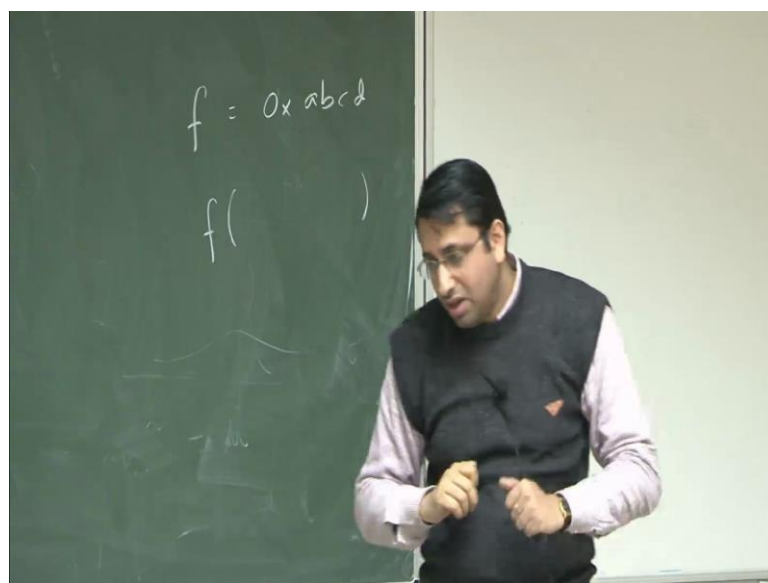


Computer Architecture
Prof. Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 05
Instruction Set Architecture, Case Study with MIPS-I

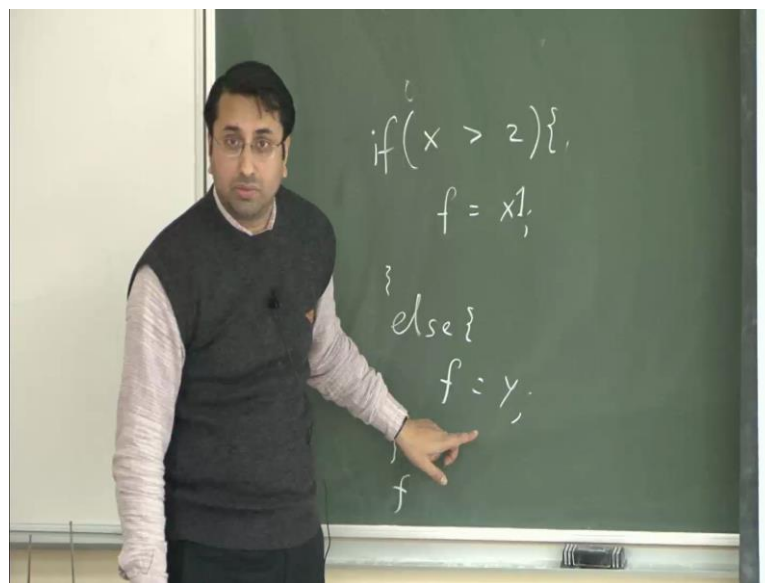
So, I recede the actually of due... Remains are a few questions at the end of last class. Later, I will still need a little more time ((Refer Time: 00:20)) what actually a function binaries, and what I really meant by compiling the switch case statements to function point or indirect jumps. So, how many of you people know about C pointers? Raise hands. Everybody knows about C pointers? So, C pointers are essentially a data type. And as you can guess, a variable of every data type ((Refer Time: 00:53)) value. So, the pointers values are slightly specially the ((Refer Time: 00:59)) addresses. So, if a pointer variable has a value 10; that means we are talking about address 10. And if there is a star in front of the pointer, that means the value at address 10. Function pointers are just pointers. They can ((Refer Time: 01:23)) the only specialty about function pointers is that, these values of course addresses. These are instructions. That is the only specially ((Refer Time: 01:34))

(Refer Slide Time: 01:42)



So, if we have function pointer `f`; and if we assign it a value something – `a b c d`; and then if you call `f` following whatever syntax we require call a function pointer; it will take you to this particular address, whatever address that may be. And whatever instruction is at that address; from that instruction, the ((Refer Time: 02:05)) will start executing. Essentially, the program counter gets changed to this particular value; that is all it means. That is a function pointer.

(Refer Slide Time: 02:20)



Now, why would you use function pointers? Because, it might have quote the programs like this; which may say like if x greater than 2; before we execute one function here; else... So, you may have a function here; you want to call a function when x is greater than 2; else, you may want to call some other function. So, what we may do is – in the function pointers, we assign f to some value and f to some other value here. So, we may give f equal to x and f equal to y; and then we may call f here... So, then depending on what the value of x is... It is x 1. You ((Refer Time: 03:06)) go to the location x 1 y; we start executing depending on the value of x. Clear – what the function pointer is? So, it is clear that, when the compiler is compiling this program, it is impossible to know what this call is going to be. It is a procedure call; that is for ((Refer Time: 03:25)) But, you do not know the target of the call; target would be one of these two; which is why we have to compile it with certain different way.

Essentially, what will happen is that, depending on the outcome of this branch, the value f will be loading to a resistor, And this procedure call will be a ((Refer Time: 03:45)) call, which will take the argument from the resistor. So, that is the indirect procedure calls, Here I actually club them all under indirect jumps. So, these are unconditional jumps by the way. This is an unconditional jump although its value depends on some condition. Now, we talked about this switch case statement last time. Who does not know about the switch case statement? Raise hands.

(Refer Slide Time: 04:24)



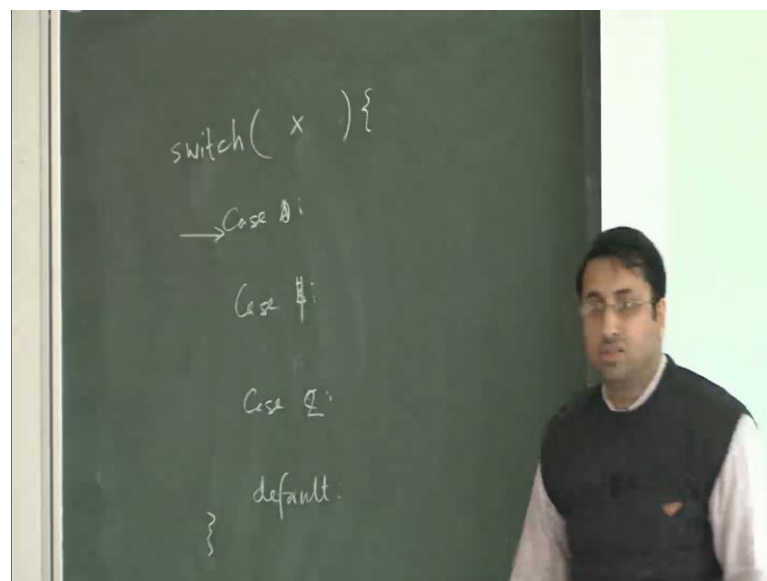
So, typically, a switch case statement would look like this. You will switch on sum value – sum variable; and you have a bunch of cases: case A, case B, case C and there is a default at the end. Now, what I mentioned last time is that, the compiler prepares a table. That is also called a jump table. So, for example, let us first assure that, these are continuous. So, let suppose: 0, 1, 2, 3. These are continuous.

(Refer Slide Time: 05:05)



So, then what will happen is that, the compiler will prepare a table. This is index 0; this is index 1; this is 2, this is 3; that is ((Refer Time: 05:14)) an array. And what will be the content of the array? So, location 0 will contain this instruction address. This will be essentially the instruction – starting address of case A. This will be starting address of case B; starting address case C and the default. Now, when the value comes up; so let us suppose you have switch x. So, what we will do is; it will use x as an index into this array. The only problem is the default part; because if it is 0, it is not 0, 1, 2; then it should be 3. So, you generally compile... Compiler will actually do that. And you can jump to... You can just use the pointers. So, you can just say this ((Refer Time: 06:07)) array. Let us suppose just call this j T – jump table. It will set your... Essentially, it is j T x as a function pointer; and then jumps to that location. Is that clear to you? Now, problem arises when these are not continuous. Suppose I say case 0, case 5, case 9 and default; how do you compile that? Here we have continuous values.

(Refer Slide Time: 06:50)



So, here we actually have 0, 1, 2 – case 0, case 1, case 2. It is not ((Refer Time: 06:54)) how you compile it. Any suggestion?

Student: ((Refer Time: 07:03))

Something simpler? If I allow you to waste space, what will you do? What is the simplest thing you do?

Student: ((Refer Time: 07:14))

No, not ((Refer Time: 07:16)) I have seen what a jump table is; but, I can tell you that, you can waste space, there is no problem.

Student: Maximum value array...

Exactly; we will find out the range – minimum to maximum. And essentially what you will do is ((Refer Time: 07:28)) Minimum will be up to 0; maximum will be up to whatever maximize ((Refer Time: 07:34)) 1 or whatever it is. And you will do this. Only thing is that, there are lot of default values in a grid. It will be a very large table. Let us see what.

Student: ((Refer Time: 07:47))

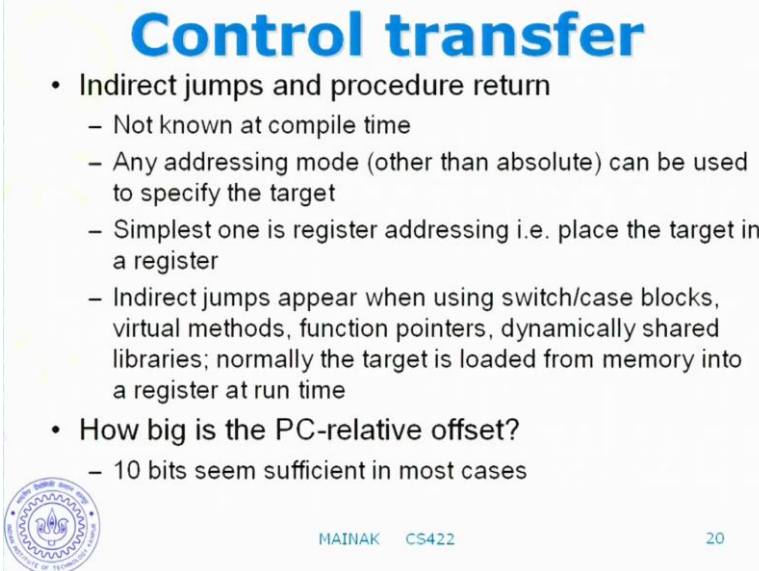
Any other option?

Student: ((Refer Time: 07:50))

What else can you say?

Student: ((Refer Time: 07:52))

(Refer Slide Time: 08:07)



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

Let us take concrete values here. Suppose this is minus 1, 5, 9. Guess.

Student: First, maximum from minus 1 to 9 ((Refer Time: 08:19))

What you mean by ((Refer Time: 08:20))

Student: Means previous case...

((Refer Time: 08:23)) That is what we say.

Student: And there will be a ((Refer Time: 08:27)) which are present to continuous ((Refer Time: 08:30))

Then, what do I say? Why do I need the secondary? I have the table ((Refer Time: 08:35)) minus 1 to minus... They are not ((Refer Time: 08:39)) Why do I need the secondary?

Student: ((Refer Time: 08:42))

That already we have seen in the first table.

Student: Sir, we can see how many cases are ((Refer Time: 08:51)) And we get the table of that size; and then we can take modulus of the cases. Suppose like we have size as 4...

So, another table of 4 entries?

Student: Yeah. So, we can map... – like one if it comes; then it will go to first. And then if it comes 8 or... Then, it will go to 0 ((Refer Time: 09:18)) Suppose that 0 in ((Refer Time: 09:30))

So, module of 4 you want; 5 and 9 will map to the same entry.

Student: So, we will...

So, what we will do?

Student: So, we will use 5 in place of 1 and we will put 9 in whatever the subsequent ((Refer Time: 09:45)) is free.

How you look up? Suppose I have 9.

Student: Similar way.

Now, how do I know that I have to do the second...

Student: We will go to ((Refer Time: 09:58)) And if it is...

It is the only field in this case – 5. So, how do I do the ((Refer Time: 10:06))

Student: Because there are 5 ((Refer Time: 10:09))

No, there are two things. Compiler will set up the table; and in the run time, it will look up the table. So, in run time, suppose the value of x is 9; I will look up the table; I will go to the entry, where 5 is currently allocated. But, I actually have to pick up the value from the next table. How do I do that? Actually, I have to ((Refer Time: 10:31))

Student: ((Refer Time: 10:34))

So, the value also...

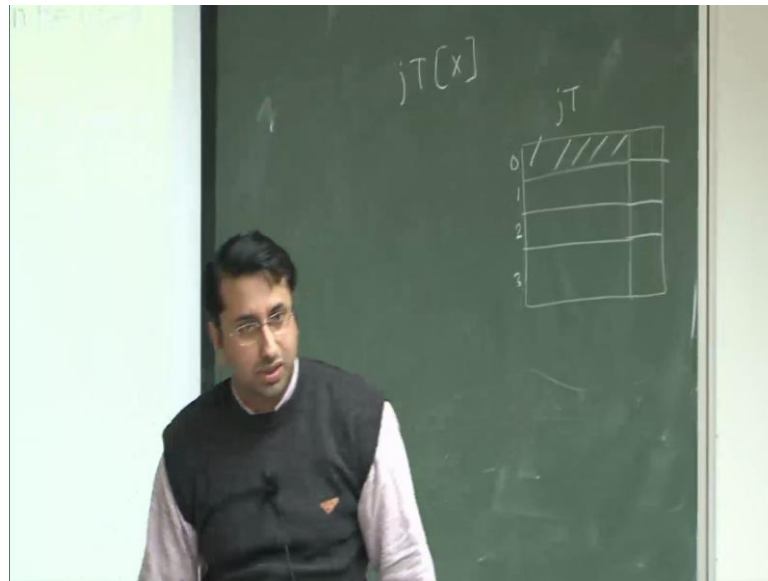
Student: That will be $2n$ ((Refer Time: 10:40))

$2n$? How do you get that?

Student: n for addresses ((Refer Time: 10:46))

Okay; fine; yes.

(Refer Slide Time: 10:53)



So, you are essentially saying that let each table entry have a field, which is actually the case value. That is fine. But, how do I know how many collisions I am going to have? How many ((Refer Time: 11:01)) we allocate actually in the tables? How many entries we allocate?

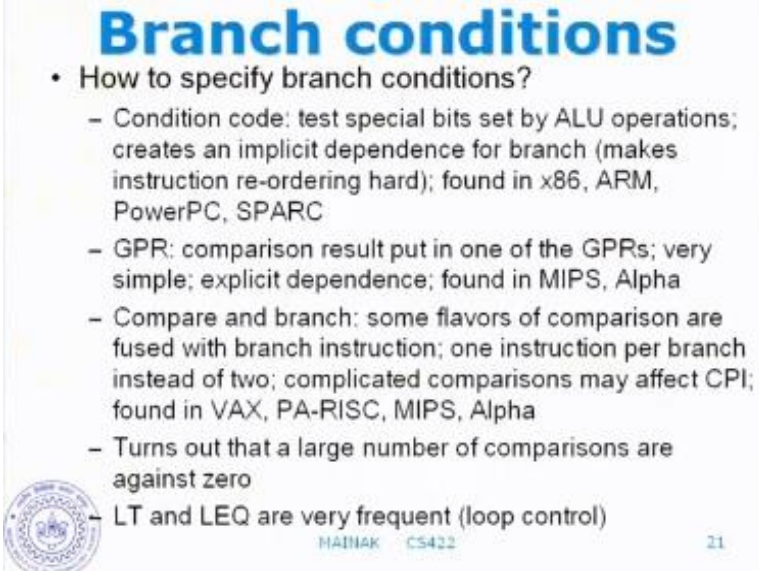
Student: It depends on the number of cases.

It depends on number of cases. So, essentially, what I will do is. So, what you are suggesting is that, in the worst case, I have a ((Refer Time: 11:18)) In the worst case. So, that is a hash table approach. Anything else can you do to reduce the worst case? Of course, on an average, hash table ((Refer Time: 11:37)) What is this problem actually? Fundamental... What is this? What am I going to do? I have a value ((Refer Time: 11:48)) I have a range. What am I doing actually? It is a ((Refer Time: 11:53)) problem. So, what do you do in ((Refer Time: 11:55))

I can solve these values; I can put the values in the table. Like we have mentioned, I have a field here. Then, I can do a binary search on this. So, that was digression. I just want to give you a little more information about what really goes on. But, here the point is that, it is going to be an indirect jump. In the compiled time, I do not know what to do; because,

I do not know the values, where the index is. So, it will be ultimately an indirect jump.

(Refer Slide Time: 12:27)



Branch conditions

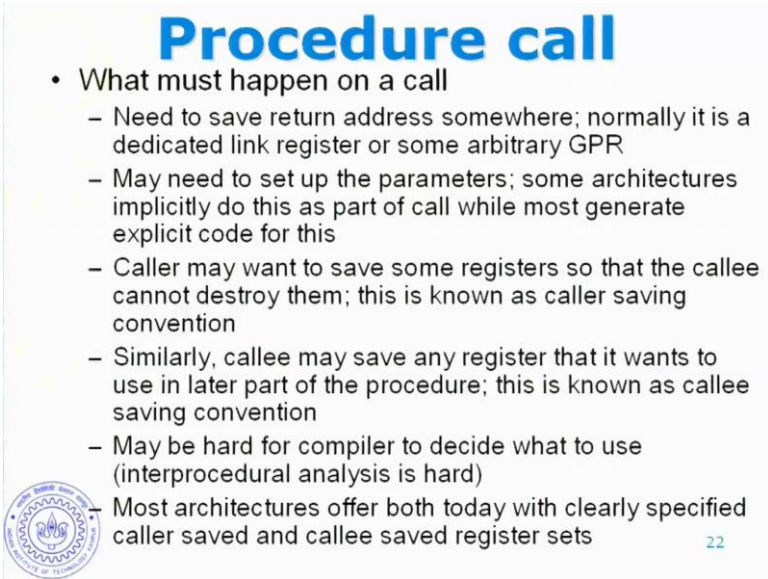
- How to specify branch conditions?
 - Condition code: test special bits set by ALU operations; creates an implicit dependence for branch (makes instruction re-ordering hard); found in x86, ARM, PowerPC, SPARC
 - GPR: comparison result put in one of the GPRs; very simple; explicit dependence; found in MIPS, Alpha
 - Compare and branch: some flavors of comparison are fused with branch instruction; one instruction per branch instead of two; complicated comparisons may affect CPI; 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)

Logo: Institute of Technology, NITK

MAHAK CS432 21

So, we talked about this – branch conditions.

(Refer Slide Time: 12:32)



Procedure call

- What must happen on a call
 - Need to save return address somewhere; normally it is a dedicated link register or some arbitrary GPR
 - May need to set up the parameters; some architectures implicitly do this as part of call while most generate explicit code for this
 - Caller may want to save some registers so that the callee cannot destroy them; this is known as caller saving convention
 - Similarly, callee may save any register that it wants to use in later part of the procedure; this is known as callee saving convention
 - May be hard for compiler to decide what to use (interprocedural analysis is hard)
 - Most architectures offer both today with clearly specified caller saved and callee saved register sets

Logo: Institute of Technology, NITK

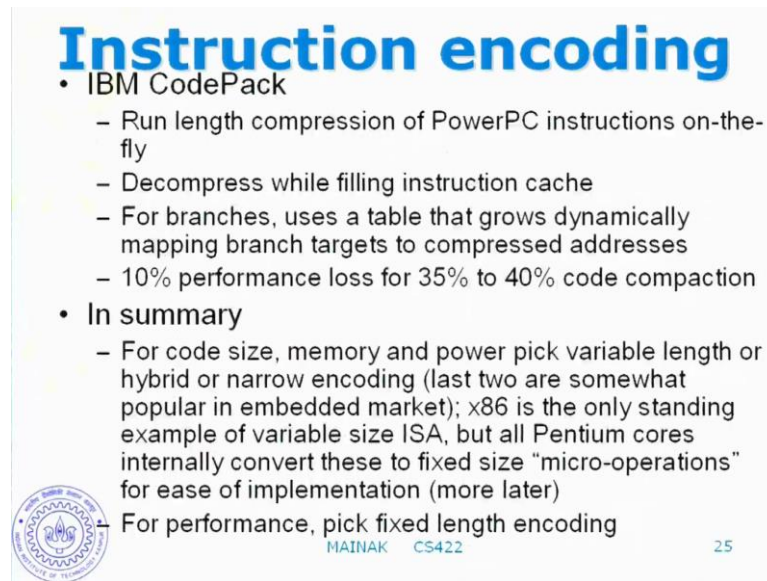
22

This one also we talked about. I also had a few questions about caller saved registers and

callee saved registers. So, this ((Refer Time: 12:40)) is simple actually. So, essentially, the point here is that, here we are talking about two functions: f 1 and f 2. So, this is the body of f 1. At some point it is called f 2. And here the function f 2 somewhere. So, f 1 is a caller of f 2 and f 2 is a callee. So, what essentially it is saying is that, if f 1 wants certain register contents to be preserved after f 2 returns; then it has to save these registers. So, these are called caller saved registers. So, before it calls f 2, it will save those registers because it expects the contents to be preserved when it comes back. On the other hands, symmetrically; if f 2 knows the state of caller saved registers; it knows that, it can go and actually overwrite those registers or use those registers without care. So, that is a two-way contract; that f 1 saved something; and f 2 can use that without caring out.

Similarly, callee saved registers are essentially those registers, which the caller might expect to be preserved. So, caller may expect certain registers to be preserved. Of course, callers could save them. But, in a convention... It is a different convention, which is callee saved convention. You can say that, I would put this responsibility on the callee; because the callee needs to modify these registers. So, it should save this actually for me. So, these are the two conventions: caller saving convention and callee saving convention. So, most architectures has not seen ((Refer Time: 14:25)) They support both. So, the compiler can choose whichever ((Refer Time: 14:31))

(Refer Slide Time: 14:39)



Instruction encoding

- IBM CodePack
 - Run length compression of PowerPC instructions on-the-fly
 - Decompress while filling instruction cache
 - For branches, uses a table that grows dynamically mapping branch targets to compressed addresses
 - 10% performance loss for 35% to 40% code compaction
- In summary
 - For code size, memory and power pick variable length or hybrid or narrow encoding (last two are somewhat popular in embedded market); x86 is the only standing example of variable size ISA, but all Pentium cores internally convert these to fixed size “micro-operations” for ease of implementation (more later)
 - For performance, pick fixed length encoding

MAINAK CS422 25

So, we talked about these also – instruction encoding; both sides: fixed and variable length instructions. So, IBM code pack is an optimization for PowerPC instruction set; where, it does a run length compression of PowerPC instructions on-the-fly. So, it does even know what the run length compression is – run length compression or run length encoding. We are talking about run length encoding. What is it? Anybody guess from the name what it could be. We are trying to compress the ((Refer Time: 15:07))

Student: ((Refer Time: 15:08))

Exactly. So, if I have five 1's – a continuous tree of n 1's; that will be replaced by n followed by 1. So, essentially, instead of n, it is the boundary ((Refer Time: 15:23)) So, that is run length compression of PowerPC instructions that is included in the IBM code pack. And it decompresses while filling instruction cache, because the instruction cache does not really know what the decompressed instruction is. So, before you bring something into the instruction cache, you have to decompress the instruction. So, we can see the trade off now. It takes time to decompress; it is not 0 apparently; that means when you are bringing an instruction until it is decompressed, you cannot start using it. So, use some cycles there. Work... When you are gaining return, you gain in terms of memory footprints. You can actually accommodate beta in ((Refer Time: 16:06)) And

how do we handle branches? That is a big problem here. Because now, a branch instruction; suppose we recounter branch instruction; it says we should go to that target. Now, how do you figure out this target? Because this target would be somewhere else in a complex form; because branch instruction the compiler does not know about this compression – on-the-fly compression. When the program runs, it gets compressed. So, branch instruction would actually have the decompressed target. But, the program has already got compressed on the line. So, how do you figure out where the target is? So, what they do is – it uses the dynamic table that grows gradually mapping branched targets to compressed address. Is this problem cleared already what I am talking about? The branch targets that go inside the instruction are actually decompressed targets. That is referred to the decompressed piece of code. Whereas, after compression, the target has moved somewhere else. You have to figure out where the compressed target is. That is done with a table. On the fly, you actually prepare the stage as and when you encounter branches.

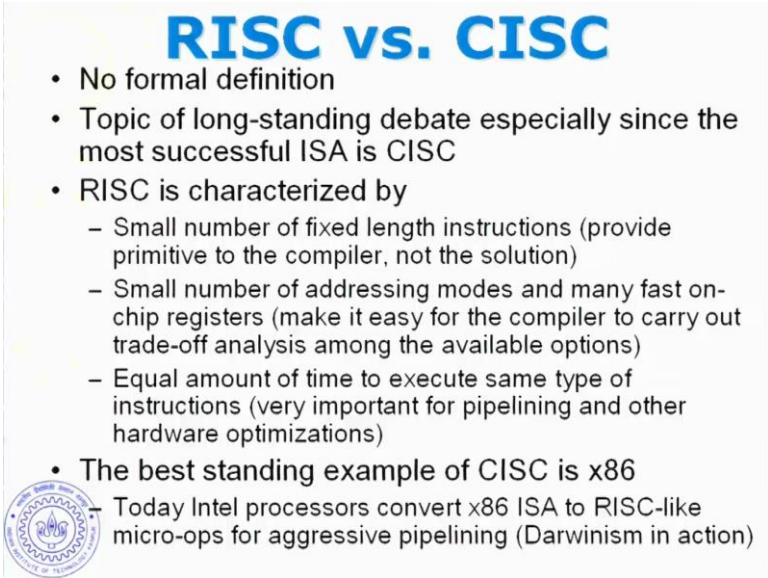
So, these are also takes time; that is pretty clear. So, there also you lose. Wherever you encounter the branch, you lose some time preparing the table and figuring out. However, these are the first time ((Refer Time: 17:27)) In the next time onwards, you look up the table and know ((Refer Time: 17:31)) So, 10 percent performance loss for 35 percent - 40 percent code compaction. That is what they report.

So, in summary, for code size, memory and power pick variable length or hybrid or narrow encoding. Because variable length allows you to get very compact code; as I told, they have been discussed last time. If we have an instruction that can be accommodated in 1 byte, it will actually happen. If will not expand it to meet the maximum limit or the maximum instruction length. Because as per the fixed instruction, it could be ((Refer Time: 18:07)) So, you can expand the smallest instructions to be equal to the largest one actually. So, if you are really worried about your code size; you are really worried about memory footprint; you really power consumption, because these are actually linked. If you are ((Refer Time: 18:26)) more memory, you find more power; that is obvious. You should pick actually the variable length or hybrid encoding. That will also we will talk tomorrow, because we may actually have to be ((Refer Time: 18:35)) depending on the instruction type.

So, last two are somewhat popular in embedded market; we discussed about MIPS-16 last time – MIPS-16 and MIPS-32. x86 is the only standing examples of variable size ISA. However, all Pentium course... I put Pentium here, because Pentium... We have heard of Pentium. On the other hand, we might not have heard of ((Refer Time: 19:00)) which are recent Intel processors. But, the point is that, every Intel processors today, internally, convert these variable length instructions to fixed size micro-operations. You might have heard of this term – micro-operations.

So, that is what the internal architecture actually sees. Internal architecture does not see any x86 instruction. They all get translated internally. And that is done only for the ease of implementation. So, we talk about implementation norm as you see there. So, the point is that, today, if you take any processor, internal architecture actually handles fixed length instructions – all architectures. However, they will doing under a ((Refer Time: 19:44)) Whatever they look like outside to the compiler, that may be very different actually. If you are really worried about performance, pick fixed length encoding. That saves lot of cycles, which makes you hardware simpler; and simpler is faster ((Refer Time: 20:01))

(Refer Slide Time: 20:03)

A presentation slide titled "RISC vs. CISC" in large blue font. The slide contains a bulleted list of points. The first point is "No formal definition". The second point is "Topic of long-standing debate especially since the most successful ISA is CISC". The third point is "RISC is characterized by", followed by three sub-points: "Small number of fixed length instructions (provide primitive to the compiler, not the solution)", "Small number of addressing modes and many fast on-chip registers (make it easy for the compiler to carry out trade-off analysis among the available options)", and "Equal amount of time to execute same type of instructions (very important for pipelining and other hardware optimizations)". The fourth point is "The best standing example of CISC is x86", followed by a sub-point: "Today Intel processors convert x86 ISA to RISC-like micro-ops for aggressive pipelining (Darwinism in action)". In the bottom left corner, there is a circular logo of the Indian Institute of Technology (IIT) Bombay.

RISC vs. CISC

- No formal definition
- Topic of long-standing debate especially since the most successful ISA is CISC
- RISC is characterized by
 - Small number of fixed length instructions (provide primitive to the compiler, not the solution)
 - Small number of addressing modes and many fast on-chip registers (make it easy for the compiler to carry out trade-off analysis among the available options)
 - Equal amount of time to execute same type of instructions (very important for pipelining and other hardware optimizations)
- The best standing example of CISC is x86
 - Today Intel processors convert x86 ISA to RISC-like micro-ops for aggressive pipelining (Darwinism in action)

So, our time... In fact, starting from the first day of history in computing, there is a

debate between RISC and CISC. So, RISC stands for reduced instruction set computers; whereas, CISC stands for complex instruction set computers. So, there is as such no formal definitions for these two things. And as you can guess from the names, the RISC computers would actually have a very minimal view of computing. They would pick the minimal set of instructions that are enough to carry out all possible computations. So, for example, RISC will not try to offer you compound instructions; that can do many things in single instruction; they will never do that. They will say you can implement exactly. ((Refer Time: 20:54)) They will only support with minimal set of instructions. But, a CISC would actually go further and implement very very complicated instructions that can be done in a single instruction. Like for example, you can copy – as we have discussed last time also. All of you knows that; we copy a string from one place to another; which can actually be eliminated by a sequence of my copies.

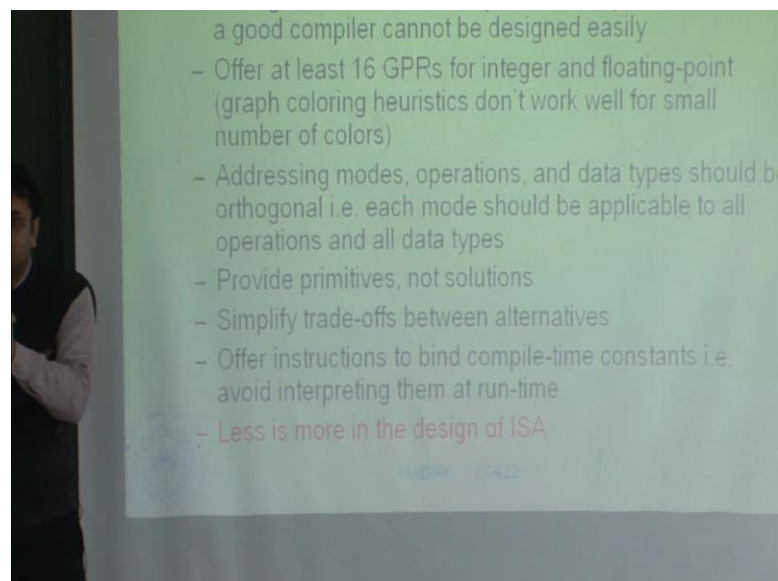
So, it is a topic of non-standing debate; especially sees the most successful ISA CISC, which is x86 is a CISC ISA ((Refer Time: 21:27)). So, people often here would tell that, well it is good, it is minimal and we have beautiful properties; but, who wants CISC. But, anyway... So, here are some characteristics of CISC. I put it here because internally, as I just told you, internally, the architectures today all handle these instructions here. The ISA expose to the compiler for x86 – maybe CISC. But, internally, they actually get translated to RISC micro-operations. So, I put it here also because ((Refer Time: 22:02)) which is one of the most popular RISC instructions in architectures. And also, will actually weighs our micro-architecture design, pipeline design around a RISC architecture, because that is what ultimately gets into in every processor today.

So, what are some characteristics? We have small number of fixed length instructions. So, here the philosophy is that, we provide primitive to the compiler, not the solution. With the compiler, actually it will meet the solution. Small number of addressing modes and many fast on-chip registers make it easy for the compiler to carry out trade-off analysis among the available options. These also we talked about earlier that, if the compiler has too many options in the difficulty in picking the right one; and actually, there is a higher chance to make a mistake in picking the right one. So, let the designer figure out what are the good options actually and offer only those, so that the compiler will always pick always the good option. And of course, if you cannot emphasize more

about having many fast on-chip registers; they will actually improve performance ((Refer Time: 23:08))

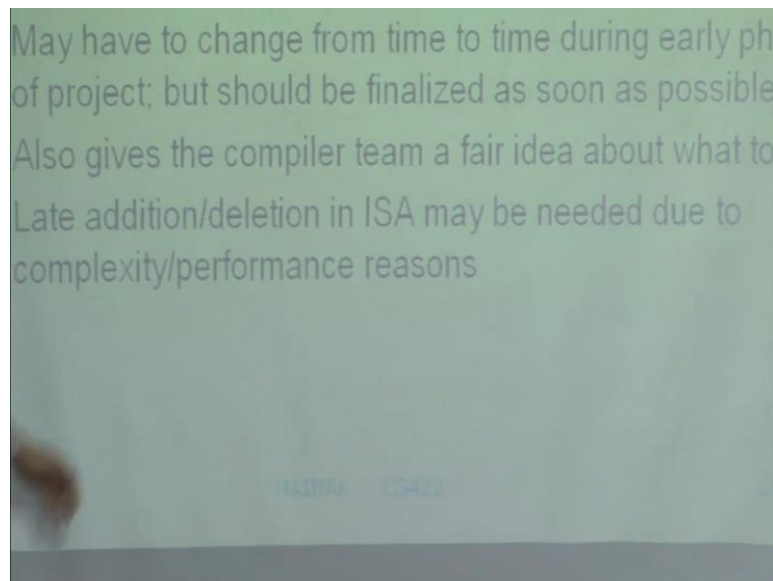
Equal amount of time to execute same type of instructions – this is again very important for pipelining and other hardware optimizations. And again this philosophy applies here. If you have same type of instructions to take same amount of time, compiler can actually pick options quickly; they can weigh options easily. That is why I have two – these two possible implementations; which one should I pick? That should be easier for you to actually select. The best standing example of CISC is x86. However, today, intel processors convert x86 ISA RISC-like micro-ops for aggressive pipeline. So, anyways... So, the point is that, when you are designing an instruction set, this is the bottom line – help the compiler line; because if you cannot help the compiler designer, you are going to have a shabby compiler. If you have a shabby compiler, your processor is going to run shabby core. If the processor is going to run a shabby core, it will report bad performance; that looks bad actually. People will say that, well, look that, your processor is so bad; the problem may be that, we have to choose the bad instruction set; which is why you could not design a good compiler. So, the compiler is very important today. Your great architecture may dramatically lose in market if a good compiler cannot be designed.

(Refer Slide Time: 24:29)



Provide primitives, not solutions as I told you. Simplify trade-offs between alternatives. Offer instructions to bind compile-time constants, that is, avoid interpreting them at run time. So, what this means is that; so essentially, what we are talking about is that, your instruction – there should be fields to carry constants; that the immediate values are talked about ((Refer Time: 24:47)) If you do not have those, what will end up happening is that, you may have a constant ((Refer Time: 24:52)) in your program. You should repeatedly get loaded from memory into a register. That is not good actually. That harms performance. So, has some way of actually having compile-time constants; that is, meaning that, you should have space in instruction itself. So, less is going the design of ISA. So, that is the whole ((Refer Time: 25:12)) So, instructions and architecture is a first step to design a processor.

(Refer Slide Time: 25:23)



Decide what instructions are important to make the change from time to time during the early phase of project. That should be finalized as soon as possible, because this gives the compiler team a fair idea about what to do. Late addition, deletion in ISA; may be needed due to complexity, performance reasons, because it may happened that, as I told you, architects are dreamers. So, they would dream about certain nice things. And finally, at the later face of the design, it will go to the circuit designers. And they will say, oh I cannot implement this; it is impossible actually ((Refer Time: 25:51)). So, at

that point, of course, the architect will push it as hard as he or she can; but, it may happen that it is just impossible. So, at that point, it comes back to the architect saying that, you should go and modify the instruction. Either delete it or break it up into smaller ones, something like that. So, that may be possible; but, that should be avoided as much as you can; which is why an architect should have some idea about how complex the circuit is going to be ((Refer Time: 26:19)) Question on instruction sets?

((Refer Slide Time: 26:23))



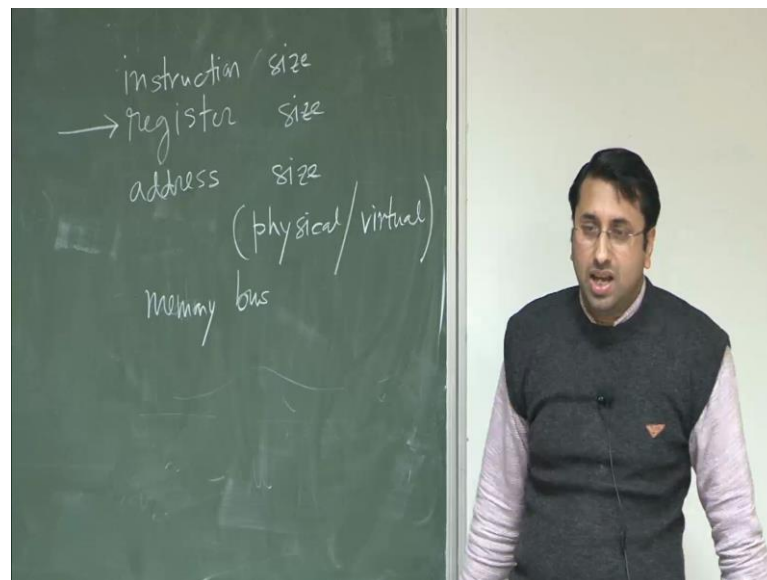
So, these all let me ((Refer Time: 26:24)) So, we will walk you through subset of 32-bit ISA just to substantiate some of the ideas that we just discussed. And we will also get see what are concrete instruction set; what it does and what it does not actually. But, before that, any question? Anything that you want to ask; anything that you want to know? So, there are four families of each instruction set – 1, 2, 3, 4. Backward compatible meaning that, MIPS-4 is a superset, MIPS-3 is the subset of MIPS 4; MIPS-2 is the subset of MIPS-3, and so on and so forth. We will focus on MIPS-1 only, which is a 32-bit ISA. So, what do I mean when I say an ISA is 32 bits. What does ISA stand for? It is an acronym.

Student: ((Refer Time: 27:18))

Instruction set architecture. What do I mean when I say it is a 32-bit ISA?

Student: ((Refer Time: 27:26))

((Refer Slide Time: 27:28))



Instruction size – any other idea?

Student: ((Refer Time: 27:37))

The register size. Anything else? Register size, instruction size; what else? I heard something about memory somewhere.

Student: Memory size

Memory size cannot be 32-bits.

Student: Memory address

Address size – physical or virtual; physical address or virtual address. So, she says physical; you say virtual. So, I will put both – physical or virtual. Anything else?

Student: ((Refer Time: 28:48))


That is the address size; you mean the data bus? Data bus. What else? So, maybe the memory bus – maybe I should put memory bus; data bus is ((Refer Time: 29:09)) This is the bus that carries bits of memory to the processor. These busses are ((Refer Time: 29:21)) What else can be... What else can we think of? We had instruction size; we have register size; we have address size – physical or virtual. We have one more dimension here – instruction address or data address; that does not matter. Okay, I will spare you on that maybe – memory bus. What else? So, should I take a board?

Let us take one by one; or, maybe I will give you a correct answer to you. This is the register size. So, we have a 32-bit ISA means the register size is 32-bit, which automatically decides the data path within the processor, because the data paths actually originate at registers, go through probably the caches ((Refer Time: 30:24)) registers. These registers decide how wide the data path we require. A 32-bit machine has no meaning to have a 64-bit data path. That is pretty clear, because your register size is 32-bit. It can have nano data path; that is possible; but, that actually comes with performance ((Refer Time: 30:42)) because to know the register, you have to switch the

bus twice. Has nothing to do with physical address size. I can run a 32-bit processor on arbitrary amount of physical memory; I do not care. However, register size decides the virtual address size. So, if the register size is 32 bits, you cannot generate larger virtual addresses, because virtual addresses get generated through program code – in program code, take data from the registers. So, virtual addresses will get translated to generate a physical address depending on how much physical memory get installed on the computer. This may be more than 32 bits; this may be less than 32 bits; that depends on the virtual or physical address.

Memory bus has nothing to do with this. How much data you want to carry from memory to processor in one go – is completely ((Refer Time: 31:43)) to how many bits the register has or how many bits the address bus has; nothing to do with that. I can have a very narrow bus and decide that. Well, I am going to switch this bus multiple times to fill up one cache line; that is okay; there is no problem. I can have a very wide bus; I can say that, well I can fill up four cache lines together. That is also fine; there is no problem. It has nothing to do with the ISA width. And instruction size has nothing to do with this. For example, you have ((Refer Time: 32:12)) I did not expect that. x86 has instruction size ranging from 1 byte to 17 bytes. It includes 32-bit ISA, even a 64-bit ISA – both. These are totally ((Refer Time: 32:24)). So, is it clear? We will not do a mistake on this set. These are ((Refer Time: 32:30)) You should know this as a computer scientist. One – 32-bit ISA. So, MIPS-3 onward, a fully 64-bit ISA. The text book has an overview of that.

((Refer Slide Time: 32:44))



Introduction

- Four backward-compatible families: MIPS I, II, III, IV
 - We will focus on MIPS I only (32-bit ISA); what is this?
 - MIPS III onwards are fully 64-bit ISAs (in textbook)
- Load/store register ISA
 - Operates only on registers
 - Can access memory only through load/store (big endian)
 - No partial register access
- RISC philosophy
 - Emphasis on efficient implementation: Make the common case fast
 - Simplicity: provide primitives, not the solutions
 - A system can be so simple that it obviously has no bugs, or so complex that it has no obvious bugs [C. A. R. Hoare]

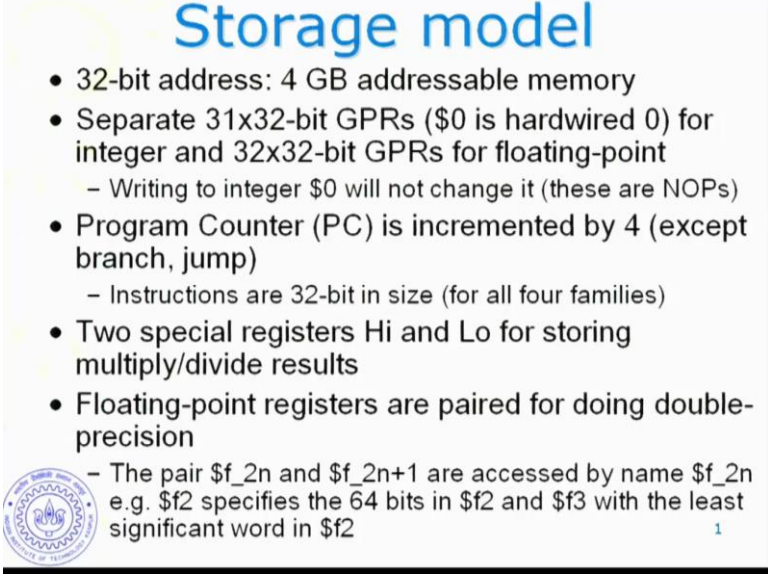
MAINAK CS422

And, actually, I have also posted MIPS-4 instructions set manual on my... If you are really really curious to know what is there; we can actually go through it; but, it is a thick manual. But, it talks about the entire set of instructions. MIPS-4 is a superset. MIPS is a load/store register ISA. What does that mean? MIPS-4 is nothing – load/store register ISA. It says certain things here. Operates only on registers. So, ((Refer Time: 33:20)) instructions can operate only on the registers. Can access memory only through loads/store. And remember that, MIPS is big endian. I hope everybody remembers that, what big endian is. And there is no partial register access. Almost true; not exactly the same. There are four instructions. They actually allow you to do a partial register access units; we will talk about it. So, it follows the RISC philosophy. Emphasis is on efficient implementation; make the common case fast. It takes ((Refer Time: 33:57)) lock as it is and tries to follow that. At every stage, we will find that, it will actually do this; it will not spend hardware on things that are rare.

Simplicity – provides primitives, not the solutions. We will come across this philosophy over and over in this course. So, here is a quotation from Tony Hoare. A system can be so simple that, so that it obviously has no bugs; or, so complex that, it has no obvious bugs. So, I just wanted to put it here; although it has nothing to it needs as such. So, point is that, your design philosophy should be that, it should be simple; do not try to deliberately make things complex. There is a point here. Essentially, you are putting burden on the verification engineers to figure out if the ((Refer Time: 34:48)). So, we

will start with data types just like you start learning a programming language in ((Refer Time: 35:00))

(Refer Slide Time: 35:02)

A presentation slide titled "Storage model" in blue text. It contains a bulleted list of MIPS architecture features. A small circular logo of the University of Hyderabad is in the bottom left corner, and a small number "1" is in the bottom right corner.

Storage model

- 32-bit address: 4 GB addressable memory
- Separate 31x32-bit GPRs (\$0 is hardwired 0) for integer and 32x32-bit GPRs for floating-point
 - Writing to integer \$0 will not change it (these are NOPs)
- Program Counter (PC) is incremented by 4 (except branch, jump)
 - Instructions are 32-bit in size (for all four families)
- Two special registers Hi and Lo for storing multiply/divide results
- Floating-point registers are paired for doing double-precision
 - The pair \$f_{2n} and \$f_{2n+1} are accessed by name \$f_{2n} e.g. \$f2 specifies the 64 bits in \$f2 and \$f3 with the least significant word in \$f2

We will start learning about MIPS ISA with the data types. So, essentially, given that, it is a language for a processor; we will have nothing but bit strings. So, what are the bit strings that are allowed? Byte is 8 bits as you know; half word is 16 bits; word is 32 bits; and double word is 64 bits. So, that is it. These are the only four data types that we have. Integers are represented in 2's complement; if you have forgotten, you should brush it up. Although may not be able to do this much. Floating point – it supports single and double precision compliant with IEEE 754. So, what the storage amount? It has 32-bit addresses. So, when I talk about addresses in the context of an instruction set architecture; that means, in the virtual addresses. It has nothing to do with the physical memory, because ultimately, this MIPS processor will get installed into billions of computers with billion different memory capacity. So, that decides the physical addresses. So, here we can only talk about virtual address size, which is determined by the register size; which is why ISA is. So, it supports 32-bit virtual addresses; which gives you 4 giga byte addressable virtual memory processor.

So, there are separate 31, 32-bit general purpose registers. It is actually there are 32. But,

one of them is hardware constant to 0; you cannot change it. So, they did it because they took ((Refer Time: 36:49)) And they said that, constant 0 is so much that, it actually requires a hardware register to maintain throughout. We cannot generate a 0 constant; it does not make sense; it is so popular. So, that is why there is a register always 0; whenever you need, use it; that is it. You do not have to generate this value 0. On the other hand, in x86, does anybody know what is a popular way of generating 0? Those who have browsed x86 course should have seen that. Anybody? How can we generate 0? That is an interesting course. What are the operations that you can do to generate 0?

Student: Subtract ((Refer Time: 37:38))

Subtract the same register. Anything else?

Student: ((Refer Time: 37:48))

XOR; yes, right. That is usually a popular way of doing it; XOR. That will give you value 0 in ((Refer Time: 37:56)) because usually XOR is much faster than doing the subtraction. Subtraction is basically ((Refer Time: 38:02)) And if you remember about this little carry addition, the carry will little ((Refer Time: 38:08)) Then, subtraction would borrow little ((Refer Time: 38:11)) It will take time actually to get the final answer. XOR is much faster. So, anyway MIPS does not have to worry about these, because it gets the value 0 for free; not quite for free it gets ((Refer Time: 38:25))

The floating point side has 32-bit 0 popular registers. So, we will see how it has to emulate double precessions ((Refer Time: 38:36)) So, riding to integer dollar 0 will not change. So, these are essentially NOTs; you can generate instructions, which actually has destination as dollar zero. So, when it executes, the hardware will actually ((Refer Time: 38:51)) Program counter is incremented by 4, except for branch in jump instruction; which means instruction are of 32 bit in size. In fact, for all four families, they have 32-bit instructions. So, what is that? Instruction size is nothing to do with ISA width. For example, MIPS-4 in 64 bit ISA has 32-bit instructions.

There are two special registers: Hi and Lo for storing multiply/divide results. So, if we

multiply two 32 bit numbers, we will get a 34 bit number. That is the... That is a largest you can get. So, the result gets split into two registers: both of these are 32-bit. So, there we will get the 64-bit multiplication result. And when you do a division, you get a quotient and a remainder right. So, remainder and the quotient will also go into these two registers. Floating point registers are paired for doing double precisions. So, the pair f_{2n} and f_{2n+1} are accessed by the name f_{2n} ; that is, f_{2n} specifies 64 bits in f_{2n} and f_{2n+1} with the least significant word in f_{2n} . So, remember there is ((Refer Time: 40:08)) So, f_{2n} have the least significant word among the 64 bits. Any question on this?

Student: Sir, the hi-lo registers are different from ((Refer Time: 40:18))

Yes; these are extra; yes. These are special purpose registers ((Refer Time: 40:24)) so that they get the result only from the multiplier and the divider.


Student: That is the reason ((Refer Time: 40:29))

Oh, yeah; So, yes, exactly. So, these are all different – totally different; yes. So, essentially, you have 32 general purpose registers; one of which is 0. We have the program counter; we have hi lo. And floating point registers – you have 32 and when you want to double precision, you ((Refer Time: 40:51)) Any question?

((Refer Slide Time: 41:03))

Computation

- ALU instructions
 - Classic 3-operand format: two sources and one destination
 - Operands: GPRs or 16-bit immediate values
 - Both signed and unsigned arithmetic
 - Basic difference between signed and unsigned arithmetic: overflow not flagged for unsigned
 - Sign extension of immediate for both signed and unsigned arithmetic; zero extension for immediates in logical instructions
 - Signed comparison is completely different from unsigned comparison (not true for signed and unsigned add)
 - Integer multiply/divide take only two operands i.e. two sources and have implicit targets Hi and Lo: ISA offers instructions to move from/to Hi/Lo registers to GPR



MAINAK CS422

1

So, how did you do computation on these data types? So, there are ALU instructions. These are classic 3-operand format: two sources and one destination. And what are the operands of these ALU instructions? These are the general purpose registers or 16 bit immediate values. Both signed and unsigned arithmetic are supported. So, what is the... So, basic difference between signed and unsigned arithmetic is that, overflow is not flagged in unsigned arithmetic. Sign extension of immediate for both signed and unsigned arithmetic; zero extension for immediate for logical instructions. So, any other immediate constant inside instructions; what you will do is... So, these constant is going to be inside the instruction. So, in a 32-bit instruction, here is a constant going to be smaller in size. But, the data path is 32 bits. So, this constant would have expanded to 32 bits. So, there are two ((Refer Time: 42:00)) bit; you can expand it to 0's at the top or you can expand it by extending sign bits. So, what they are doing here is that, you have to always extend the sign bit. So, for example... Say if you remember your 2's complement ((Refer Time: 42:20)) So, it is essentially...

If the immediates are let us say... In your instruction, your immediates are represented in 16 bits for example. In that case, what will happen is that, you are talking about 16 bits to 16-bit 2's complement constants. So, the most significant bit will actually signify the sign. So, if it is a negative number; you have a 1 there. That will get extended. So, when you get 32-bit number; you can go back and check. Or, if you already know about it that, if we have a 16-bit negative number; if you put 1's in all the upper places; you get 32-bit

negative number of the same value; the value does not change. And if an upper bit is 0; which means it is a positive number; you just extend it by 0. So, essentially the point is that, you always do a sign extension of the immediate for arithmetic operations.

For logical operations, the immediates are always treated to be positive constants and they are always extended. So, you cannot do logical operation on negatives. So, it does not make any sense. When you are exhorting ((Refer Time: 43:25)) exhorting a bit ((Refer Time: 43:26)) Sign comparison is completely different from unsigned comparison, which is actually not true for signed and unsigned act. So, unsigned comparison would actually treat the values as unsigned values. And signed comparison will actually treat them as negatives or positive depending on its axis.

Integer multiply/divide take only two operands; that is, two sources; and have implicit targets – hi and lo. The instruction set offers instructions to move from or to hi slash lo registers to GPR, because what will happen is that, you do a multiplication. So, now, the result is hi and lo. So, you have to move the hi and lo values to general purpose registers to be ((Refer Time: 44:12)) So, there are instructions for doing that. If they move from hi to a general purpose registers, you can move from lo to general purpose register. Or, you can move the other general purpose register into hi. That is also wrong actually. Although... For work purpose, you will use that. So, here is a list of ALU instructions that, MIPS supports. You can see that, list is pretty small actually; it will fix in one slide. In fact, the whole arithmetic logical...

(Refer Slide Time: 44:48)

ALU instructions

• Arithmetic	• Logical
add \$3, \$2, \$1	and \$3, \$2, \$1
sub \$3, \$2, \$1	or \$3, \$2, \$1
addi \$3, \$2, 100	xor \$3, \$2, \$1
addu \$3, \$2, \$1	nor \$3, \$2, \$1
subu \$3, \$2, \$1	andi \$3, \$2, 10
addiu \$3, \$2, 100	ori \$3, \$2, 10
slt \$3, \$2, \$1	xori \$3, \$2, 10
slti \$3, \$2, 100	sll \$3, \$2, 10
sltu \$3, \$2, \$1	srl \$3, \$2, 10
sltiu \$3, \$2, 100	sra \$3, \$2, 10
mult/multu \$3, \$2	sllv \$3, \$2, \$1
div/divu \$3, \$2 // Lo=q, Hi=r	srlv \$3, \$2, \$1
mfhi \$4	srav \$3, \$2, \$1
mflo \$4	lui \$3, 40

1

So, let us go through each of these just to understand, because we will probably refer to some of these Mnemonics; sometimes in the design. So, this is easy. And so the MIPS registers are always represented as dollar followed by the integers. These are integer registers or sometimes called the general purpose registers. So, these solve the first one; adds a dollar 1 to dollar 2 and puts the result in dollar 3. The second one is the subtract operation – dollar 2 minus dollar 1; could be dollar 3. We have at immediate, these ones. So, this is a constant 100 added to dollar 2. The result goes to dollar 3. Notice that, this constant is actually encoded in the instruction; it does not have to come from ((Refer Time: 45:36)) Add unsigned dollar 1 plus dollar 2; put it dollar 3. Sub unsigned – subtract unsigned; add immediate unsigned. So, remember that, whether you have at immediate unsigned or at immediate, the only difference is the over project; there is no other difference. In both the cases, this will be sign extend – this particular constant.

Set less than; so dollar 2 is less than dollar 1; you put 1 in dollar 3; otherwise, you put 0 in dollar 3. Set less than immediate; dollar 2 less than 100; you put in dollar 3. So, what would be a good application of this? Can you think of s-l-t-i – set less than immediate.

Student: ((Refer Time: 46:27))

((Refer Slide Time: 46:29)) yes, thank you. The ((Refer Time: 46:31)) upper bound check. So, dollar 2 is the loop index; if the upper bound is 100, you will be checking this every time – oh, which dollar 2 crossed 100?

Student: Arithmetic you shift the ((Refer Time: 46:44)) along a... As you... In right shift, you should keep most different ((Refer Time: 46:52)) You maintain the sign of the...

Exactly. So, when you are shifting right, you are creating space on the most significant side. So, you have to figure out what you fill in there. So, shift right logical fill in 0; shift right arithmetic will fill in the sign. So, essentially you fill in the most significant shift as you go on. So, this one shifts by 10 bits; this one also shifts by 10 bits. But, in this case, the final result will be dollar 2 shifted by 10 bits with 0 filled in on the upper 10 bits. Here dollar 2 shifted by 10 bits; upper 10 bits are filled with the sign bit of the ((Refer Time: 47:30)) And then you have shift left logical variable; that is, you want to shift by a

variable amount. The shift amount you can put in the register; shift right logical variable, shift right arithmetic variable and lui. So, this is load upper immediate. These are very interesting instruction. It is called a load upper immediate; which means you load dollar 3 with the upper 16 bits filled with 40. Lower 16 bits are 0. So, these are very useful instruction; we will see the application of this.

Student: The lower 16 bits are set to 0.

Yes.

Student: And unchanged.

No, not unchanged; they are set to 0. So, that is... So, you remember that, this is not a memory operator; just loading a constant and it is actually inside your aim. It is part of the logical operation.