

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

**Lecture - 06**


**Case Study with MIPS-I**

Introduced the basics of the MIPS ISA.

(Refer Slide Time: 00:14)

## 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]




And, we talked about the data types.

(Refer Slide Time: 00:17)

## Data types

- Bit strings
  - Byte is 8 bits
  - Half word is 16 bits
  - Word is 32 bits
  - Double word is 64 bits
- Integers in 2's complement
- Floating-point: single and double precision
  - IEEE 754 standard



MAINAK CS422


1

I will just remind there are four data types byte, half word, word and double word. So, the double word instructions are relevant only if there are 64-bit ISA. And, the 32-bit ISA will only have up to 32-bit instructions – up to word. Integers are 2's complement.

(Refer Slide Time: 00:43)

## 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<sub>2n</sub> and \$f<sub>2n+1</sub> are accessed by name \$f<sub>2n</sub> e.g. \$f2 specifies the 64 bits in \$f2 and \$f3 with the least significant word in \$f2

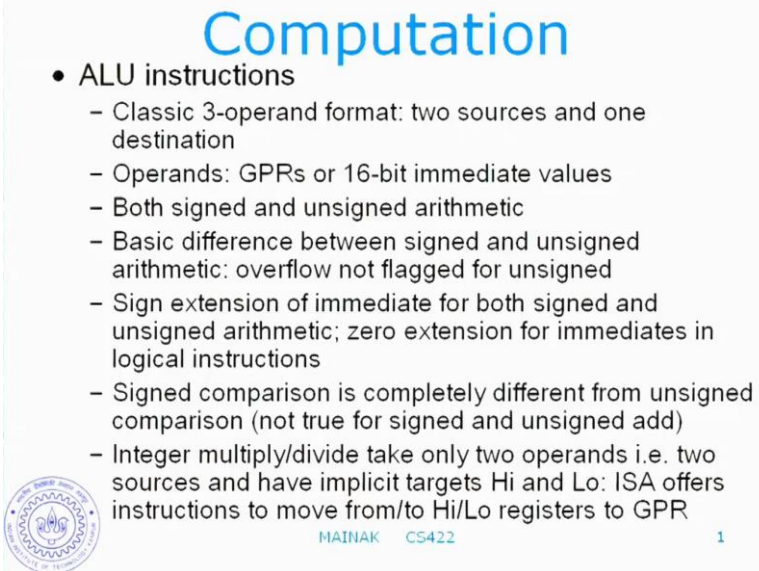


1

Floating points – single and double precision; these are the registers that MIPS had. Just to remind you quickly, it has thirty 32-bit integer registers; one of which is hardwired 0. It has 32-bit floating point registers; two of which are paired to get double precision when you need. Although that you have program counter, which is incremented by 4 on

sequential instructions – meaning that your instructions are 32 bits in size. There are two special registers: hi and lo for storing multiply/divide results. And, this is how you pair up the floating point registers.

(Refer Slide Time: 01:27)



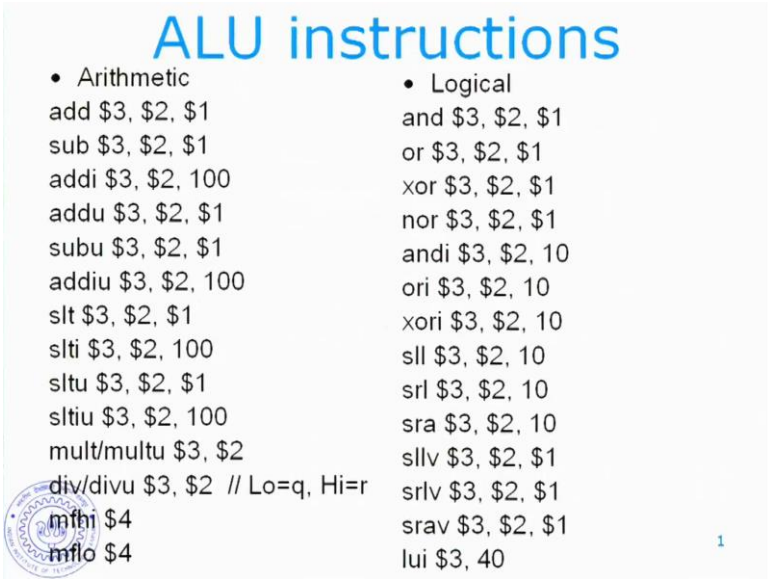
## Computation

- 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

And, look at ALU instructions, which have classic 3-operand format; that is, two sources and one destination.

(Refer Slide Time: 01:38)



## 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
mfm \$4	srav \$3, \$2, \$1
mflr \$4	lui \$3, 40

1

And, look at several other examples. This is pretty much what it is – you can go and look at the C files on it posted; that actually lists what exactly these instructions does. So, if

you have any confusion... It is written in C not in English. So, there should not be any ambiguity that, oh! what does this sentence mean exactly. There should not be anything like that. It is a C statement; you should be able to exactly figure out what it is doing.

I just want to highlight a couple of things. Especially, I have talked about this last time also. So, if you look at these two instructions: add and add unsigned; and, if you look at what they do; they do exactly the same thing; there is no difference actually. The only thing is that, we ignore the overflow in one case, which is... Does anyone remember?

Student: Unsigned case

The unsigned case; in unsigned case, we ignore the overflow. In the other case, we actually take the overflow into account; and, possibly take and accept it whenever if your overflow exceptions are set. That is one thing. The second thing is that, when you look at the immediate instructions like add immediate; where, you know that, one of the operands is an immediate value; like here it is 100. So, in this case, in the immediate instructions, so, there are again two values: add immediate and add immediate unsigned. In both cases, this particular immediate value will be signed extended. So, essentially what it means is that, again the only difference between add i and add i u is that, in the unsigned case, we ignore the overflow. But, in both cases, the immediate constant will be sign extended.

Student: Sir, in unsigned case, one is signed and one is unsigned; is it?

No, you can specify the immediate number here in add unsigned; yes, that is fine. It will be sign extended. So, only... So, that... This is somewhat confusing; but, keep this in mind; this is the only difference is that, in one case, we ignore the overflow; in the other case, we do not. It has nothing to do with the sign extension actually. In both cases, we will be doing a sign extension. So, you can set less than immediate, is essentially a comparison; it is checking whether 2 is less than 100 or not. So, in this case, 100 will be treated as a sign number. So, if you specify a negative number here, it will be taken as a negative number. But, if you specify a negative number here, you will be sign extended; you will get a 32-bit operand. But then, the whole 32-bit thing will be treated as an unsigned large number. So, keep that in mind.

So, again... So, why are we doing a sign extension, zero extension? Because of course, this is a 32-bit operand. To be able to do the operation, I need to extend this to a 32-bit operand. But, usually, this is going to be less than 32, because this is going to be part of the instruction, which is itself 32 bits. So, clearly, this is going to be smaller than 32-bits. And, mult and mult unsigned; so, again the same difference. So, keep this in mind – arithmetic instructions on unsigned and signed difference is that, you ignore the overflowing ((Refer Slide Time: 04:59)) So, here the result goes to two specific registers: lo and hi. In case of division, quotient goes to lo, remainder goes to hi. And then, you have two instructions to move from hi and to move from lo, so that... For example, this instruction will copy hi into dollar 4. So, now, you can actually do conventional arithmetic on the result of the division or multiplication, because lo and hi do not... You are not allowed to use ((Refer Slide Time: 05:33)) highly ready other arithmetic instruction as an operand. So, clearly, you have to bring it first to the general purpose register; then, only you can use them. Any question on this left side – the left column – the arithmetic column? So, this is pretty much it actually. So, I might have missed out a few – one or two here and there; you can look up the C file; you will have the exhaustive list of all arithmetic instructions.

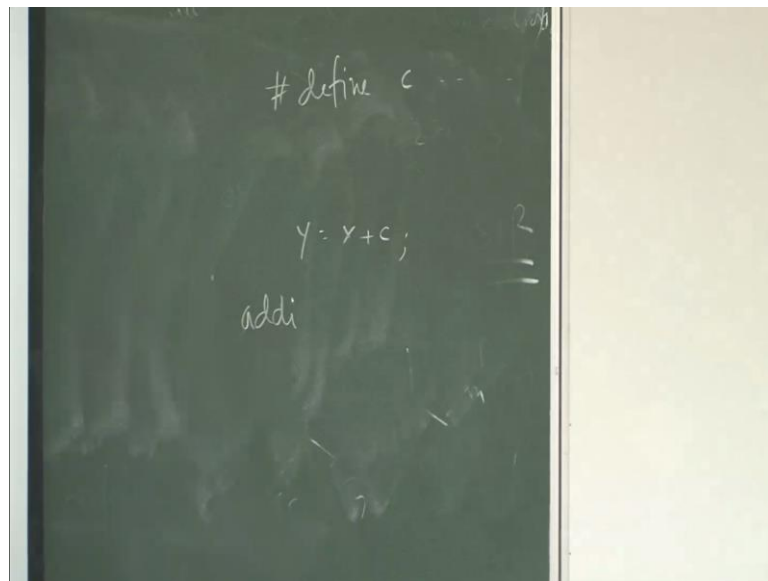
So, the logical side – there are really is not much to explain; that, the mnemonics are self explanatory – AND, OR, XOR, NOR and immediate. Again here you will be doing 0 extension in all logical operation. So, these are not sign extended; these are actually 0 extended. OR immediate, XOR immediate, shift left logical, shift right logical, shift right arithmetic – these all we discussed last time. Essentially this means that, you shift dollar 2 to the right by 10 positions while shifting in the sign bit, instead of shifting in zeroes. Shift left logical variable – this one has a special property that, whatever you mentioned in dollar 1... So, dollar 1 is a shift amount – by how much I could shift. It will only take the lower 5 bits, because a maximum shift amount cannot be more than – cannot be more than?

Student: 32-bit.

Cannot be more than 32 bit; exactly. So, it will pick up only the lower 5 bits in dollar 1 and ignore everything else. You can specify a very large number in dollar 1; it is going to ignore everything, but the last 5 bits till ((Refer Slide Time: 07:13)) 5 bits. Same for shift right logical variable, shift right arithmetic variable. lui – this one is used to only affect...

Actually, it affects the whole register; but, what it does is that, it puts 40 in the upper 16 bits and 0 about the lower 16 bits in dollar 3. So, essentially, this one stands for load upper immediate. So, you see the practical application of this. Can anybody guess how I might want to use lui; what could be a good application? Any guess about that? How would I load a constant into register. So, I want to operate on a constant. What are the options do I have? Suppose I want to do the operation  $x \text{ plus } c$ ; were,  $c$  is a constant.  $c$  is a constant known at compiled-time. So, essentially, I have this kind of a program.

(Refer Slide Time: 08:22)

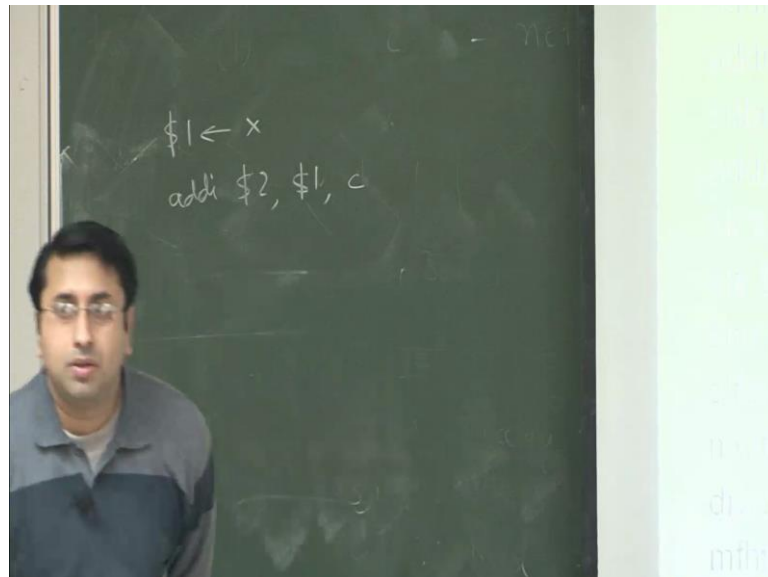


Hash define  $c$  is some value here. And then, somewhere, the program I say  $y$  equal to  $x$  plus  $c$ . So, compiler knows the value of  $c$ . So, what are the options do I have? How do I compile this instruction? Yes? How do I compile this instruction?

Student: ((Refer Slide Time: 08:53))

Yes. So, that is one option. Why do not we add immediate? That is the obvious one to do. Add immediate. So, by the way, remember that, in most cases, the compiler will generate an add immediate as a mostly ((Refer Slide Time: 09:08)) unsigned, because it will usually generate the instruction; where, over flow is actually done down. Overflow instruction is done down. So, the obvious answer is that, why not add immediate. Does anybody see any drawback of using add immediate? So, essentially, what I will do is I will put x in a register. So, let us be explicit on that.

((Refer Slide Time: 09:33))



So, I ((Refer Slide Time: 09:31)) say r 1 gets x. And, let suppose that, y is allocated in r 2; then, I will generate add i – following the loops; no equation. Some – that the value of c. Does anybody see the problem in this? Can I do this all the time?

Student: The value of c should be...

Should be?

Student: Bringing less in this way

Should be what?

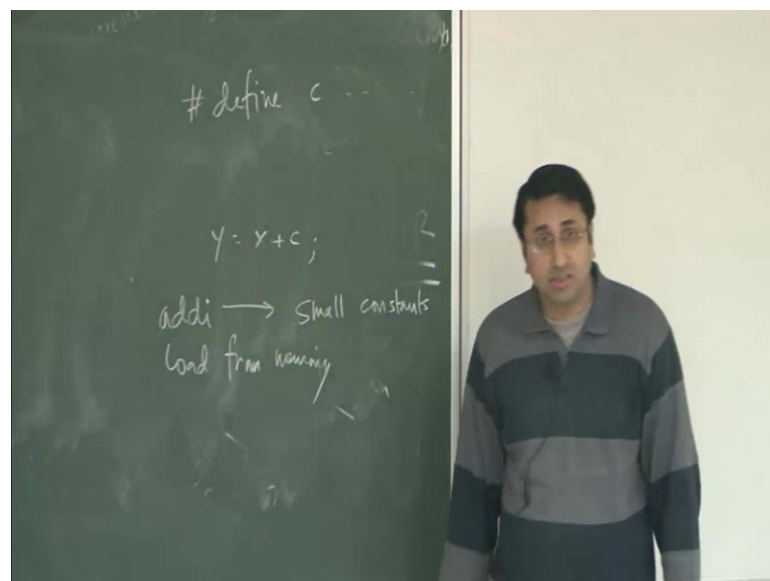
Student: Should be less in this way

Should be less in this way?

Student: Because the instruction is of 32-bit and the ((Refer Slide Time: 10:17)) and the other operand will take some... There will be limited...

Exactly. So, when the designers came up with MIPS encoding, they only have allocated some fixed number of bits for encoding an immediate value. So,  $c$  should be within that range. Like for example, if I say when my immediate value is encoded in 16 bits; then,  $c$  better be not more than totally 16 minus 1. That is the largest value again ((Refer Slide Time: 10:50)) So, clearly, there is an immediate. What do I do if  $c$  is larger than that? What options you will have? How do I compile this? Does everybody follow what I am trying to say? Why addi has a problem when  $c$  is large? But, I cannot of course impose a constant on a high level language program or single code; you are profiling for architecture; for which constant can be only this much. That is a non sense. So, there are few questions compiling for large constants; how do I do that?

((Refer Slide Time: 11:25))



So, what other options do I have? So, addi for small constants. What else?

Student: ((Refer Slide Time: 11:35))

And?



Student: ((Refer Slide Time: 11:39))

Keeping allocated throughout? No, what you mean by that? Assign critical registers means what?

Student: ((Refer Slide Time: 11:50))

How does it get that value? How do you bring that value into that register? Rohit?

Student: ((Refer Slide Time: 11:59))

You load it. And so, whenever you need, you bring it from there. So, essentially what you are suggesting is that, allocate c in memory; whenever you need the value from memory to the load instruction from memory to the register. Is that what you suggested?

Student: No

No. But, anyway that is an option; let me gets me illustrate here – load from memory. What you have to say?

Student: Keep it ((Refer Slide Time: 12:30))

How does the register get the value? That is the question.

Student: One time load and then use that register...

One time load. So, this is what the register is suggesting; you load from memory what you are saying is that, I would pin that register forever. So, maybe ((Refer Slide Time: 12:49)) compiler; of course that, it is not a very good idea all the time, because essentially what you are doing is this constant may be used in some localized places in the program. But, you are peeling it for the entire life of the program; which constraints the compiler in allocating registers. What else can I do? Load from memory is a costly operation actually; memory is slow. So, this is often called interpreting a constant at run time – this particular operation. Even if you know it at compiled time, you are actually generating the constant from the memory; which is a very very very big bad solution. What else can I do?

Student: Can we use an auto increment three times?

There is no auto increment instruction here in this list. Suppose you can use two instructions, instead of one while generating its constant; but, no memory operation. Can you do that?

Student: ((Refer Slide Time: 13:57))

How? How they do that?

Student: First of all, note that is 16 bits...

Which 16 bits?

Student: From this constant c.

Which 16 bits?

Student: Upper 16 bits

Upper 16 bits of c; which the compiler can calculate at compiled time.

Student: Again that shift...

Shift?

Student: Shift that 16 times and then again load the...

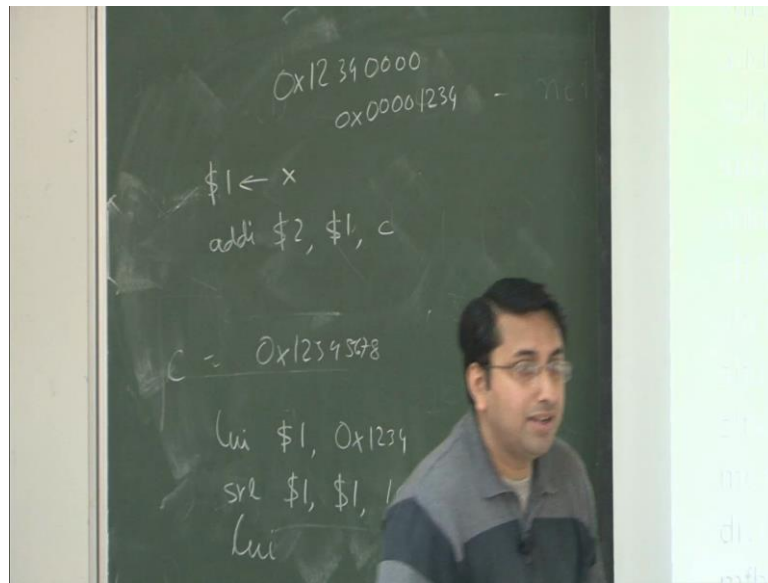
No, what do you shift by 16 times?

Student: That the value of c; it will be some 32 bits.

Yes.

Student: So, first of all would sixteen bits...

(Refer Slide Time: 14:35)



So, let us take an example. Suppose  $c$  is the  $0x12345678 - 5678$ . It is 32-bit constant. Then?

Student: So, first of all, would 16 – upper 16...

What is that? 1 2 3 4. So, I will say `lui`. Let us say dollar 1 –  $0x1234$ .

Student: And then, dollar 2

Dollar 1 – keep dollar 1

Student: And, the shift it in 16 bits...

Shift who? Shift what?

Student: Shift  $c$

Shift  $c$ ? See here what you are trying to do? You are trying to get the lower 16 bits?

Student: Yeah

And then, do what? Do what?

Student: We can get the lower sixteen bits...

I want dollar 1 to contain  $c$ .

Student: You have to firstly get the lower bit and then shift right and then to the...

Can you give the instructions equals as early?

Student: Sir, here we can get 0 x 5 6 7 8 and...

When?

Student: Shift right...

No-no; wait-wait; what is the instruction? You have to pick one from this list. Be very very very specific.

Student: s r l

s l l?

Student: Sir, r l

s r l? What would you shifting? What is the shift amount?

Student: 16 ((Refer Slide Time: 16:08))

What is dollar 2?

Student: Dollar 2 ((Refer Slide Time: 16:11))

Dollar 2 is?

Student: Is dollar 1.

So, after lui?

Student: Yes sir.

Okay, I will put an s l l. Let us see what you are saying. s l l...

Student: s r l

s r l; s r l?

Student: And, dollar 1

Dollar 1...

Student: ((Refer Slide Time: 16:31)) Sir, here dollar 1 and...

Dollar 1, dollar 1, 16? That will have a...

Student: ((Refer Slide Time: 16:41))

4

Student: ((Refer Slide Time: 16:49)) ...again will load the value in the most ((Refer Slide Time: 17:00))

Wait, wait, wait. Is that all?

Student: Sir, now, the upper bits...

Now, what is the quantity of dollar 1 ((Refer Slide Time: 17:06)) to say after this?

Student: Double 0 double 0 1 2 3 4

Double 0 double 0 1 2 3 4? Let us go one by one. What is the quantity of dollar 1 after this instruction?

Student: One is equal to ((Refer Slide Time: 17:21))

1 2 3 4 and garbage? No.

Student: 0 0 0 0

1 2 3 4 0 0 0 0. This is dollar 1. Now, I shift it by 16 bits. What do I get?

Student: We get a 0 0 0 0 ((Refer Slide Time: 17:42))

That is dollar 1 gives. Then, what you are saying?

Student: Then, l u i ((Refer Slide Time: 17:49))

Again, l u i?

Student: ((Refer Slide Time: 17:56))

I want this value; remember that.

Student: First of all, load c into some register.

How?

Student: ((Refer Slide Time: 18:12))

That is what we are trying to do actually.

Student: Whole 32 bit is from memory ((Refer Slide Time: 18:16)) Once we have to use...

No, why do you want to...

Student: ((Refer Slide Time: 18:20))

Why? For what purpose? Compiler knows the constant. So, it can extract any arbitrary bit from c at compiled time. It can do that. I only want to generate the code to make sure that, if I want c to be in a register, I have that, so that I can compile this particular statement – y equal to x plus c.

Student: Sir, we can load rest of the bits in other register – any other register?

Rest of the bits?

Student: In any other register.

How do I do that?

Student: lui dollar 2 and rest of the values.

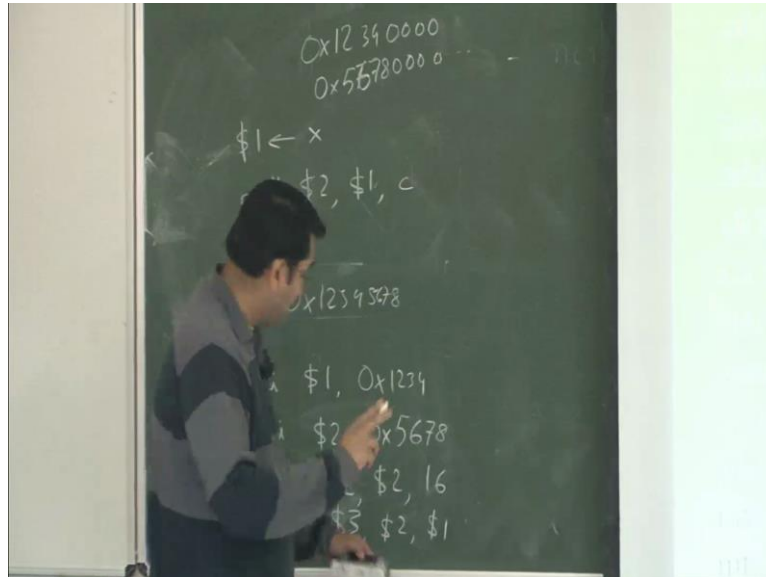
lui dollar 1 0 x 1 2 3 4 will give you this particular content.

Student: Will it do a shift?

Shift what? How do I get 5 6 7 8? ((Refer Slide Time: 19:17))

Student: We do two shifts; first, less than ((Refer Slide Time: 19:20)) You load 5 6 7 8 in other register – different register l u i dollar 2 will ((Refer Slide Time: 19:27)) 5 6 7 8.

((Refer Slide Time: 19:31))



That will give you 5 6 7 8.

Student: Will you shift this 5 6 7 8 ((Refer Slide Time: 19:36))

Okay, and then?

Student: And then, you do ((Refer Slide Time: 19:38))

Then, I do a?

Student: ((Refer Slide Time: 19:40))

Can I... Does everybody follow what he is saying? So, he is saying... Let me write it down. We will then optimize. Is that what you are suggesting? Can we optimize it?

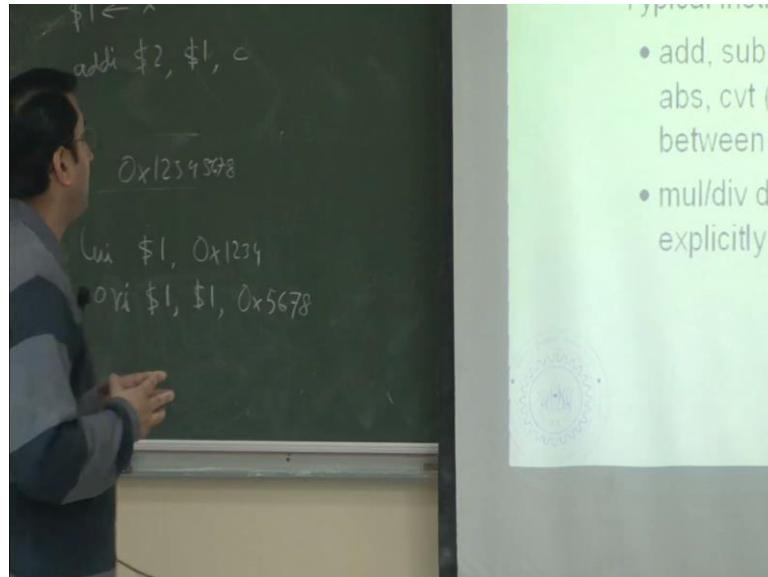
Student: Sir, we do a lui ((Refer Slide Time: 20:29)) instruction and then do a add 5 6 7 8; add dollar 1 – add dollar 1 dollar 1 5 6 7 8.

Add immediate you mean?

Student: Yes sir.

Like? I can do that. So, I can remove these.

(Refer Slide Time: 20:54)



Student: Dollar 1 dollar 1 ((Refer Slide Time: 21:01))

Right? That will give you the same outcome? So, what else can I do? Can I use anything else as an add i from this list?

Student: This is not the same...

This will not give the same thing? Why?

Student: Because this is 1 2 3 4 and 5 6 7 8 at the lower bits.

So, dollar 1 will have this one at the end of lui; and, 5 6 7 8 will get added here. Can I use anything else as an add immediate from this list? We are almost there.

Student: ((Refer Slide Time: 21:50))

Or, r. So, this is what a good compiler will generate. It would avoid ((Refer Slide Time: 22:02)) the adder because addition is usually not a single cycle operation. They do cycles; but, all is ((Refer Slide Time: 22:09)) You can do it in the cycle. So, these are...

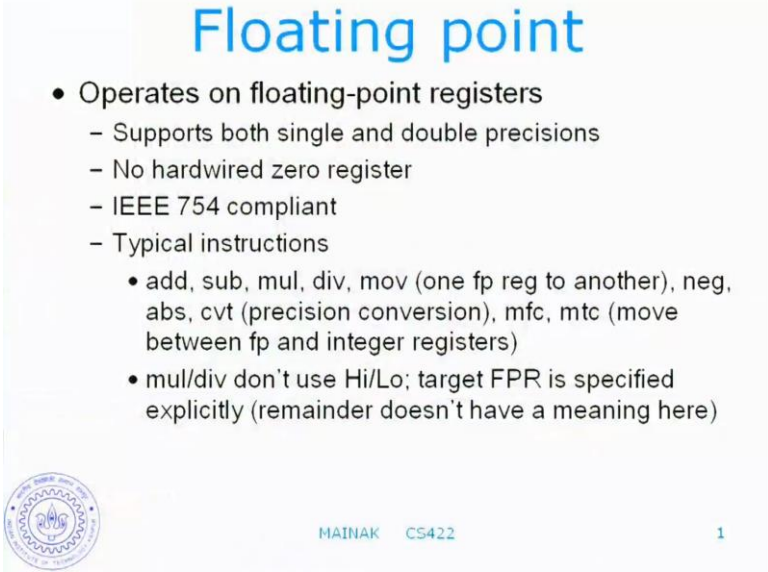


This is the biggest use of lui – loading a large constant in an instruction in two cycles; two cycles or... This will be two cycles. Is it clear to everybody? Do we...

Student: The cycles that we did in lui could be more than the other instructions or...

No, this constant will go into the immediate field to same 16 bits.

(Refer Slide Time: 23:00)



## Floating point

- Operates on floating-point registers
  - Supports both single and double precisions
  - No hardwired zero register
  - IEEE 754 compliant
  - Typical instructions
    - add, sub, mul, div, mov (one fp reg to another), neg, abs, cvt (precision conversion), mfc, mtc (move between fp and integer registers)
    - mul/div don't use Hi/Lo; target FPR is specified explicitly (remainder doesn't have a meaning here)

MAINAK CS422

1

Floating point operations – these operate on floating point registers. Suppose both – single and double precisions; there is no hardwired 0 register. So, you have to interpret 0 for that. IEEE 754 compliant. Typical instructions add, sub, mul, div, mov; where, essentially move from one floating point register to another; negate, absolute value, cvt is a precision conversion. So, if you want to convert to single precision, double precision; essentially, this is used when we are trying to a type casting in a high level language. For example, you are type casting a float to double; compile would generate a cvt instruction. or vice-versa. Here we have seen mtc – move between floating point integer register. So, this c might be a little ((Refer Slide Time: 23:32)) y at the same move from c and move to c. The reason is that... Can anybody guess what does the c stand for? Because I would have actually expected that, it would say move from f and move to f – mfa or mta.

What is a c? What is it referring to?

Student: Conversion

No, it is not conversion; it is moving from something; that is what the c stands for.

Student: Cache?

Not cache. What can you think off? Any other word related to microprocessors – starts with c? ((Refer Slide Time: 24:14)) But, the same thing...

Student: CPU

CPU? Actually, close;

Student: Compiler coprocessor

Coprocessor; thank you very much. So, the floating point unit actually was a coprocessor in the early MIPS processor. ((Refer Slide Time: 24:29)) as well. It used to come with a separate coprocessor. So, that is exactly what is referred to move from coprocessor, move to coprocessor. So, the first instruction – mfc moves from a floating point register to an integer register. And, mtc moves from an integer register to a floating point register. And, this is again generated whenever you try to do type casting in the high level language. Suppose your cast and caster has initial value to float; the compiler would generate one such instruction. Mul and div do not use hi-lo registers, because in this case, the targets are actually floating point registers. So, a floating point mul instruction actually will have a specified target – some floating point registers. The remainder does not have a meaning here. So, just have a result. So, only divide a floating point number by other number; you just get a floating point number; there is nothing like quotient or a remainder. Any question?


Student: So, how do we address this floating point registers?

So, you have names. You can name floating point registers.

(Refer Slide Time: 25:45)

## 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<sub>2n</sub> and \$f<sub>2n+1</sub> are accessed by name \$f<sub>2n</sub> e.g. \$f2 specifies the 64 bits in \$f2 and \$f3 with the least significant word in \$f2


1

We have this floating point 04 percent; you can name them.

(Refer Slide Time: 25:48)

## Load/store

- One address mode for memory
  - Displacement only (always sign-extended)
  - Most loads/stores are aligned (except lwl/lwr, swl/swr)
  - Loads are supported for signed and unsigned data; unsigned loads zero-extend the loaded value
  - Supports three sizes: byte, half word, word (double word is supported in 64-bit ISA)
  - Byte load: lb \$3, 4(\$13) or lbu \$3, 27(\$20)
  - Half word load: lh \$3, 12(\$10) or lhu \$6, 0(\$7)
  - Word load: lw \$20, 52(\$29) [no unsigned flavor]
  - Byte store: sb \$3, 5(\$2)
  - Half word store: sh \$3, 56(\$29)
  - Word store: sw \$2, 20(\$2)
- The immediate is *not* shifted by load/store size; it is just sign-extended and added to the base register content
- In addition, floating-point load/store: lf \$f1, 60(\$22)



So, that takes care of your integer arithmetic, integer logic – floating point arithmetic.

Now, we come to the memory side of it. So, load/store operations. There is only one addressing mode supported. So, we looked at 10 in the past. So, MIPS support exactly 1.

And, that is the displacement addressing mode; always sign extended. So, essentially, you have a... So, it looks like this for example. So, we have a base register and you have a displacement. The displacement is always sign extended. So, most load stores are

aligned, except we talk about this load word left, load word right; and then, lwl, lwr. Similarly, store word left and store word right. So, we talk about those. So, these are the only exceptions, which are non-aligned loads and stores. And, those four instructions also allow you to actually access sub registers. It is some part of a register, while keeping the rest on other. Loads are supported for signed and unsigned data. Unsigned loads zero-extend the loaded value. So, keep in mind that, the signed and unsigned loads are loads differ in this volume; that is, once you reach the data from memory, whether you sign-extend the data or zero-extend the data.

Displacement is always sign-extended. Supports three sizes byte, half word and word. Double word is supported in 64-bit ISA. So, 32-bit ISA does not have any double word instructions. So, byte load – there are two as we have already mentioned. One is load byte signed – lb; do not write the sign s here explicitly. And, load byte unsigned. So, this is what it looks like. It takes essentially one source register, one displacement constant and target register. So, displacement in both cases will be sign extended ((Refer Slide Time: 27:46)) – meaning that, if you specify a negative displacement, it will be treated as a negative displacement; it will be subtracted from this actually. And, the final value that you load will be sign-extend in this case to fill up a 32-bit register. Whereas, this one is zero-extended.

Half word load – lh and lhu – load half word and load half word unsigned. Word load – load word; there is no unsigned flavor. Why is that?

Student: 32...

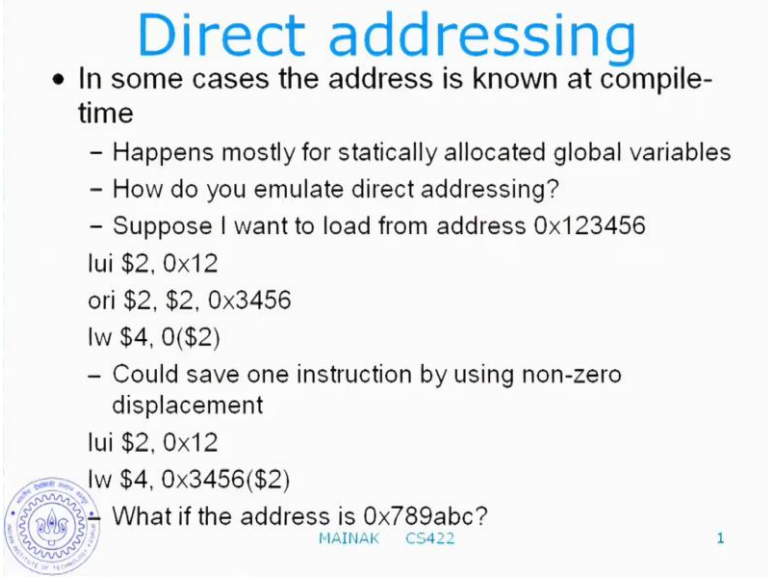
Exactly. So, it already returns a 32 bit value. There is nothing to extend actually. So, even if we had a load word unsigned, do ((Refer Slide Time: 28:26)) It can be exactly same. Similarly, we have a byte store. So, there is no unsigned version for store. We want to store a byte; so, what we do is you take two source registers. Dollar 2 will be used as a base register for address; which will be added to the displacement. Displacement would be sign-extended. And, you take the value for dollar 3; you will take the least significant byte of dollar 3 and send it to this particular address. Similarly, you have half word store – sh and word store – sw. So, sw would store the entire register dollar 2 at this particular address. So, you can exactly figure out how these instruction

have to execute in the c file. You can go and look out actually – quick byte extracts and all these things.

The immediate is not shifted by load/store size. So, that is very important to understand, because in many books, you will find that, not with respect to MIPS; but, in general, they say that, this number – 27 will actually be multiplied by something before adding to dollar 20. MIPS does not do that. It will just take this displacement as it is; will add it to this base; generate the address. And, whatever that address will be used to load and store. That is what is mentioned here. The immediate displacement will not be shifted by load/store size. It is just sign extended and added to the base register content.

In addition, you have floating point load store. For example, you have lf – dollar f 1 60 dollar 22. So, these instructions are being notorious in the sense that, it will actually have an integer register source for generating the address and a floating point register target. So, you can imagine a store floating point instruction will actually have a floating point register source and an integer register source. So, these actually interface with the both integer pipeline and the floating point pipeline. So, that way they are little harder to implement it. But, otherwise, they execute exactly in the same way. This is the sign-extended added to dollar 22; get the address; load the value to dollar f1. And, here of course, you do not have these unsigned byte, half word, word – all these things. You have only two: one is lf, that is, single precision load; one is lf.d – double precision load. Depending on that, it will load either 32 bytes from here or 64 bytes from here. It is 32 bits and 64 bits compiles here. Any question on load/store?

(Refer Slide Time: 31:02)



## Direct addressing

- In some cases the address is known at compile-time
  - Happens mostly for statically allocated global variables
  - How do you emulate direct addressing?
  - Suppose I want to load from address 0x123456

```
lui $2, 0x12
ori $2, $2, 0x3456
lw $4, 0($2)
```

- Could save one instruction by using non-zero displacement

```
lui $2, 0x12
lw $4, 0x3456($2)
```

- What if the address is 0x789abc?

1

So, in some cases, the address is known at compile-time; happens mostly for statically allocated global variables. So, compiler knows the exact add ((Refer Slide Time: 31:12)). So, naturally, the question is why should I have to interpret the address, that is, generate the address at run time. Why cannot ((Refer Slide Time: 31:22)). So, how do you emulate direct addressing? So, suppose I want to load from address 0x123456. So, this is exactly where lui comes very very ((Refer Slide Time: 31:32)) So, what you do is you first load the upper 16 bits, that is, 0x12. And then, you or with the load 16 bits; and, what you have essentially dollar 2 is the address. So, you can shift 0 dollar 2 ((Refer Slide Time: 31:47))

Could save one instruction by using non-zero displacement. I could be also this. That is also... It is going to be better, because it saves one instruction. Clear everybody? What if the address is 0x789abc? What is the special about that address? So, let essentially would the question is asking you is that, can I use the same thing for this address? The answer must be no, why is that? So, what I am saying is that, I cannot really say lui dollar 2 0x78 and load word dollar 4 0x9abc dollar 2. I cannot do that. Why?

Student: Displacement is getting the imitation from maximum displacement.

This is 16 bits maximum displacement 9abc – 16 bits. The displacement is ((Refer Slide Time: 32:51)) I have not yet mentioned that. We will come to that – exact encoding base. So, MIPS is a 16 bit immediate value in all cases. So, any constant that was not into the

instruction would cannot exceed 16 bits. So, this is okay – 9abc – 16 bits. But, this is going to be wrong. Why is that? What is the problem? Yes?

Student: 9 abc would be sign extended

Okay, thank you. So, somebody is paying attention; that is pretty clear. So, yes, 9abc... What is the MSD of 9abc – the most significant bit?

Student: 1

1. So, remember that, displacement is going to be sign extended. So, if sign extend 9abc; then, fill up with all 1's. So, the address that, we are going to get will not be this actually; it is something else. You add up two things. So, you cannot do this. What should we do? What should I do here then? How do I get this done? Can I use this? No? Yes. Is that all I find? 9abc here? Why?

Student: ((Refer Slide Time: 34:08))

9abc...

Student: 8 value.

And, it has to be extended to get a 32-bit operator.

Student: Sir, it is not signed.

What is not signed?

Student: Immediate value ((Refer Slide Time: 34:23))

How do you figure out whether signed or not?

Student: ((Refer Slide Time: 34:25))

I do not have any problem with this one here. Now, the problem is here – shifts to here. We have an immediate value, which is 9abc here. But, it is okay; why?

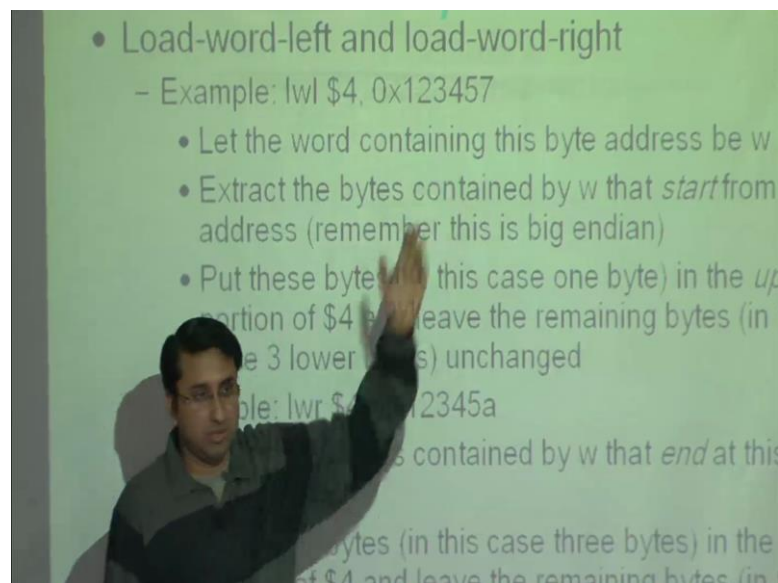
Student: Zero-extended

Who is zero-extended? Why? Why not sign-extended?

Student: Because it is logical...

Logical operation; thank you. So, logical operation often give immediate value that is always zero-extended. So, this scheme is okay. So, compiler has to figure out by looking at this address, which one can I pick? This one or this one? So, in most cases, it will be able to pick this one. But, in some cases, it will be able to pick this one depending on the address. So, remember that, it cannot really figure out by looking at how big the address is, because 78 – 7abc is okay actually. So, you cannot just make a comparison by looking at the address range; you have to look at only these 16 bits and ask is the MSD 1 or 0? If it is 1, it will generate this code; otherwise, it will generate this code. Question? So, keep this in mind. Arithmetic immediates are sign-extended; logical immediates are zero-extended.

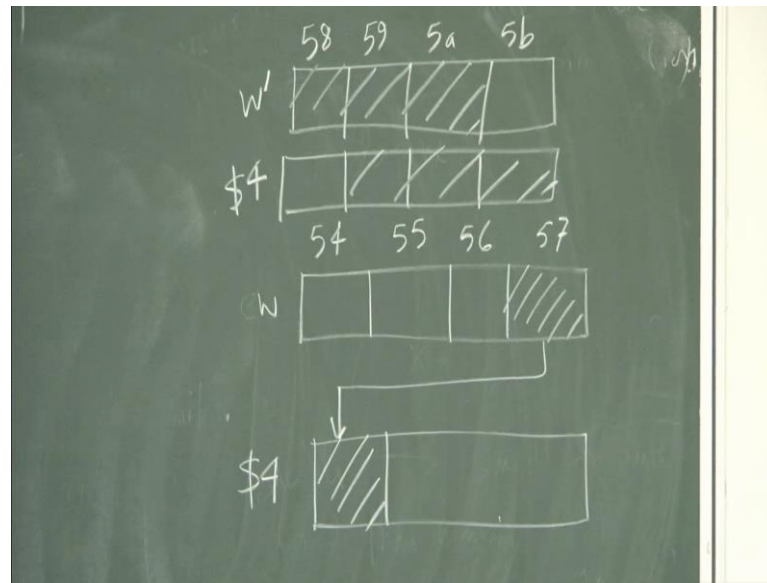
(Refer Slide Time: 35:59)



Now, load-word-left and load-word-right. So, let us try to understand what we do. Let us take this particular example. Of course, this is not negative at MIPS, because you cannot really mention the address like this. You have to generate the address in some other bits, so that it repeats the displacement more. But, here it is just an example. So, I am trying to do something from an address, which is 1234567. And, it looks like a load operation; and, I want some value in the dollar 4 at the end of this. So, let the word containing this byte address be *w*. So, what does that mean?



(Refer Slide Time: 36:36)



So, 123457. So, let us see what is the word. You have 4 bytes and 57 56 55 54 fine. So, we can give... So, you have 54 here, 55 here, 56 here, 57 here. Extract the bytes contained by `w` that starts from this address. So, it starts here. So, in this case, it is going to be only 1 byte, because I will say extract the bytes contained by `w` that starts from this address. Put these bytes; in this case, just 1 byte – this one in the upper portion of dollar 4 and leave the remaining bytes unchanged. So, essentially at the end of this particular instruction, the dollar 4 will have this byte copied here; remaining 3 bytes remaining unchanged. This is load-word-left – load-word-left of dollar 4. Starting from that address, take the bytes load-word throughout the left end of dollar 4; load those bytes throughout the left end of dollar 4. Is it clear to everybody what he is doing? So, this is the only instruction I am...

We have `lwr` also; I will talk about that. So, these are the only instructions in MIPS that allows you to do unaligned word accesses – unaligned word access, because the aligned word address would have given this address. That allows you to modify a portion of a register. This is the only instruction, because everything else would actually overwrite dollar 4 completely. It preserves the lower 3 bytes. So, `lwr` does exactly the same thing just in opposite direction. So, let us take this example; it says 12345a. So, let us see what is `w` in this case. 12345a. We have 5958 – 58; 58 is on the big end; 59 5a 5b. These are word containing this byte address; extract the bytes contained by `w` that end at this address; that end at this address; so, looking at these bytes – three bytes.

Put these bytes – in this case, three bytes – in the lower portion of dollar 4; and, let the remaining bytes; in this case, the upper byte unchanged. So, at the end of the execution, dollar 4 will have these three bytes copied on this side leaving this one unchanged. Why are these instructions at all important? Under what circumstances you generate these instructions? Can anybody think of a use case? Think about using both of these together. Yes? What if I teach these two words as w? Let w remain different. w and w prime.

Student: Circular right

Circular right?

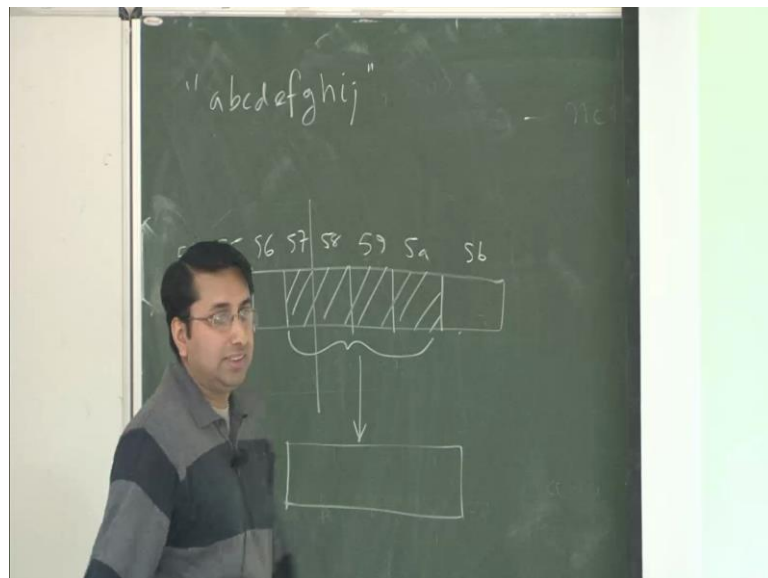
Student: Circular right ((Refer Slide Time: 40:42))

Circular right? What is that?

Student: ((Refer Slide Time: 40:46))

So, let us teach w and w prime.

((Refer Slide Time: 40:55))



So, we have 54 55 56 57 58 59 5a 5b. If I use these two instructions one after another, I would have copied ((Refer Slide Time: 41:32)) So, I would have copied this portion into dollar 4. Is that clear everybody at least? If I use these two instructions one after another, I would have copied those four bytes into dollar 4. Why do I have to do in this way? Is

there any other way of doing it? Once you realize that the answer is no, you then immediately see the use of this instruction.

Student: Variable...

It is not variable like... I am just copying a word actually. I am copying a word standing 57 58 59 5a into other instruction; that is it. Why cannot I use load word in this case? It is not aligned. Yes, exactly. So, it is very useful why you want to copy a word from an unaligned address. And, these are only way you are going to do it in MIPS; and, there is no way actually. And, what kind of programs might generate unaligned word accesses? If I ask you to write a c program that compiles into lwl and lwi instructions; can you do that?

Student: Use union.

Use what? Union? Something similar; yes, maybe yeah; you should be able to do by unions. I will give you all... Suppose I give you a string; I will give you a string, which is something – a b c d e f g h i j – something, etcetera. And, I want to... What?

Student: ((Refer Slide Time: 43:17))

Say it?

Student: ((Refer Slide Time: 43:19))

I want to extract some characters from some arbitrary places. The most efficient code will be these two instructions.

Student: Sir, byte will be always a ((Refer Slide Time: 43:30))

Byte will be?

Student: Sir, byte will be 1 by 10

1 by 10; exactly.

Student: So, every access will be aligned.

Every access will be aligned; exactly. So, one way to do that would be to read one byte at a time; yes. But, that will take four instructions to reach 4 bytes. I can use just two to load 4 bytes like this. Yes, of course, you can synthesize it using bytes – one byte at a time. But, this is going to be much more efficient. So, I will actually... Today, I will post on the course of one C program; and of course, it is MIPS ((Refer Slide Time: 44:05)) in C actually – simple C program; which tries to do something on this. Takes a string and tries to extract some arbitrary characters. You will see that, these two instructions are again generated.<sup>4</sup>

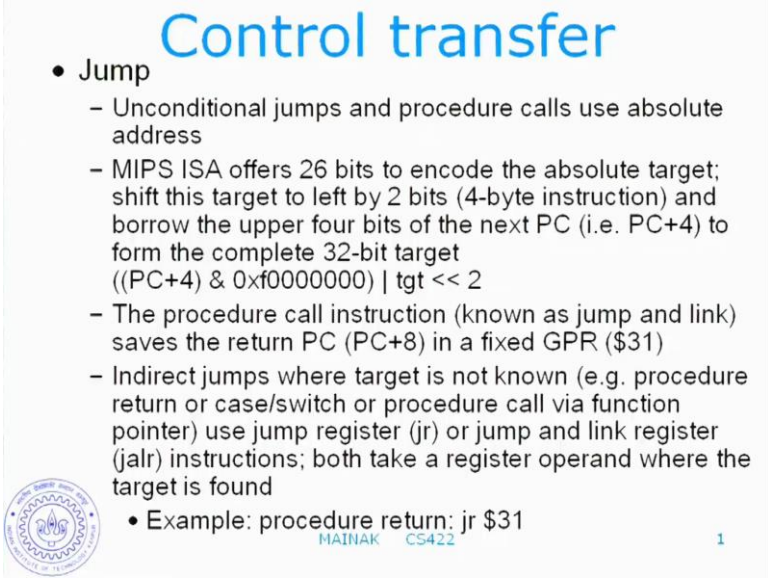
Student: Sir, what about unaligned 16 bit accesses – unaligned half-word?

No, you cannot do that.

Student: We cannot do...

No, we have to load a word and then shift; that is only way we doing it. Like for example, if you want to load half-word 57 58, you are able to do that. You load this one and then do a shift together. Any question?

(Refer Slide Time: 44:48)



## Control transfer

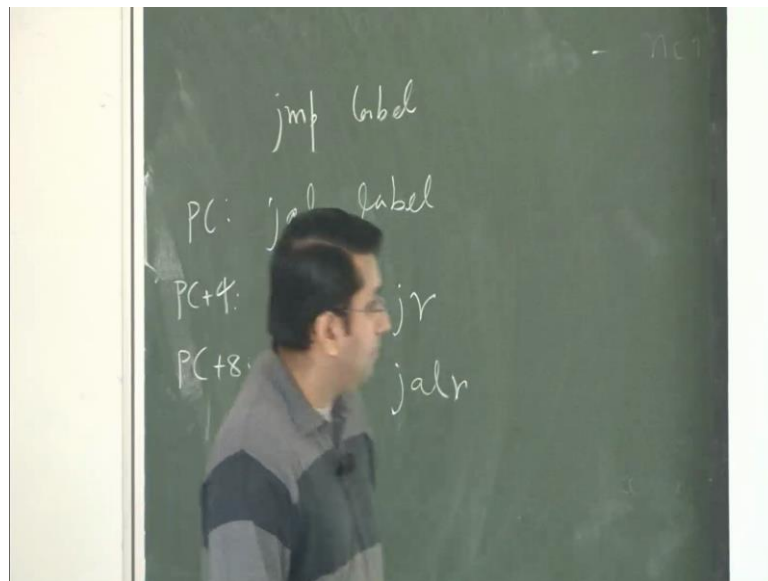
- Jump
  - Unconditional jumps and procedure calls use absolute address
  - MIPS ISA offers 26 bits to encode the absolute target; shift this target to left by 2 bits (4-byte instruction) and borrow the upper four bits of the next PC (i.e. PC+4) to form the complete 32-bit target  
 $((PC+4) \& 0xf0000000) | tgt \ll 2$
  - The procedure call instruction (known as jump and link) saves the return PC (PC+8) in a fixed GPR (\$31)
  - Indirect jumps where target is not known (e.g. procedure return or case/switch or procedure call via function pointer) use jump register (jr) or jump and link register (jalr) instructions; both take a register operand where the target is found
    - Example: procedure return: jr \$31

MAINAK CS422 1

Control Transfer – So, there are jump instructions. So, these are essentially unconditional jumps and procedure calls. They use absolute address, because the compiler knows that ((Refer Slide Time: 45:00)). So, hereby procedure call; I will direct procedure calls for immediate calls. So, MIPS ISA offers 26 bits to encode the absolute target. So, shift this

target to left by 2 bits because the instructions are four byte instructions. So, any legitimate PC will have last two bits 0. So, essentially, when you are specifying a bunch target, there is a meaning of including those two bits also, because they are implicitly 0. So, what you do is – you specify 26 bits, which can inner 28-bit target by shifting into zeroes. And, borrow the upper 4 bits of the next PC; that is, PC plus 4 to form the complete 32 bit target. So, essentially, what I am saying is that, suppose at a particular PC, you have jump instructions.

(Refer Slide Time: 45:51)



It says jump is on label. And, this label is known at compile time; that is what we are talking about here – unconditional little jumps. What the compiler will do is – it will take the label; shift out the lower 2 bits, because the lower 2 bits are always 0 for any legitimate program counter. And then, what you do is – it will take 26 bits – 26 bits are allowed only. It cannot be more than that; which means there is a span of this label – how far you can jump; there is a limit to that. So, this is how it computes the final target. It takes the upper 4 bits. So, 28 bits would given to be upper 4 bits to form legitimate PC. PC is 32 bits. So, it takes the upper 4 bits from the next PC – PC plus 4; and, shifts in the target in the lower 28 bits. So, the procedure call instruction known as jump and link in MIPS.

So, this is jal; that is a mnemonic used for procedure call – jump and link – saves the return PC; which is PC plus 8 ((Refer Slide Time: 47:05)) Why it is PC plus 8 and not

PC plus 4 returned in this? When you make a procedure call, you should return to the next instruction and start executing. So, it jumps one instruction down. It comes that as PC plus 8, not PC plus 4. So, ((Refer Slide Time: 47:27)) Does anybody know why?

Student: Sir, we have some ((Refer Slide Time: 47:35))

No, you are saying that, return address is PC plus 8.

Student: Stack will be downward doing...

No, it has nothing to do with the stack. I am saying I have at a jal instruction; which says jal label. And, I have a PC here. The next instruction is PC plus 4. I should be returning here. So, return address should be PC plus 4. But, what MIPS does is it saves the return address as PC plus 8.

Student: ((Refer Slide Time: 48:08)) saves the next instruction...

That is PC plus 4.

Student: And, return also – have some return address also.

This is the return address. We are not talking about the return value of the function. That will go on the stack of ((Refer Slide Time: 48:22)) We are just trying to figure out where I should resume my execution when the function completes. And, what MIPS does is – it is PC plus 8; it keeps over my instruction. PC plus 8 is here – the next one.

Student: What is the next part?

That is just a name. It is linking with a procedure. Guess?

Student: PC plus 4 is already in the pipeline.

PC plus 4 is already in the pipeline. So?

Student: Next cycle – PC plus 8...

Next cycle? No. I may take billion cycles to compute that procedure. PC plus 8 should not be in the pipeline; should not. You cannot execute PC plus 8.

Student: ((Refer Slide Time: 49:14))

Already? Those are reverse direction ((Refer Slide Time: 49:19)) I will give you the answer. I thought if somebody knows; anyway. So, it saves the return address, which is PC plus 8 in a fixed register, which is dollar 31. This is known as a jump return register. Or, I will call the link register. So, that is the procedure call instruction. So, the label is encoded exactly the same – in a same way. So, how far can you jump? 2 to the power of 26 instructions. So, that is the jump stand; cannot be more than... which is a very large actually – stand. 2 to the power of 26 is what? 64 million instructions I can jump.

Indirect jumps, where target is not known – that is, procedure return or case switch or procedure call by a function pointer uses a jump register instruction or jump and link register instruction. So, there are two instructions to do that: one is jr; another one is jal r. Both take a register operand when the target is found. So, for example, here when you return from a function; where you are returning is not known here, because you may return to many different places depending on from where you have call. So, it says jr dollar 31. Remember that, we used dollar 31 to save the return address. It will actually take the return address from here and go through the ((Refer Slide Time: 50:51)).

Alright, I will stop here. So, next time, we will try to demystify this ((Refer Slide Time: 50:56))