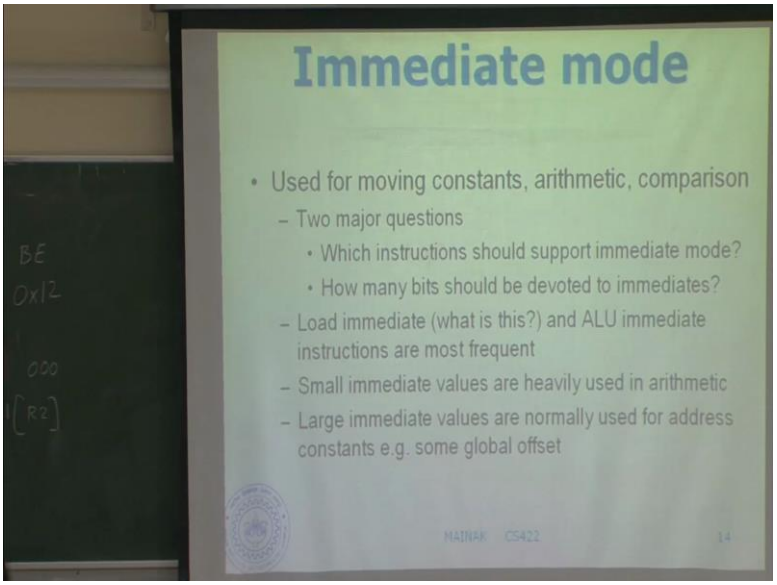


It varies a lot, terms out that 0 displacement is most frequent. And again here you do the same thing; you take for benchmark applications, study them, and find out the statistics. Well tell me the histogram of displacements. I have seen, may be in this entire benchmark suit I have seen, may be 100 of billions of load store instructions. Give me the displacement historiography for this. So, from that we transfer that 0 displacement is the most frequent. And most displacement is usually positive. You would seldom have negative displacements. But you have to keep mind, that these are all conditioned upon what the compiler register, because you remember that compiler is to act generic reason purpose. Large displacements requiring 14 plus bits, are normally negative, so they need sign extension. So, we will talk about this soon, what that actually means.

Those who remember little bit about binary addition, might remember what sign extension means and why that is needed, but any way we will talk about that in next lecture. Storage layout and hence the programming language may influence the displacement, you have to keep in mind, but any way the takeover point here is that; 0 displacements are frequent, most displacements are positive, and positive displacements are usually small. So, what is means is that, you can you are fine if you have you know small number of bits important displacement So, will see what MIPS did exactly in the next lecture. How many bits important displacement portal. Any question of this.

Similarly, the immediate mode, same question arises how make a constant can I put in my instruction. So, so here of course, there is a very clear tradeoff. If I find that my program has, mostly large constant, and I cannot put them in instruction; that means, I have to store them in memory, and it will require extra instruction bring them to the processor actually. And of course, if I can put all of them inside my instruction, then when instruction is fixed, the value will come along with that actually.

(Refer Slide Time: 42:24)



So it is used for moving constants, also used in arithmetic operations comparisons. So, two major questions; which instruction should support the immediate mode, number one. So, clearly arithmetic instruction, should support the immediate mode, because I would definitely require manipulating constants. I would require adding a constant. I would require multiply a constant. I might require dividing a constant. I might require logically a operating on a constant like; logical ending, oaring, exouring or all those things. So, of course, I should probably require this in memory instructions, because if I want to specify an absolute address, I should be use immediate. How many bits should be devoted to the immediate? Same question has displaced, how constant can I put in a memory instruction. So, load immediate, and ALU immediate instructions are most frequent. So, these instructions, load immediate are not really loads. These are just moving a constant to a register. They do not access memory.

So, again if you look at the benchmark statistics, you will find small values are heavily used in arithmetic. also you can you can think about, the programs that you have written in your life, then ask how many times have you used the very large constant, you look fine to many cases. In most cases you will be using small constants; in most cases actually 0 is the heavily used constant. Large immediate values are normally used for address constant; like some global offset and all those things. So, usually these are anyway kept in registers. So, we will talk about which registers are used for keeping these global address and all.