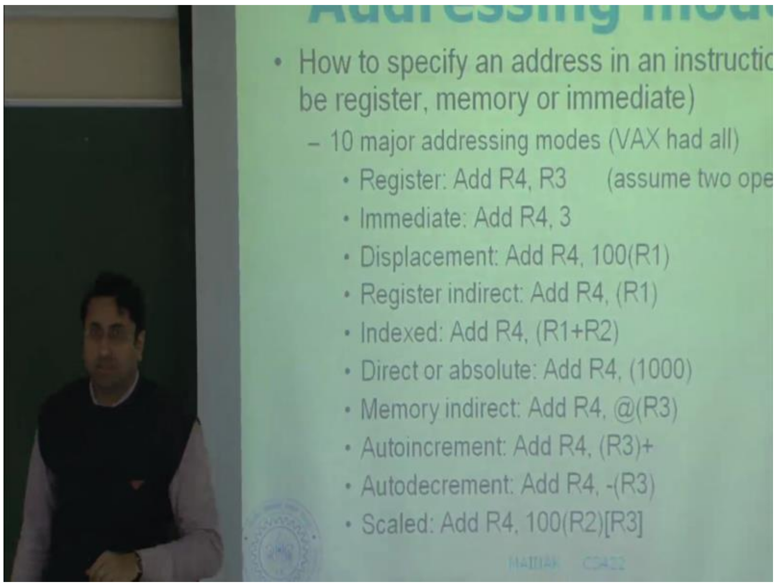
Computer Architecture

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

Lecture - 4
Instruction Set Architecture

Last time we talked about these addressing modes. So, any question on these. So, I want to mention a couple of things, before moving on. We also talked about how big the displacement should be, how big immediate value should be and so on and so forth.

(Refer Slide Time: 00:49)



- How to specify an address in an instructio be register, memory or immediate)
  - 10 major addressing modes (VAX had all)
    - Register: Add R4, R3 (assume two ope
    - Immediate: Add R4, 3
    - Displacement: Add R4, 100(R1)
    - Register indirect: Add R4, (R1)
    - Indexed: Add R4, (R1+R2)
    - Direct or absolute: Add R4, (1000)
    - Memory indirect: Add R4, @(R3)
    - Autoincrement: Add R4, (R3)+
    - Autodecrement: Add R4, -(R3)
    - Scaled: Add R4, 100(R2)[R3]

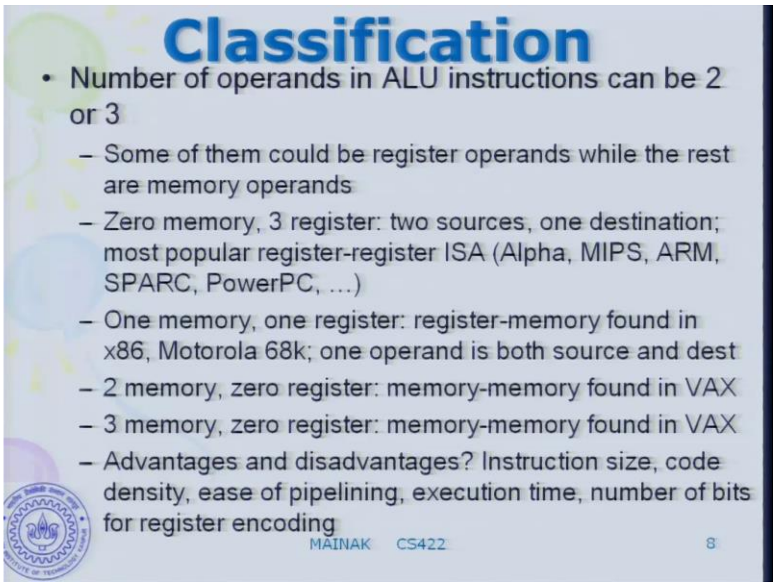
So, if you look at, so there are couples of things. The first thing is that, so these are essentially, as you said, these are registered operations. These are nothing to do with the memory. Whereas the rest would go an access memory in some way or the other way. And the special case was, these one, who is actually accesses memory acquires. Now of course, as you will go along, will find that, these two are usually the most frequent, addressing modes; that you observing programs.

So, natural question that arises is under, what circumstances the compiler generate, such instructions. Because, see compiler has the ability to always generate these instruction register indirect. You can always put address in a register and can generate these instructions to access that particular memory. So, the question is, why should the compiler need, these instruction. So, that is the very valid question. So, that is one question, you will try to answer.

The second question is, which I want to address first is, if you look at these addressing modes, the way you address the memory. Particularly, if you take these two, you find that we have a very compact to have representing the address in the instruction. Essentially, what you are doing is, they are putting the address in the register and we do not have to put the address in the instruction any more.

As long as I put the register index 1 and register index 4, my instruction is complete and of course, along with Op-code of Add.

(Refer Slide Time: 02:27)



Classification

- Number of operands in ALU instructions can be 2 or 3
  - Some of them could be register operands while the rest are memory operands
  - Zero memory, 3 register: two sources, one destination; most popular register-register ISA (Alpha, MIPS, ARM, SPARC, PowerPC, ...)
  - One memory, one register: register-memory found in x86, Motorola 68k; one operand is both source and dest
  - 2 memory, zero register: memory-memory found in VAX
  - 3 memory, zero register: memory-memory found in VAX
  - Advantages and disadvantages? Instruction size, code density, ease of pipelining, execution time, number of bits for register encoding

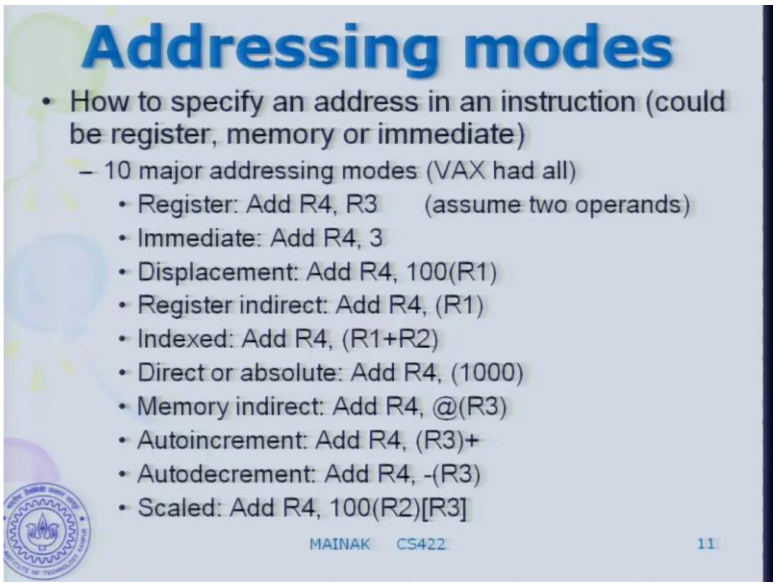
MAINAK CS422 8

So, previously, if I back at the little bit, we were talking about these classification here. That we had, these classification based on, number of memory operands, some register operands and so on and so forth. And you said that, if we have memory operands, your instruction size might be increase, because you have to put a big address in instruction. But of course, that is not exactly correct or you can easily see that.

Because, you have other compact ways of representing address. You can put the address register and congest, put the registering index and instruction. That is enough. So, but of course, if you are using absolute address; your instruction, then of course, you will have to put the full address. But, actually, nobody does it. That is very in frequent way of accessing memory.

And the obvious reason now, should be clear that, it would increase your instruction size. You would always put the address in register. And then, generate the corresponding instruction, like one of these.

(Refer Slide Time: 03:18)



Addressing modes

- How to specify an address in an instruction (could be register, memory or immediate)
  - 10 major addressing modes (VAX had all)
    - Register: Add R4, R3 (assume two operands)
    - Immediate: Add R4, 3
    - Displacement: Add R4, 100(R1)
    - Register indirect: Add R4, (R1)
    - Indexed: Add R4, (R1+R2)
    - Direct or absolute: Add R4, (1000)
    - Memory indirect: Add R4, @(R3)
    - Autoincrement: Add R4, (R3)+
    - Autodecrement: Add R4, -(R3)
    - Scaled: Add R4, 100(R2)[R3]

MAINAK CS422 11

So, only here, we talked about putting the absolute address, but you sell of ((Refer Time: 03:24)). So, the second question was that, why should compiler generate these instruction, displacement based address? Because a compiler can always generate these one. You can put the address in a register and generate these instructions. So, this one you will find, very often, when accessing the stack.

So, usually, our stack pointer, which is registered. And if you want to access something in the stack, you generate negative displacement here on the stack point. Negative or positive depending on which way your memory goes. Similarly, if you have some pieces of global data, usually global data are stored in a specific region of memory. And the base of that memories pointed by another register called the global pointer.

So, if you want to access some data in the global memory, you would essentially generated displacement with respect to the global point. So, these are the cases, where you will find that displacement is instruction getting generated by the compiler. And of course, as I said this is the special case of that with the displacement of 0. For example, if you want to access top of the stack, it will generate this instruction with R 1 being the stack point.

Student: ((Refer Time: 04:45))

No, here for example, we are allowing, you to do early operations, directly on a memory operand. So, this would be like a memory registered ((Refer Time: 05:00)) like that.

Student: ((Refer Time: 05:02))

No, it is a memory register actually. One operand, these a memory address is only that, we have a compact representation of the memory address. So, you are using register to store the memory addresses. There is no other difference, whereas if you just remind others, if you forgotten, if you only have loads to instruction, allowing memory operands. Then, that would be essentially registered is indirect, where an early operation will operate only on registers.

Student: ((Refer Time: 05:39))

Sorry, say again, what is the exact question, so...

Student: ((Refer Time: 06:01))

Exactly, so you are saying, why not used this one. That is a very good point actually. So, I was exactly about to mention that. So, why would I need this one, if, I have this one, I should be able to exactly synthesis this. The point is that, your displacement would be limited to the certain size in the instruction. Because, see displacement will actually appears at constant instruction. So, there will be limited number of bits, given to you for encoding displacements.

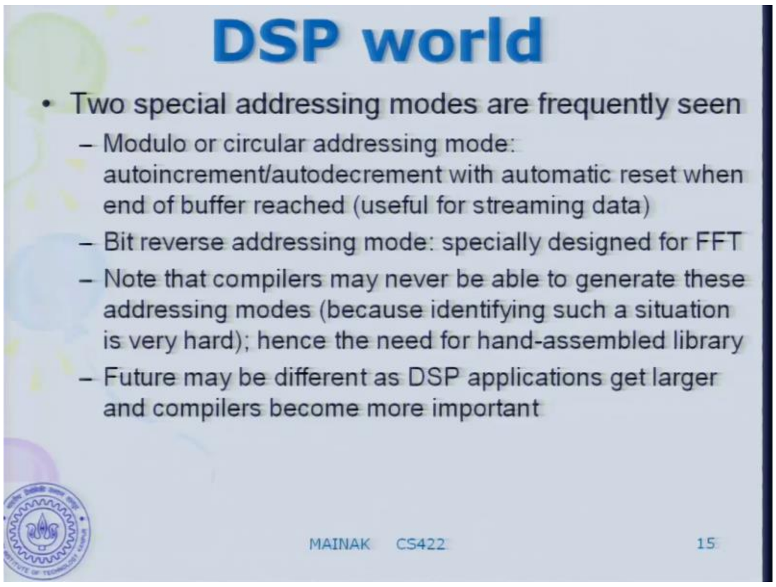
Suppose, you want a very large displacement, how you do that, but, then you have to user to this one. You have to use the index ((Refer Time: 06:39)). We will put the discussion in R 1, put the base in R 2 and generate these instructions. That is an answer to the question, anything else on addressing modes, any other questions? So, in these last three cases, these auto increments, auto decrement, scale. So first of all, let me remind you, what this is?

So, essentially here, we are taking the value at location pointed to by R 3. And adding it to R 4 and putting the result in R 4 and also incrementing R 3. That is what, you say. Similarly, in auto decrement you, decrement R 3 and the scaled case, we essentially adding R 2, R 3 in 100 and generating a memory location. In certain processors, you will find that, there is one more extra parameter. And that is usually programmed into the processor, before you invoke auto increment instruction.

And that is, how by how much you should increment. Here, you are assuring by defalcation by 1, need not be. So, usually, have an instruction before auto increment, which would actually programmed that session register, which would be implicit here. So, you would increment by that amount and the reason is that, you might have an application, where your array elements are not exactly 1 byte.

So, here, we say that, you could initialize R 3 to the array base and you just want to add all the elements of R 3, cumulative way. So, in that case, you just do this instruction a loop and automatically, the R 3 will get incremented. So, now, if you have array elements and not a byte, you will want to increment by a larger amount. And that is exactly but this becomes handling, that how by how much, I should increment that particular flexibility.

(Refer Slide Time: 08:25)



DSP world

- Two special addressing modes are frequently seen
  - Modulo or circular addressing mode: autoincrement/autodecrement with automatic reset when end of buffer reached (useful for streaming data)
  - Bit reverse addressing mode: specially designed for FFT
  - Note that compilers may never be able to generate these addressing modes (because identifying such a situation is very hard); hence the need for hand-assembled library
  - Future may be different as DSP applications get larger and compilers become more important

MAINAK CS422 15

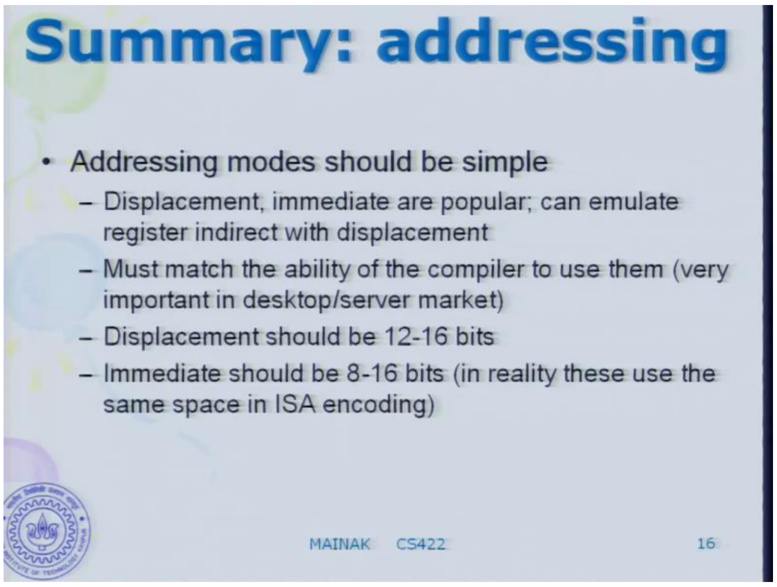
So, digital single processors, it see to special addressing modes very frequently. This is, usually called modulo or circular addressing mode. Here, auto increment, auto decrement with automatic reset, when end of buffer reached. So, this very useful for streaming data, you are streaming through a data. And when you reach the end, this automatic reset to that. So, that, you have to prepare to stream again, next time.

There is another addressing mode found in digital single processor called the bit reverse addressing mode, specially designed for Fast Fourier Transform. So, I will not going to details of this. You know about FFT, you would be able to understand, why that is needed? If you do not know about FFT, I would suggest that, you go home and read about it.

You will understand why, this is actually needed bit reverse, addressing mode? Now, the important point is that, that compilers may never be able to generate these addressing modes. Because, identifying such a situation is very hard. As I told you last time, first of all, the compiler will have part time figuring out that the FFT is going to on. You given a large piece of quote and it is very, very difficult for the compiler to figure out, the curve is an effective actually.

So, I should be generating bit reverse addressing modes. That is almost impossible. So, essentially, what happens is that, you would have had assembled library, due to actually, hard quote these addressing modes in an instructions. You will real on the compiler. So, as we go along DSP applications get larger and compilers become more and more important. Because, gradually, we are of course, improving your DSP compilers at one.

(Refer Slide Time: 10:01)



Summary: addressing

- Addressing modes should be simple
  - Displacement, immediate are popular; can emulate register indirect with displacement
  - Must match the ability of the compiler to use them (very important in desktop/server market)
  - Displacement should be 12-16 bits
  - Immediate should be 8-16 bits (in reality these use the same space in ISA encoding)

MAINAK CS422 16

So, in summary, addressing modes should be simple. Displacement and immediate are popular. So, again, I would actually remind you that, whenever you talk about the immediate addressing mode, it does not refer to accessing memory, is referring to operating on a constant. So, it is not actually accessing memory, remember that. And you can always emulate register indirect with displacement. Because, displacement is just a special case of displacement is that 0.

Your addressing modes must match the ability of the compiler to use them, very important in desktop server market. Because, here, you know hand quoted assemble libraries are just out of question, even the wide variety of programs you are done. Displacement should be 12 to 16 bits. So, this has pretty much become the factor today, based on your statistics collected from applications. We discussed this last time also; that the displacements are usually small.

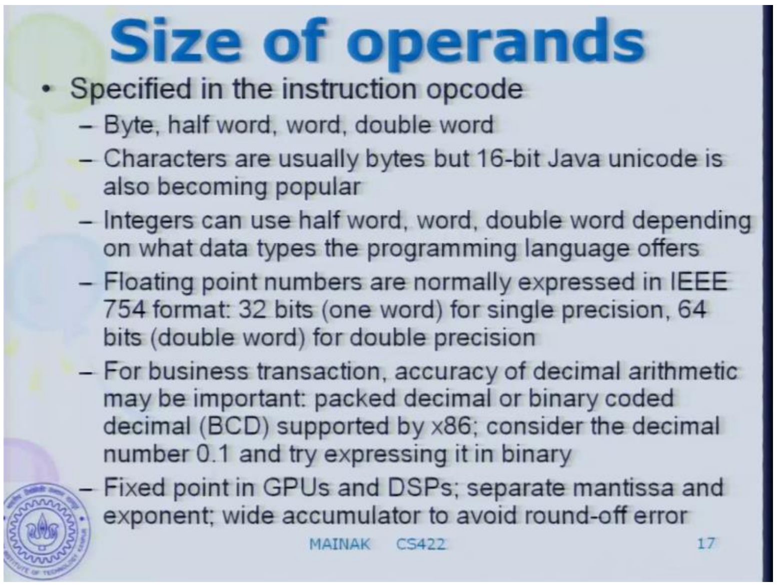
So, this transfers to be enough in most cases. Immediate should be 8 to 16 bits. That is again based on your statistics collected from applications. In reality, this used the same space in the ISA encoding. So, the immediate and the displacement, actually use the same bits in the instruction encodings. So, will look at concrete example, where we talk about the MIPS.

Student: ((Refer Time: 11:24))

At least, one register operand you mean, the reason is that, if you, when your designing the ALU, for example. Usually, the ALU generate a result in register. So, that is the reason, why you want to have at least one register in the instruction. So, you could hide that as well. See, I mean it is all about, what you expose to the outside world. So, under needs, what is really happening is that, whatever memory operands you have, ultimately the memory is accessed and brought into a temporary register inside the CPU.

Otherwise, you would not be able to input the adder, out of question. Because, adder takes inputs from two register and put the result in another register. You could hide all these temporary registers, from the programmer. So, programmer would actually see, accessing to memory locations, getting results back in another memory location. That is possible.

(Refer Slide Time: 12:21)



Size of operands

- Specified in the instruction opcode
  - Byte, half word, word, double word
  - Characters are usually bytes but 16-bit Java unicode is also becoming popular
  - Integers can use half word, word, double word depending on what data types the programming language offers
  - Floating point numbers are normally expressed in IEEE 754 format: 32 bits (one word) for single precision, 64 bits (double word) for double precision
  - For business transaction, accuracy of decimal arithmetic may be important: packed decimal or binary coded decimal (BCD) supported by x86; consider the decimal number 0.1 and try expressing it in binary
  - Fixed point in GPUs and DSPs; separate mantissa and exponent; wide accumulator to avoid round-off error

MAINAK CS422 17

So, next question is, how big should my operands be, in my instructions. So, this is usually specifies in the instruction opcode. So, here, again, we are talking about the memory instructions. Because, when your register instruction of course, your register size fixes operand size, automatic. So, usually, this is a specified in an instruction opcode, like we talked about last time, bytes, half words, words, double words.

So, in this class, we will follow that convention that, double word is 8 bytes, 64 bits. Word is 32 bits and half word is 16 bits and byte is of course, 8 bits. Characters are usually bytes, but 16 bit java unique code is also popular. Integers, so second here, what am trying to do is, when you have a high level language program; that will have data types, characters, integers, real's, floats, depending on the what language you program.

Now, the compiler, you will have to compile that data type and you will have to map it to some operand type in the machine actually, in the instructions. So, there has to be a mapping, one to one mapping between you are high level language data type and your instruction data type. So, for example, when you operating on a character, your instruction should be actually a byte instruction is to generate some instruction that manipulate bytes.

So, integers can get mapped to half word, word, double word, depending on what data types; that programming language actually offers. Because, it barriers a lot. For example, if you are doing programming in C, you would have shorting, you will have integers. You will have long integers. There are Ben barriers or flavors. So, it all depends on, how it exactly interpreted.

So, essentially compiler would know the meaning of these things. Like for, you say, shorting compiler would know, what it means actually would be automatically generate the corresponding data size to your instructions. Floating point numbers are usually expressed in IEEE 754 format. So, this is the standard actually. So, 32 bits or 1 word for single precision, 64 bits or double word for double precision.

And there is another 80 bit extended precision. So, these are again standards. So, whenever the compiler sees that, we have a double variable. It will probably try to generate double precision 64 bits. If the machine supports, otherwise, that value will be broken down to single precision 32 bit numbers and the corresponding instruction will interpret these two together actually.

So, will again look at, when you look at MIPS, 32 bit MIPS. You will see, how it actually manages 64 bits data types. For business transaction, accuracy of decimal arithmetic may be important. In fact, it is very important. So, usually there, you would use packed decimal or binary coded decimal. So, these are supported by x 86. So, you can actually generate BCD operand types in your x 86 instructions.

So, what is primary coded decimal? That is anybody knows, what is BCD? It is just a number system, just like your binary extra decimal. What is BCD?

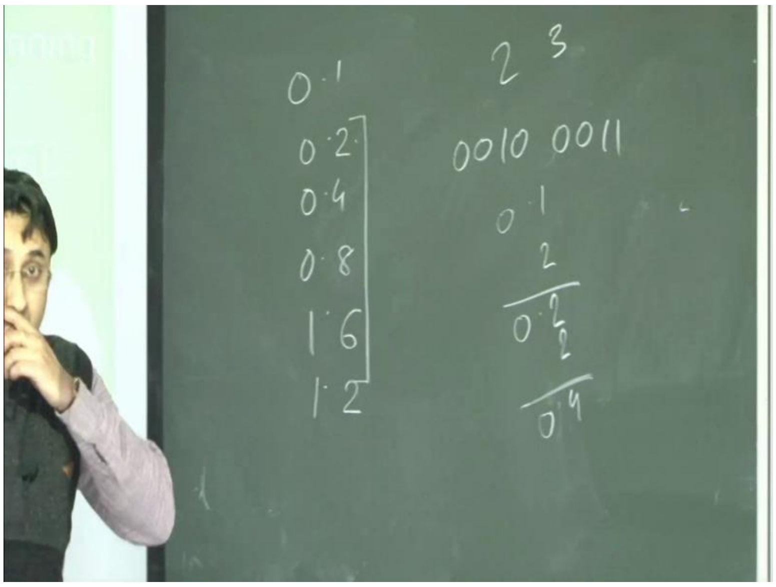
Student: ((Refer Time: 15:50))

Decimal, exactly for each digit, so for example, if I write number 23, in BCD encoding would be 0010, 0011, you need 4 bits, 0 to 9. That is the BCD encoding of 23. So, why would you do that, here, it is an example. Suppose, you want to express the decimal number point 1 in binary, how do you do that, point want to binary. What is the procedure?

Student: ((Refer Time: 16:31))

Multiplied by 2...

(Refer Slide Time: 16:35)



And so what is this going to generate 0.1, then 0.2, then 0.4, then 0.8, 1.6, then

Student: ((Refer Time: 16:49))

1.2.

Student: ((Refer Time: 16:53))

So, you are back here, it recurs now, this part. It is going to be a recurring binary representation. It is not going to terminate. So, clearly, but you have to terminate it somewhere, because your machine has only finite precision. It can only store fined number of bits. The problem is that, suppose you make a transaction of 20 rupees 10 paisa, you converted into binary, inside your machine. You keep losing money, slowly, over time, it will build up by a huge amount, I would say.

So, that is the problem, why the business transaction machines, like you are the ones; that is used in banks and ATM's would probably never use binary representation. Routinely, use only BCD or would go to this packed decimal, is it clear to everybody. The problem of facing binary, your transactions will be inactivate and you start losing money or may be gaining money, depending on which way, you round up.

And of course, I showed you, how to presented interior. So, of course, you can represent a fraction also in BCD. You will have to put the fixed point somewhere and we have encoding all that. Fixed point GPU's and digital single of GPU's, stand for Graphic Processing Unit. And digital single processors are also very common. So, here, essentially what you do is, you have separate mantissa and exponent your representation.

So, what are my operands, the final question, simplest operations are used most frequently. For example, 10 x 66 instructions were found to be sufficient to cover 96 percent of the entire, take integer 92 so on. It says, it is an old, but this statistics has not change. Commonly, supported operations are arithmetic and logic, load store operations, control transfer, like branch operations, system calls or talking to the operating system or talk to your devices, like the keyboards, displays and all these things.

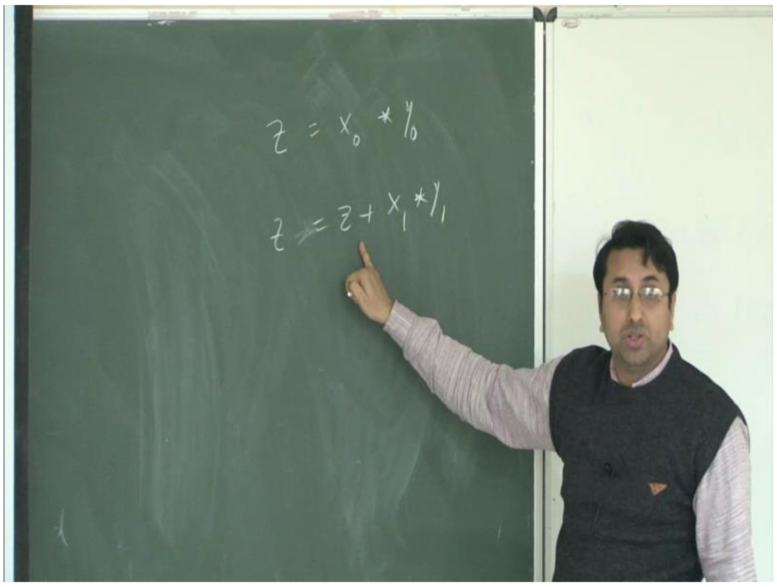
And of course, the floating point operations, floating point arithmetic operations. Optionally, supported decimal arithmetic, supported by x 86, string operations, again supported by x 86. By graphics here, I actually, so it is there. So, graphic operations are often uploaded to a co processor. For example, your graphics processor should support that the special graphics instruction.

Media and signal processors normally operate on narrow width data. So, here, also by media again multimedia extensions and all this your instruction also coming to same category. So, possible to execute multiple such operations in parallel. So, these operation called as single instruction multiple data operations, SIMD. For example, you have 4 weights SIMD in 32 bit.

Intel processor would actually have 128 bit wide registers. So, that, it can offer 32 port 32 bits operands to were parallel. Beta single processors, normally support saturating arithmetic. Because, so by saturating arithmetic, I mean that, suppose, you have 4 bit registers and the current value is a same 14, you act 2 2 it. So, it will saturate 50 not actually over flow a make a makes.

The reason is that, on over flow taking an exception is a order of question, especially in real time application. Real time application you just cannot think about like that, because you sole the deadline; that would a big problem. So, saturate the maximum value. The DSP also use something called multiply accumulator operation. So, these are essentially a huge multiply act and very important, for example, calculating inner product of two vectors. So, essentially what are you doing is that, you adjust doing this x i, y i. So, what it do it is, it would multiply an act together, because you can fuse the whole thing into one instruction.

(Refer Slide Time: 21:41)

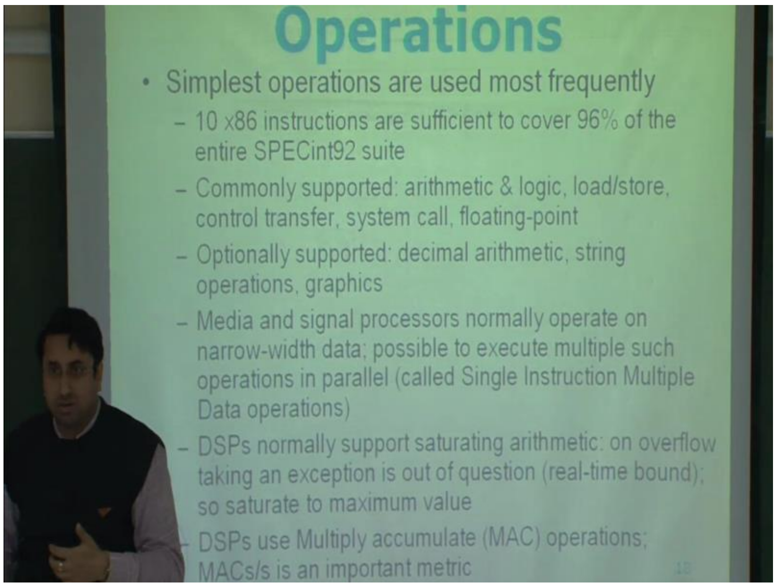


So, let us show you how it actually operates, so initially you do x 0 times y 0 putted into some registers set, there after that every instruction will be to in this. So, this is one instruction multiply, accumulate together it owned be two instruction. So, that is the mater alright. So, essentially you can complete doing an inner product of two inner versions of vectors using a subsidiary extractors, you own required to enough versions to do. So, MAC's per second is probably the most important metric used by physical signal cost.

Student: ((Refer Time: 22:28))

It would take less time competitive doing a multiply an act, may not be same were bigger than each of these, but taking together to be solved.

(Refer Slide Time: 22:59)

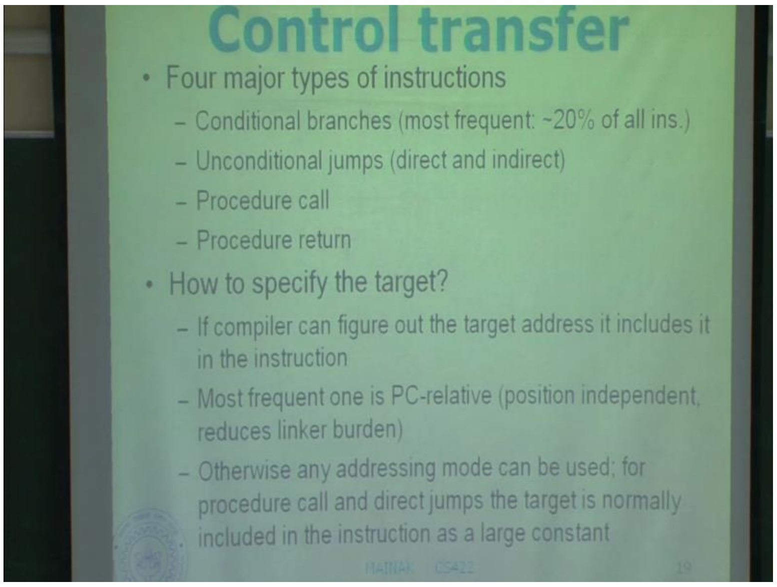


Operations

- Simplest operations are used most frequently
  - 10 x86 instructions are sufficient to cover 96% of the entire SPECint92 suite
  - Commonly supported: arithmetic & logic, load/store, control transfer, system call, floating-point
  - Optionally supported: decimal arithmetic, string operations, graphics
  - Media and signal processors normally operate on narrow-width data; possible to execute multiple such operations in parallel (called Single Instruction Multiple Data operations)
  - DSPs normally support saturating arithmetic: on overflow taking an exception is out of question (real-time bound); so saturate to maximum value
  - DSPs use Multiply accumulate (MAC) operations; MACs/s is an important metric

So, let us take look at, so who will spent some time talking about arithmetic logic, although unsure there is master talking about, you know may about this the ACT, SUBTRACT, MULTIPLY, DIVIDE, AND, OR, NOR, XOR, all these operations are arithmetic logic, load store not must talk about, load from memory to a registers, store from a register to a memory, the next talk little bit about control transfer.

(Refer Slide Time: 23:19)



Control transfer

- Four major types of instructions
  - Conditional branches (most frequent: ~20% of all ins.)
  - Unconditional jumps (direct and indirect)
  - Procedure call
  - Procedure return
- How to specify the target?
  - If compiler can figure out the target address it includes it in the instruction
  - Most frequent one is PC-relative (position independent, reduces linker burden)
  - Otherwise any addressing mode can be used; for procedure call and direct jumps the target is normally included in the instruction as a large constant

MAINAK CS422 19

So, there are four major types of instructions, a conditional branches most frequent about 20 percent of all instructions. So, which basically needs that more or less every 5'th instruction is a branch. So, there fairly frequent, unconditional jumps could be direct or indirect. So, direct unconditional jumps will actually encode your jump target attracts in the instruction.

So, that from the instruction you know, where to go for as unconditional indirect jumps would actually take you target from resist, if you read a register know where to go. Procedure calls is also unconditional jumps because, there is a condition on that you call a procedure you go that only difference with this is that, without do some extra work to make sure that to come back to the right point when you written.

So, that is procedure return, it is also another unconditional jump. But, it does again some extra work you make sure that you restarting execution from the point, where you call the procedure. So, naturally the question arises in all these cases is how to specify the target. Because, is that is the most important think, when you talk about a branches structure.

Otherwise, there is nothing very interesting, because, you are opcode will tell you that, this is a branch instruction. Of course, conditional branches would have another extra thing that is the condition. So, let us first see how to specify the target, if the compiler by the way procedure calls may be of two types, maybe I will talked about this one talked about the targets, so let us hold on to that for some time, see that compiler can figure out the target address it includes it in the instruction as simple as.

Most frequent one is the PC relative targets, so these are essentially position independent because, it reduces linker burden. Because, what it says that well if your now here, I would like to jump 16 more instruction down that is the PC relative target, so finally, it does not matter where this code ultimately goes in what address because, this op sets still remains un change.

So, then you do not have to think the compiler does not have to worry about finally, where should this code set, if it has to generate this absolute address, which is why you will often find that compiler would generate this kind of addresses PC relative. Because, is the easiest want to do, because and the decouples, the compiler from the linker because, the linker will find to decide where the code sits. Otherwise, any addressing mode can be used for example, so for PC relative addressing we would see that will actually use the immediate addressing mode.

Because, you put the offset as the immediate value in the instruction, so internally what will happen is that, the machine will take the immediate value add it to the PC to generate the targets. And of course, the immediate can be positive or negative, you can go backward you can go forward, either way you can go for procedure call and direct jumps the target is normally included in the instruction as a large constant, how the there is an exception.

A procedure calls can be indirect, meaning that the compiler actually may not know the target, when compiling this program, can you think of, it is ALU?

Student: Pointers.

Pointers.

Student: ((Refer Time: 26:56))

Sorry, say it...

Student: ((Refer Time: 26:59))

Function points is exactly anything else, any other situations?

Student: ((Refer Time: 27:07))

Multiple.

Student: ((Refer Time: 27:11))

Dynamical linked it, so let us talked about statistics binaries, we are statistics binaries only any other situations, another function pointers, indirect procedure calls. So, another example is perusal methods would actually internally your perusal methods get compile into function pointers. The another example is, switch case statements, you know think about how a switch case statement would get compiled, how will it get compiled either switch we have munch of cases.

Student: Would it be conditional jumps.

What is the clean way of compiling switch case? So, there is a restriction on your case argument, it has to be an integer you know why right does everybody know that, that case arguments are to be integers, it cannot be a predict point numbers for example, why is that.

Student: ((Refer Time: 28:36)) Loses the value.

Loses the value.

Student: ((Refer Time: 28:40))

That is not, that is an implementation detail what would be a clean way to compile switch case, would it be a series of conditional branches or I will tell you what it normally does probably will I do not know those who are taking compiler course this semester will learnt, so essentially the compiler means a table. So, each case target will be the table entry. So, case 0 target will be in the first row of the table, case 1 will have to second row of the table and so on.

And where into the switch at run time, it will resolve over the switch value is an. It will actually call the procedure at that particular row of the table. So, essentially it is an indirect when it is not exactly a procedure call, it is actually indirect unconditional jump you can think of it that way, it goes there at run time it resolves the values and calls the corresponding, essentially changes the PC to point to the content of that row of the table anyway.

So, per indirect procedure calls of course, you cannot put the target in the instruction because, you do not know, what you would usually do is you will have an instruction first to note the target into a register. And that register will be used in your procedure calls, so at run time the value will be picked from the register and you would take the ((Refer Time: 30:17)) So, here.

Student: ((Refer Time: 30:22))

Sorry, what?

Student: ((Refer Time: 30:24))

Right, yes, exactly.

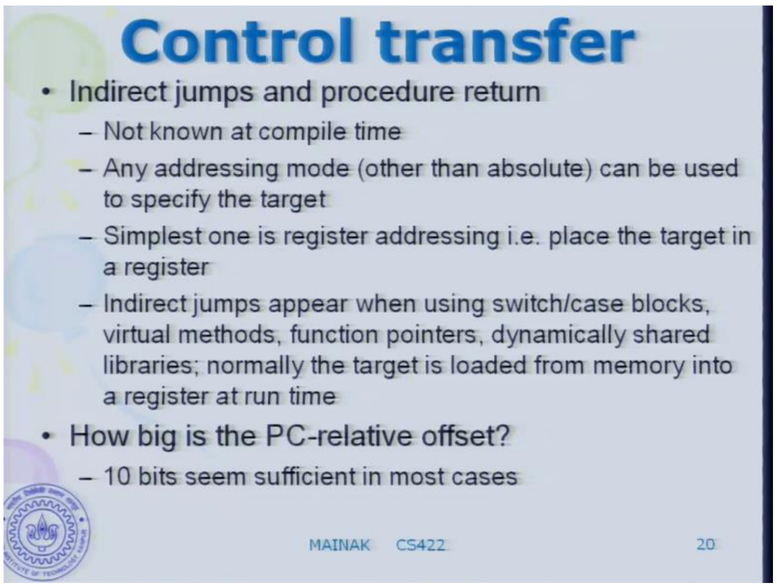
Student: ((Refer Time: 30:28))

Sequentially in the sense.

Student: ((Refer Time: 30:35))

I see, so it will actually having a mapping for that also yes of course, the actual value mapping to the that number of row.

(Refer Slide Time: 31:07)



Control transfer

- Indirect jumps and procedure return
  - Not known at compile time
  - Any addressing mode (other than absolute) can be used to specify the target
  - Simplest one is register addressing i.e. place the target in a register
  - Indirect jumps appear when using switch/case blocks, virtual methods, function pointers, dynamically shared libraries; normally the target is loaded from memory into a register at run time
- How big is the PC-relative offset?
  - 10 bits seem sufficient in most cases

MAINAK CS422 20

So, indirect jumps and procedure return, so again procedure return that is also not known at compile time what that, where I should written I have a procedure, which I am trying to compile and at the end I have written statement do I know where to return why not.

Student: ((Refer Time: 13:21))

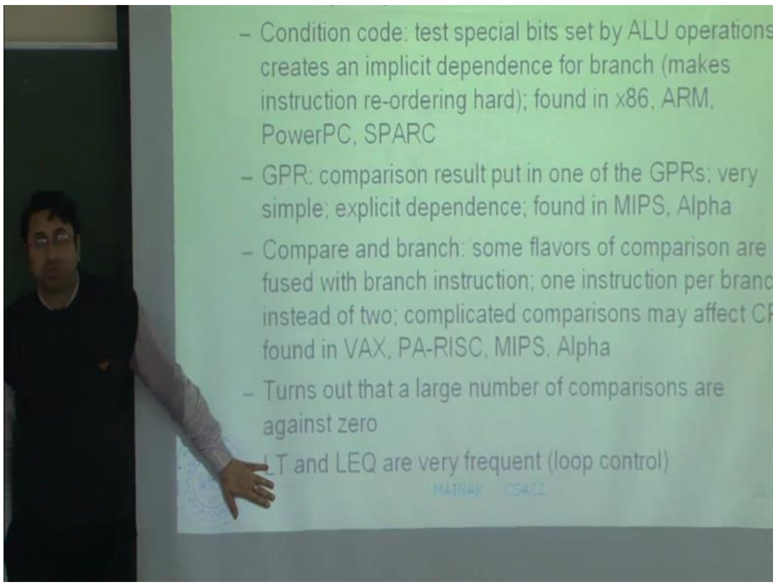
Exactly, so there will be multiple places from where I can call this procedure. So, depending on your execution, it might have to return several other places. So, not known at compile time. So, you can use any addressing mode, other than absolute, because, absolute is used only for any no at compile time. So, the simplest one is register addressing that is place the target in a register. That is what is used replace a target and register and used that register in your instruction for jumping.

Indirect jumps appear, when using a switch case blocks, virtual methods, function pointers, dynamically shared libraries, somebody pointed out. Normally the target is loaded from memory into a register at run time, how big is the PC relative offset 10 bits seemed sufficient in most cases. So, again the question is I am here now I have a PC relative branch instruction, what is the range that, how far, can I go positive one or negative trans out 1024 is in most cases in a. So, 1024 this direction or 10 KB offset.

So, the next component of the conditional branches is the condition how to specify branch conditions. So, there are several ways of doing that, one possibility is to be condition code that is a session bits said by the elude operations. So, this is exactly, what is used by x 86. So, you would first do the elude operation like, suppose you want to check for greater than equal to, so you would first do the comparison and depending on the comparison outcome elude would set some flag somewhere.

Then, you will have a branch instruction which would depending on the steers of that flag will either branch to some location or just continue.

(Refer Slide Time: 33:20)



- Condition code: test special bits set by ALU operations creates an implicit dependence for branch (makes instruction re-ordering hard); found in x86, ARM, PowerPC, SPARC
- GPR: comparison result put in one of the GPRs; very simple; explicit dependence; found in MIPS, Alpha
- Compare and branch: some flavors of comparison are fused with branch instruction; one instruction per branc instead of two; complicated comparisons may affect CP found in VAX, PA-RISC, MIPS, Alpha
- Turns out that a large number of comparisons are against zero
- LT and LEQ are very frequent (loop control)

So, it creates an implicit dependence for the branch and that makes your instruction re ordering part. So, simple reason is that, your branch instruction now would depend on the ALU instruction. Unless, the ALU instruction, you know completes the branch cannot go and more problematic thing is that, this particular flag. That this ALU instruction sets usually does not appear that part of the instruction. It is an implicit target, that you should actually implicit the change that flag. That is the most problematic part.

Because, if a compiler is looking at the instructions and trying to reorder them. If you it can easily make an mistake, thinking that, this instruction, this two instruction look actually impend. But, actually implicitly the ALU instruction would change a flag with a branch instruction depends on. So, all these things will have to be kept in mind, when writing the compiler. So, found in x 86 ARM power PC spark, the other option is to use the general part of registers, instead of you know special flags.

The comparison a result is put into the one of the purpose registers and the branch will actually check that register before branching. So, here it is explicit may instruction, the ALU operation would actually have this particular register as it is destination and the branch will have this register as a source. So, it is a very clear dependence between these instructions. So, there is an explicit dependence found in VAX and Alpha.

And the third option is, compare and branch that is you few these two things together in a single instruction, some flavors of comparison are fused in branch instruction. So, instruction for branch, instead of two, so, you save an instruction you do not have to spread into the two instruction. You can do it on one. But, of course, what flavors of comparisons, you can fuse with the branch will depend on the complexity of the comparison occurs.

Because, we are remember that I am now putting it in a single instruction, it better be sit. So, complicated comparison may affect your CPI and these found in VAX. This is the PA-RISK processers here is, MIPS, Alpha. So, notice that, MIPS has both these flavors and actually mixing both of this. When, you take about the MIPS ISN, turns out of the large number of comparisons are actually against 0.

So, this is probably the most frequent comparison. That you would like to know, whether it is less than 0, greater than 0 in all on, whether equal to 0 and, so on. And also less than and less than equal too are very frequent, because of the loop control, the loop control is usually less than or less than equal to say you are tell me, if i is less than or less than equal to my upper boundary, procedure quoted.

So, what must happen on a quote, so some of SPARC already know, what quote has be happen. So, the most important thing that has to happen is, you need to save the return address somewhere. That you can come back to the right place, normally, it is the dedicated link register or some arbitrary general purpose registers. So, this particular term is used in the MIPS word the link register.

So, I want to introduce you to this and will actually use this also later in the course. But, anyway the point is that there is usually a dedicator register or it would pick up one of the general purpose registers. And put your return address there, also you may need to set up the parameters that are going to be past to the procedure. Some architecture simplicity do this as far of the call, while most generate explicit code for this.

So, some of the architecture for example, MIPS would actually generate explicit code for doing passing the parameter and exactly how you pass the parameter will depend on the architecture will talk about that later. So, you may want to pass the parameter to some register, you may want pass the parameter to memory, so you can do maybe otherwise. The caller meaning that from wherever you are calling the procedure may want to save some registers.

So, that the caller cannot destroy them by calling I mean the procedure itself. So, for example, I might be doing some computation and now I need to call a procedure. But, I will require these computations after the procedure, so I want to save the registers, where I have results of these computations. So, this is known as caller saving convention. So, the caller before calling the procedure would say whatever is important.

Symmetrically, you can think of a situation, where a callee may save any register. That it wants to use in later part of the procedure. So, what may happen is that caller says that well I do not care.

So, if let the callee do whatever, if it need to modify, so the callee before starting he figures out, these are registers ((Refer Time: 38:24)) modify, so let be save them first, that I can restore them at the end of the procedure. So, this is known as callee saving convention, so both of these are fine you can use mix of them. So, callee may save certain registers, callee may save certain registers and so on.

And of course, it may be hard for the compiler to decide what to use because, often we learn your compiler course inter procedural analysis is hard. So, when you are calling a function it may be very difficult to figure out, what the register is that, the procedure, you are calling will actually modified, that means we have to figure out most. So, architectures offer both today we clearly specified caller saved and callee saved register sets.

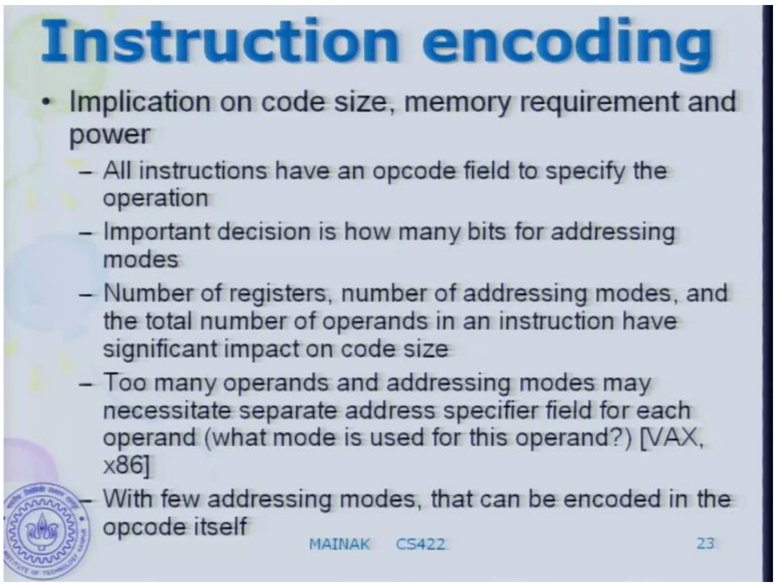
So, what happens ultimately is that to make the compiler job easier, architecture actually specified especially. So, am talking very much in terms of MIPS here x 86 is actually does not have any of these notions. In MIPS architecture what you say is that, I have this four registers, which are caller sets meaning that before you call a function, the caller has to save this four registers. In other words, the callee is free to modify this register, without even a worring about it that is what it really means.

Similarly, it would also have this set of your for registers, which are callee saved register meaning that, the callee before starting the procedure will actually save this registers, so caller did not worry about these registers. And which in other word, which is says is that the callee saved register contents will be restored at the end of the procedure. So, if the caller can if it wants to save something it can use these registers.

Student: ((Refer Time: 40:11))

Yes, this is the most flexible one, this giving compiler full freedom of picking whatever register it wants to be.

(Refer Slide Time: 40:36)



Instruction encoding

- Implication on code size, memory requirement and power
  - All instructions have an opcode field to specify the operation
  - Important decision is how many bits for addressing modes
  - Number of registers, number of addressing modes, and the total number of operands in an instruction have significant impact on code size
  - Too many operands and addressing modes may necessitate separate address specifier field for each operand (what mode is used for this operand?) [VAX, x86]
  - With few addressing modes, that can be encoded in the opcode itself

MAINAK CS422 23

So, combining all these ideas essentially what you finally, want is an instruction encoding get that you have defined certain instructions, we are already define in what instruction would have about operands. And how much of the displacement you would require; what addressing modes you require and finally, what you want is the encoding of the instruction. That how many bits, I should have instruction what fills my instruction should have on and so on and so forth.

So, it has an implication code size memory requirement and power, all instruction have an op code field to specify the operations. So, this is mandatory in your instruction you should have an op code bit, which says what this instruction is, is add operation, is it a branch operation, is it a load operation. What is it actually? Important decision is, how many bits for addressing modes, because, this one is easy if figure out, how many instruction, how many operations, you want to support log off.

That would be this, this many bits required for the op code as simple as that. How many bits for addressing modes also number of registers, number of addressing modes and total number of operands in instruction have significant impact on code size. Because, number of registers would determining how many bits is require to address the register because, log of that would be the number of bits. You would require to address the register.

Number of addressing modes would essentially decides how many bits you would require for a displacement or any other encoding of the addressing mode or even in the worst case you may have to do following that we are, so many addressing modes that you will have to special bits to specify what addressing mode is this. Like, if we have you know 32 addressing modes, you will have to actually reserve 5 bits in an instruction and say, this is using that jumbo mumbo addressing mode addressing mode actually.

So, those 5 bits will have to be there. And total number of operands will of course, will have to go in the instructions like for example, if you have if it is decided that my instructions are going to be very long, it is going to 6 operands. Even if you have 32 registers and all the operands are the register operands, you are going to require 30 bits to specify this operands, ((Refer Time: 42:40)) you have 6 operands.

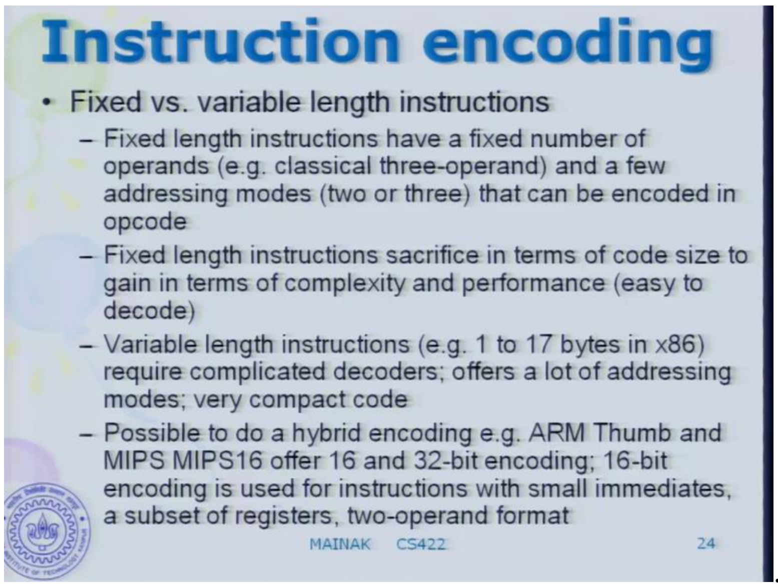
So, there is a limit on this that, you know, how many operands you can really, really encode of course, it is always good to have large number of operands. Because; that means, you can execute complicated instruction in a single instruction, so overall we have to wave all these things with the CPI and as I told you the usual method is that, you would finally, figure out 2, 3, 4 candidates instruction sets. We probably have compiler for all 4, it compile your bench mark into all this codes.

And it actually stimulate all these four possible binaries, see the final performance and decide which one finally, goes. Too many operands and addressing modes may necessitate separate address specified field for each operand. For example, what mode is used for this operand, this is another thing, we can have an actually cross production, we have 6 operands, we have 32 addressing modes, you can say operand one uses addressing mode 30 operand two you say that addressing mode 25.

So, you can actually think of these across production now. So, now that they you know very quickly explores your instruction size, so x 86 has always things actually, which is why x 86 actually has variable length of instructions, instruction length is actually now fixed. With few addressing modes certainly encoded in the Op-code itself, usually this is what you really want.

So, you can send that, well I have two addressing modes and if I have you know 4 load operation, I just have double the numbers I will have 8 load operation and I will actually have of course, for separate addressing modes that is easy in fact.

(Refer Slide Time: 44:36)


Instruction encoding

- Fixed vs. variable length instructions
  - Fixed length instructions have a fixed number of operands (e.g. classical three-operand) and a few addressing modes (two or three) that can be encoded in opcode
  - Fixed length instructions sacrifice in terms of code size to gain in terms of complexity and performance (easy to decode)
  - Variable length instructions (e.g. 1 to 17 bytes in x86) require complicated decoders; offers a lot of addressing modes; very compact code
  - Possible to do a hybrid encoding e.g. ARM Thumb and MIPS MIPS16 offer 16 and 32-bit encoding; 16-bit encoding is used for instructions with small immediates, a subset of registers, two-operand format

MAINAK CS422 24

.

So, one more thing that we have to decide at this point is whether you are going to have a fixed instruction length for all instructions or you are going to vary the instruction size for different instructions. So, fixed length instructions have a fixed number of operands like you have a classical 3 operand instructions and a few addressing modes 2 or 3. That can be encoded in the op code. So, essentially the point is that all my instructions are going to have a constant lengths.

That helps the instruction fetcher a lot, because it knows that if I fetch 100 bytes I can calculate how many instructions I have fetched. Because, I know the length of each instruction, also fixed length instruction satisfies in terms of code size to gain in terms of complexity and performance. Because, essentially what you doing is that, you might be aviating bits, you make your all instructions of equal sides. Because, but may happen is a some instruction he could actually encoding with smaller bits.

But, you would actually increase them to this boundary to make sure all the instructions are of equal lengths. But, of course, you gave in terms of complex performance, because, now they are easier to de code because, you know that, I have fetch 4 bytes, which means I have one instruction now, I can start de coding. And the other hand, if you are variable length instructions we have to keep on fetching. Until, you have the full instruction and that is very difficult to know when we have a full instruction.

So, variable length instructions for example, you have x 86 having 1 to 17 bits of instructions, you can have all for possibilities actually required complicated de coders of us of addressing modes and produces a very compact code. So, that is the advantage fixed line encoding. So, MIPS 16 essentially say that, well if I can encoded instruction a bite I am going to do that, I am not going to increase it to match some constant like for everybody.

And also possibly do a hybridist coding, like your ARM Thumb MIPS 16, offers 16 and 32 bit encoding 16 bit. Encoding is used for instruction fetch small mediates, subset of registers and two opponent format. Whereas, if you want to support in a more operands, bigger register number, you switch to according to ((Refer Time: 46:49)).