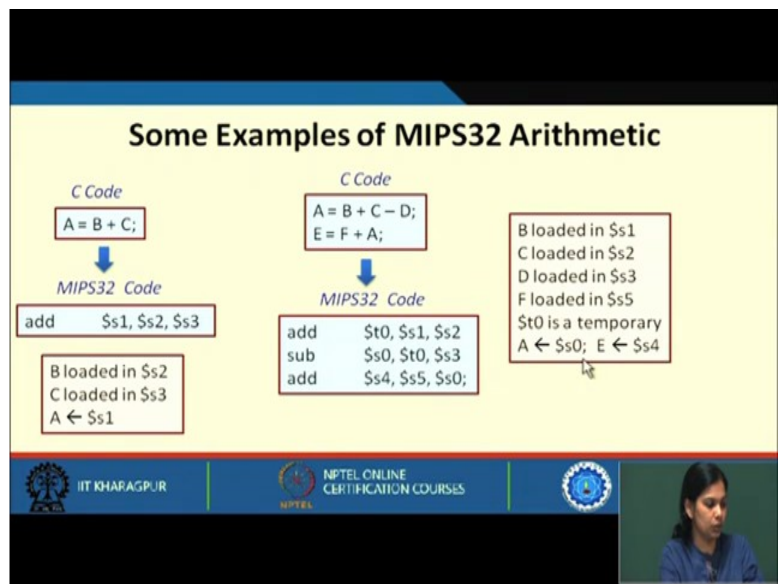
Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture – 10
MIPS Programming Examples

Welcome to the 10th lecture. In this lecture we will be discussing MIPS programming examples.

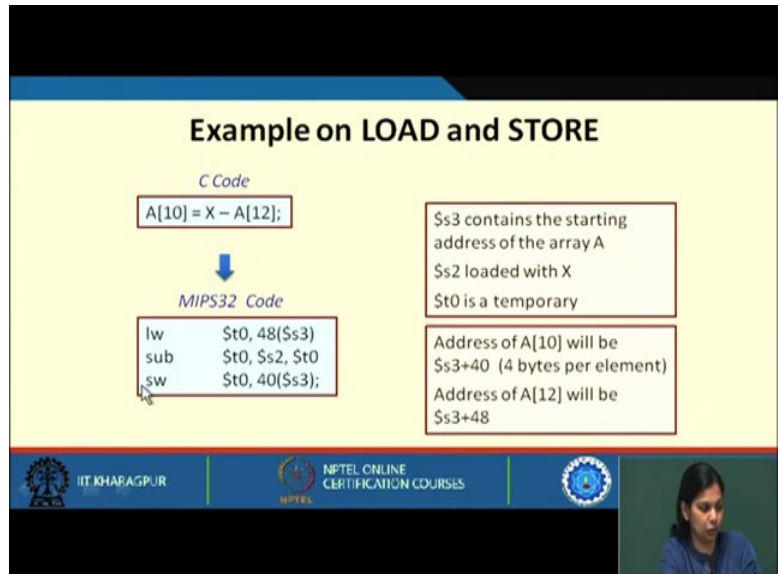
(Refer Slide Time: 00:33)



So, we will start with how C code, that is a code written in high-level language, can be converted into MIPS32 code. So, this code is A = B + C. This code can be converted into add $s1,$s2,$s3. So, B and C should be loaded in $s2 and $s3 and the result will be stored in $s1, that is A.

This is a set of two set of instructions. In the first case it is performing A = B + C - D and in the next we are adding F with the computed value of A. How we can do that? So firstly, add $s1, $s2 and store it in $t0, then subtract $t0 and $s3. So, in $s3 D should be loaded and finally, we add F with A; F is loaded in $s5 and E is in $s0. So, B is loaded in $s1; C is loaded in $s2; D is loaded in $s3; and F is loaded in $s5. And finally, once we compute the result it is stored in $s4, which is E. So, from $s4 we have to move it to E, and partially this result should be stored in A. So, we stored the result $s0 in A.

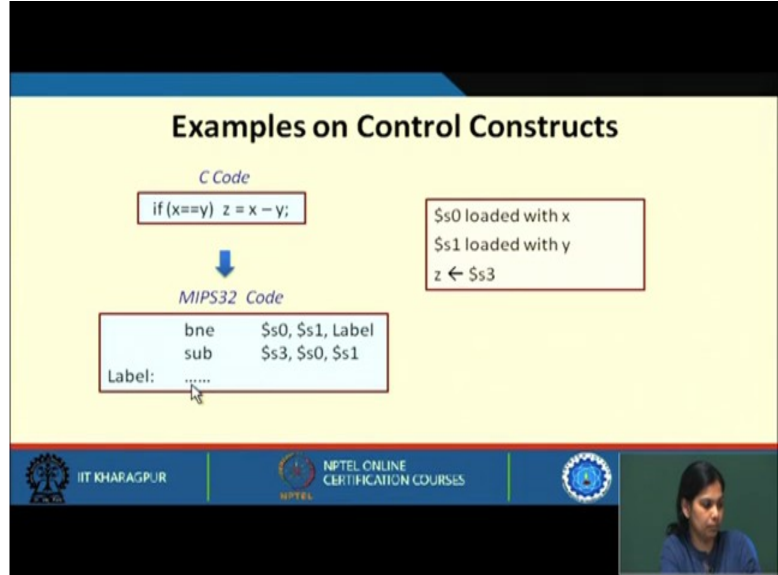
(Refer Slide Time: 03:07)



Let us see some example of load and store. So, this is an array element A[10]. So, A is a location and is an array in a consecutive memory address where we have stored various data. And in the 10th location what we are trying to load X minus A[12]. Let us see how we can convert this code. Firstly, each instruction is 32-bit. So, the 12th word will be 4 * 12 = 48. So, we need to load the initial address of A in $s0 and then we will be adding this initial address of A. that is in $s3 plus 48. And that particular address the word which will be having will be A[12]; that word will be loaded in $t0, then X must be loaded in another register, let us say it is loaded in $s2, and we do $s2 - $t0 and we store it in $t0. And finally, this particular result is stored back in location A[10].

So, the address of A plus 4 * 10 = 40. So, $s3 contains the starting address of an array of this array A. So, this will be starting address plus 4 * 10 and this is starting address plus 4 * 12. So, $s2 loaded with X and once we load this in $t0, then this particular value because in $s3 we have loaded the address added with 48 which will be loaded in $t0. And finally, we subtract it and store it in $t0, and finally this $t0 is stored in this particular location. So, address of A[10] will be $s3 + 44 * 10. And then address of A[12] as I said will be $s3 + 48, that is what I have used it. So, to represent a C code we have to use the following set of assembly language code.

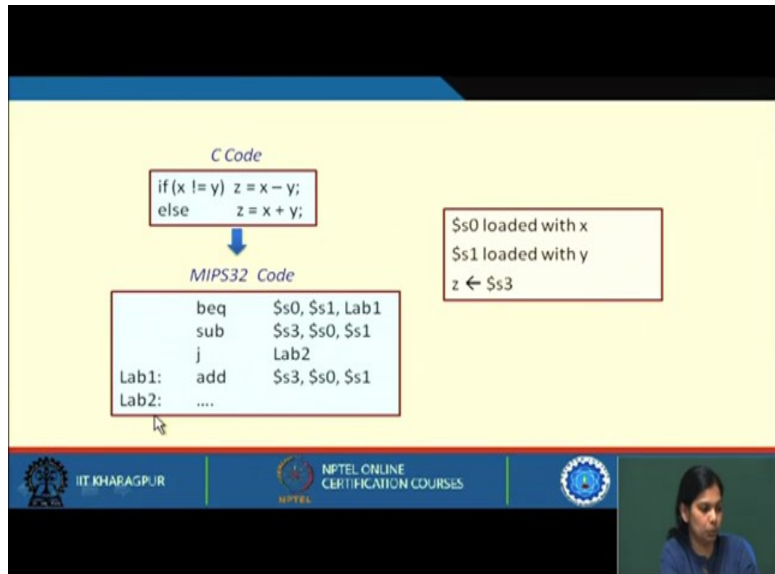
(Refer Slide Time: 06:02)



Let us see some example of control constructs. So, in C we often do this if x equals to equals to y, then do something, or if x not equal to y do something. So, let us see how we can convert this particular C code. So, if equal to then we have to do this if it is not equal then you do something else. So, we take a branch instruction which is branch if not equal; and what this instruction is doing branch if not equal that means, $s0 is not equal to $s1 then go to this Label if it is not equal; if it is equal then this condition will not be met. So, the next instruction will get executed that is what we want it; if x is equal to y, z will be equal to x - y. So, branch if not equal. We are checking for not equal in $s0 and $s1. So, in $s0 and $s1, x and y are loaded; if it is not equal then go to this Label; and if it is equal then the next statement will execute. So, it is a sequential execution.

If this instruction it is a branch instruction and the condition here fails the condition is not matching, $s0 is not equal to $s1. If that is so then it will not go to this particular Label; rather it will go to the next statement that is sub $s3,$s0,$s1. So, it will subtract $s0 and $s1.

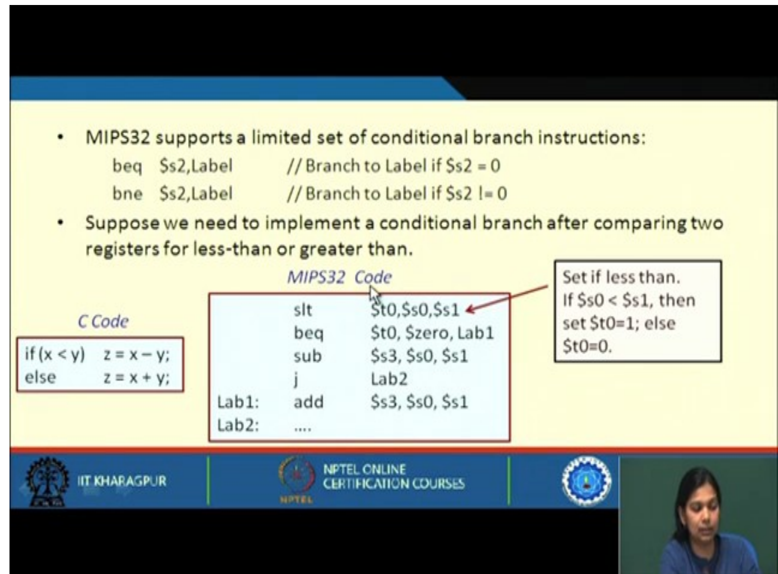
(Refer Slide Time: 08:27)



Let us see some more C code. If x not equal to y then z = x - y; else z = x + y. How the MIPS code can be written branch if equal, so we are not checking for this we are checking for equal. So, if it is not equal then we have to execute this; and if it is equal then they should be executed. So, I am first checking if it is equal. So, I am doing branch if equal $s0 and $s1. In a similar fashion x will be loaded in $s0, and y will be loaded in $s1. So, we load x and y in $s0 and $s1 and check whether it is equal or not. If these two are equal, then we go to Lab1; that means, this else part now I am executing. If both these are equal, we go to Lab1 and add x plus y that is in $s0 and $s1, and store it in $s3.

And let us say if it is not equal then we have to execute x - y. So, I am executing that if it is equal then I am going to label if it is not equal I will execute the next statement. And in the next statement I am doing sub where and subtracting $s0 - $s1 and storing it in $s3 and then I am directly going to jump to Lab2; that means, I will not execute this rather I have to skip this because this will be executed based on this level. So, after this condition is satisfied that is x not equal to y then I have executed this statement; and after this I will go to later part of the code and will not execute this. So, I am jumping it to Lab2.

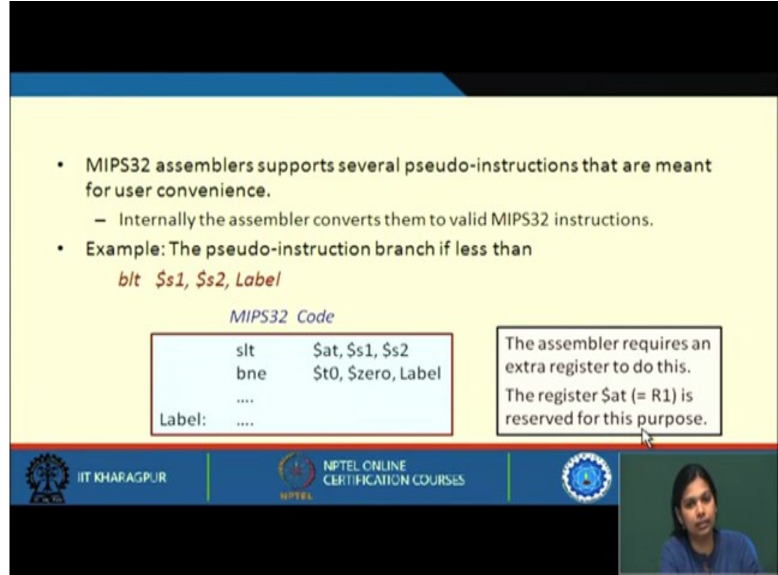
(Refer Slide Time: 10:45)



MIPS support a limited set of conditional branch instructions: branch if equal, branch if not equal. Suppose we need to implement a conditional branch after comparing two registers for less than or greater than then what we will do. If let us say $s1 is less than $s2, I will do this; if $s1 is greater than $s2, I will do this; so like this if x is less than y it should do this else it should do other thing.

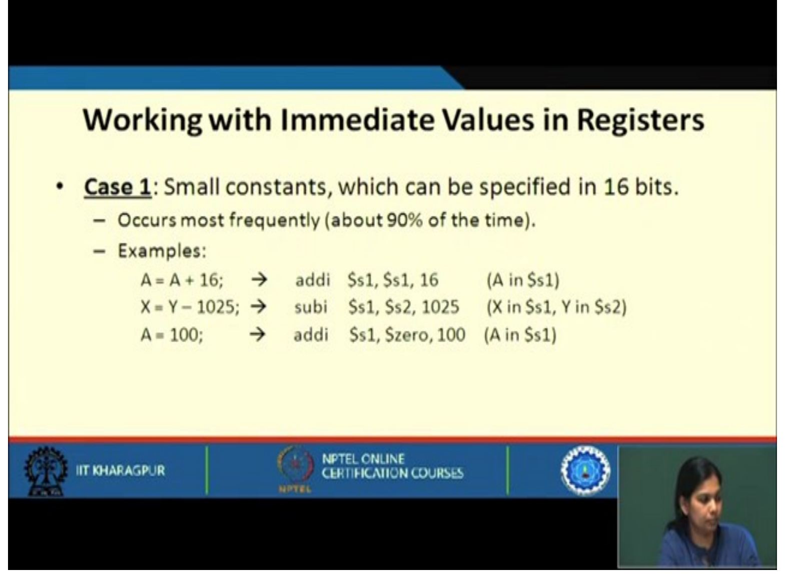
So, there are some instruction set if less than; that means, we will set $t0 to 1 if it is less than that means, if $s0 is less than $s1 we will set $t0 to 1. And then we will do branch if equal if $t0 now equal to 0 then you add that if the else part we are going if it is not equal to zero; that means, x is less than y. So, we are doing sub $s3,$s0,$s1. And similar way we have done in the previous thing, we are jumping to Lab2 because we do not want to execute this particular statement.

(Refer Slide Time: 12:38)



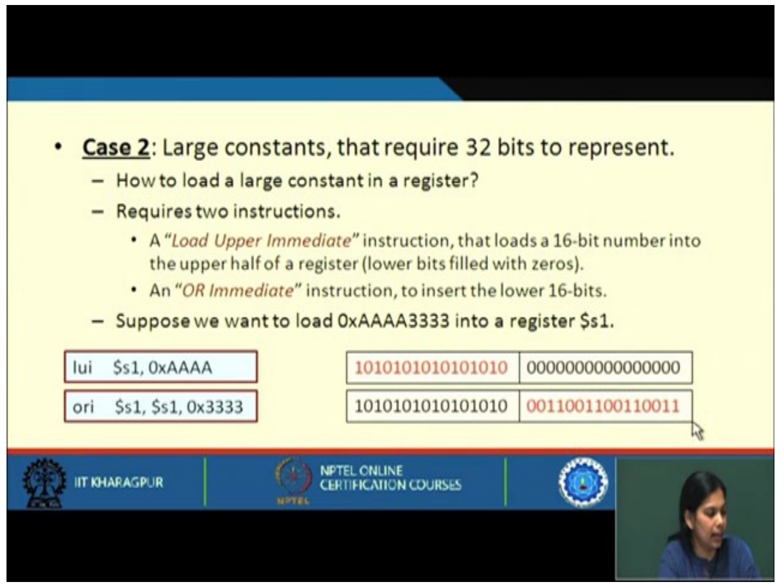
MIPS32 assemblers supports several pseudo-instructions this I have already discussed previously that are meant for user convenience. But even if they are pseudo-instructions internally they have to be converted into some valid MIPS32 instructions. So, some pseudo-instruction we already have for branch less than (blt). This instruction will get converted into slt and bne instructions. The assembler requires an extra register to do this; this register is $at, and the register $at is R1 is reserved for this particular purpose.

(Refer Slide Time: 13:47)



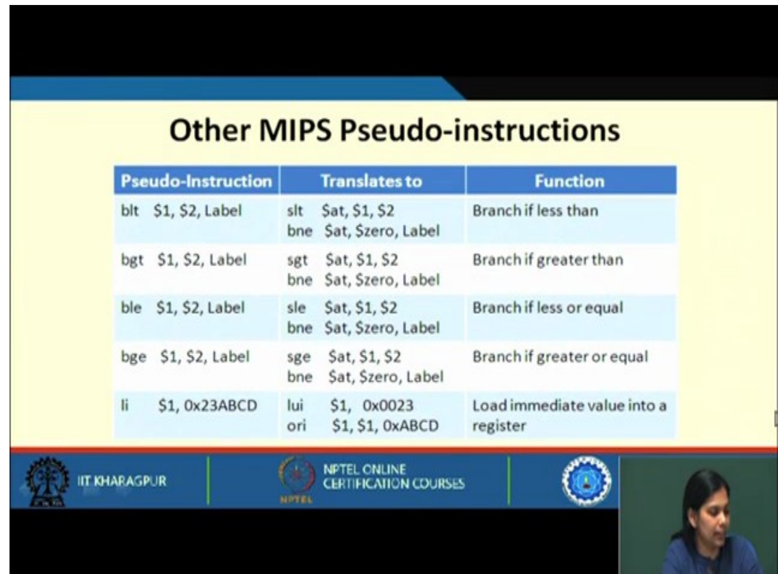
Let us see working with immediate values in registers. In any programming language, we need to add some constant value. So, those values are immediate values that we need to add. So, in such cases this can be specified in 16 bits and it occurs most frequently that is about 90% of the time you have to add some constant value. So, like A = A + 16. So, how will you do it, addi $s1,$s1,16. Now we want to do: subi $s1,$s2,1025. So, in $s2 you have to load Y, and then finally $s1 should be stored in X. A = 100; so in A you have to store this 100. So, $s1 equals to 0, you add zero with 100 and you store it in $s1 and $s1, A in $s1 here.

(Refer Slide Time: 15:00)



In Case 2, we can see that how large constants that requires, say 32 bit to represent, can be loaded. It requires two instructions, the first one is load upper immediate. So, instruction that loads a 16-bit number into the upper half of the register, and then we OR immediate that instruction to insert the lower 16-bit. So, using these two instructions we can load a 32-bit numbers into a register. So, 32-bit number we cannot be loaded with a single instruction rather it requires two instructions to do this. So, load upper immediate. So, in the upper immediate, first we load this here and then or with this 0X3333. So, we or with this and finally, we get that entire thing.

(Refer Slide Time: 16:13)

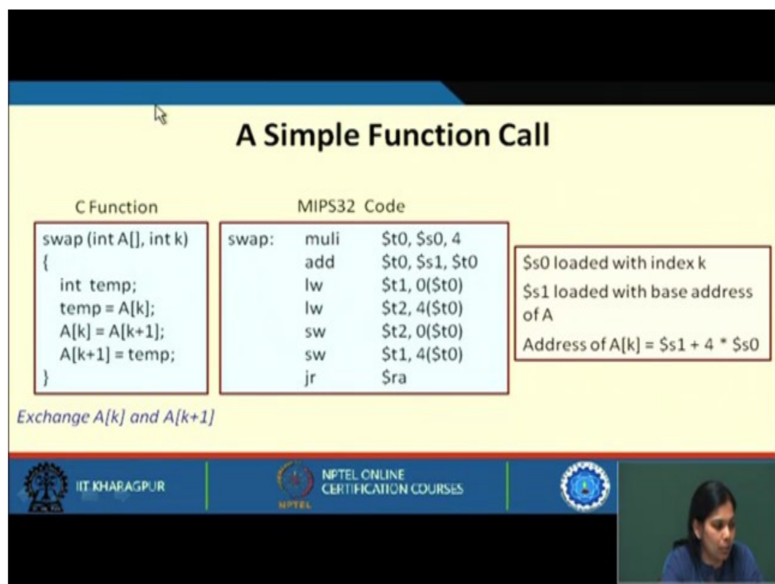


So, there are other MIPS pseudo instructions like branch if less than, branch if greater than, branch if less than equal branch if greater equal. So, while doing programming you will see that you will be using many such pseudo instructions which is required for your programming, but all those pseudo instruction, when you run through any simulator it will get converted into low level MIPS32 instructions.

(Refer Slide Time: 16:48)

There are some more pseudo instructions like move, load address, branch if less than equal, branch if greater than or equal and load immediate.

(Refer Slide Time: 17:01)



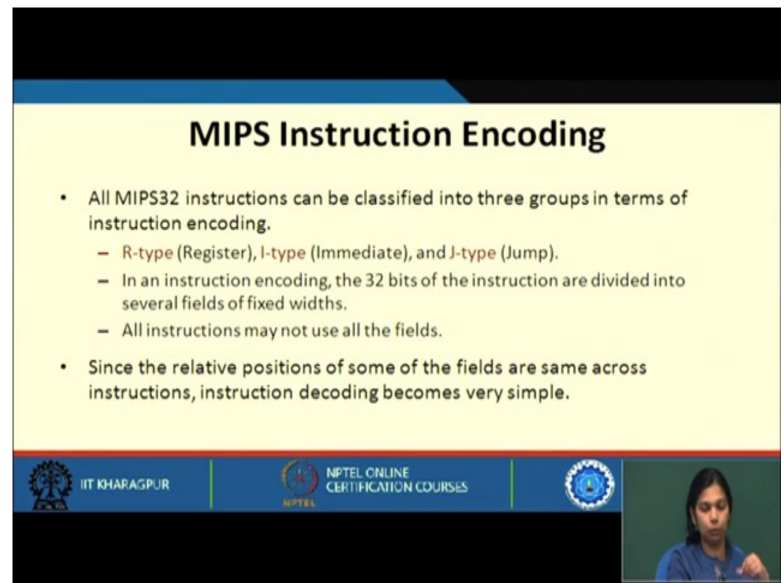
Let us see a simple function call like for adding two numbers. We can simply add two numbers or we can also write a function that will take the two numbers and it will add it will take any two numbers and it will do the needful. Here in this case, it is a swap function. So, in temp we are taking a temporary variable of integer type, and then we are keeping A[k] in that temp, then A[k+1] is stored in A[k], and temp is stored in A[k+1].

Let us see how this can be written using a MIPS32 code. So, first we do muli $t0,$s0,4. So, $s0 is loaded with index k, which index we want to swap that index. And the next one $s1 is the loaded with the base address of A because this particular array we are taking. Then the address of A[k] will be $s1 + 4 into that particular $s0 what will be the index in the same way we did in the previous example. So, muli will multiply $s0 into 4 and it will be stored in $t0 then we are adding $s1 and $t0 and we are storing it in $t0, then we load the two words. We load the two words and we store it in two registers that is $t1 and $t2. So, 0($t 0), $t0 is the base address where base address of A. So, we have added $t0 with $s1 where we have loaded the base address and that is what is in $t0. So, 0($t0) means we will get this whatever is the value of k depending on that that particular word will be loaded in $t1. And the next word will be loaded in $t2.

Now, we simply have to store this particular word in this location, and this particular word in the next location. So, this is what we have done first we are loading the word from 0($t0) into $t1; loading the next word that is $t0 + 4 into $t2. Now $t2 we have to

store it in 0($t1), that is what we are doing store word $t2 into 0($t1) and then we have to store this $t1 into 4($t0) and jump to return address $ra. So, from the main program we came to a function that is swap, we perform this operation, and then we will go back to that function again.

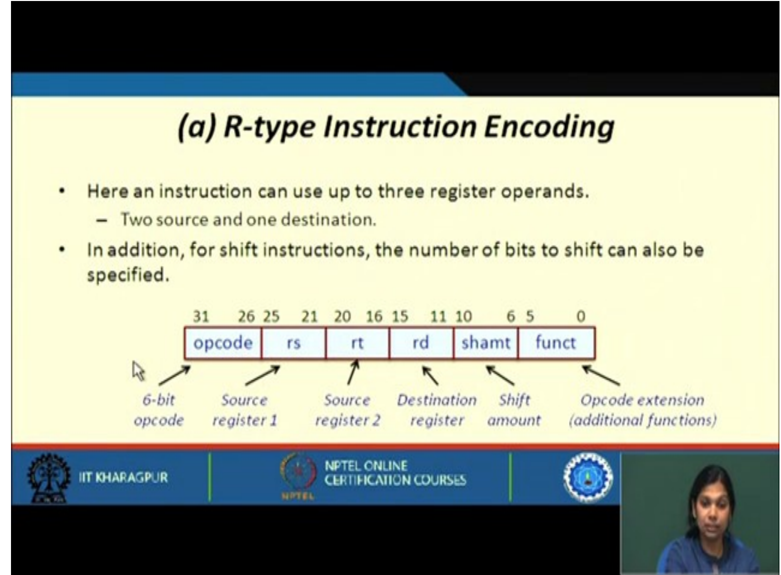
(Refer Slide Time: 20:37)



Let us now see MIPS instruction encoding. All MIPS instructions can be classified into three groups in terms of instruction encoding, you can classify it into three groups. So, the first one is R-type that is register typed; it has got I-type that is immediate and J-type which consists of jump. So, in an instruction encoding the 32 bits of the instructions are divided into several fields of fixed widths. We have already seen instruction encoding from a general perspective. In this case we will be seeing instruction encoding for MIPS and all instruction may not use all the fields, but those will be specified, those are fixed fields. An instruction may use it an instruction may not use it.

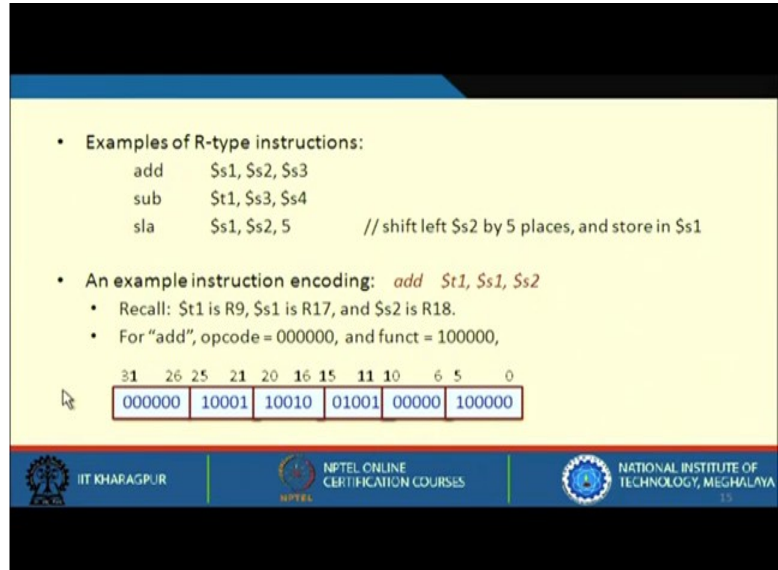
Since the relative positions of some of the fields are same across instruction, instruction decoding becomes very simple. So, as it is already known that this particular field from this particular position to this particular position will be this field; from this particular position to this particular position it will be the other field, and so on. So, the instruction decoding becomes very simpler.

(Refer Slide Time: 22:11)



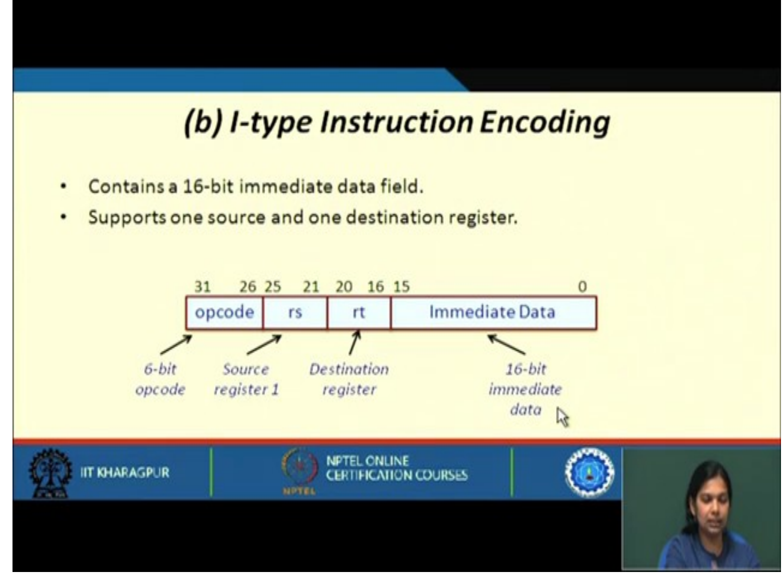
Let us first see R-type instruction encoding. Here an instruction can use up to three register operands. So, there will be an opcode, there will be two source registers rs and rt, and one destination rd. As you know there are 32 total registers, so 5 bits will be specified for this. In addition to this opcode and registers, you will have for shift instruction the number of bits to shift can also be specified, using. this particular field shamt. Let say you want to shift a particular value right to these many position that how much bit position can be specified in this instruction itself. And for that opcode what is the function that is specified here. So, this is an R-type instruction encoding, this is as I said 6-bit opcode, source register one, this is source register two, this is destination register, this is the shift amount the amount of shifting you wanted to do that can be specified here, and the opcode extension. So, if this opcode let say 00000 then we will say that this is an ALU operation and based on that then we will see which ALU operation it is add or mul or div sub etc.

(Refer Slide Time: 24:00)



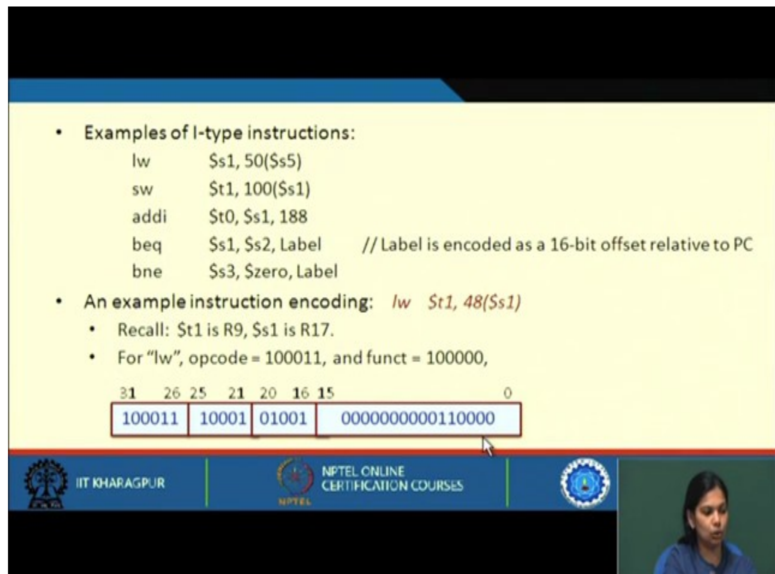
Let us see some example of R-type instruction add $s1,$s2,$s3 like this. So, these are source registers and this is the destination register recall $t1 is R9, $s1 is R17 and $s2 is R18. And for add this opcode is 000000 is for ALU and for add it is 100000. So, now if you see this encoding, so this here goes your opcode that is 000000 and then the function goes here that is 100000. First register is $t1, or R9, so this is 01001. Similarly, $s1 and $s2 are the two source registers R 17 or 10001, and R18 or 10010.

(Refer Slide Time: 25:17)



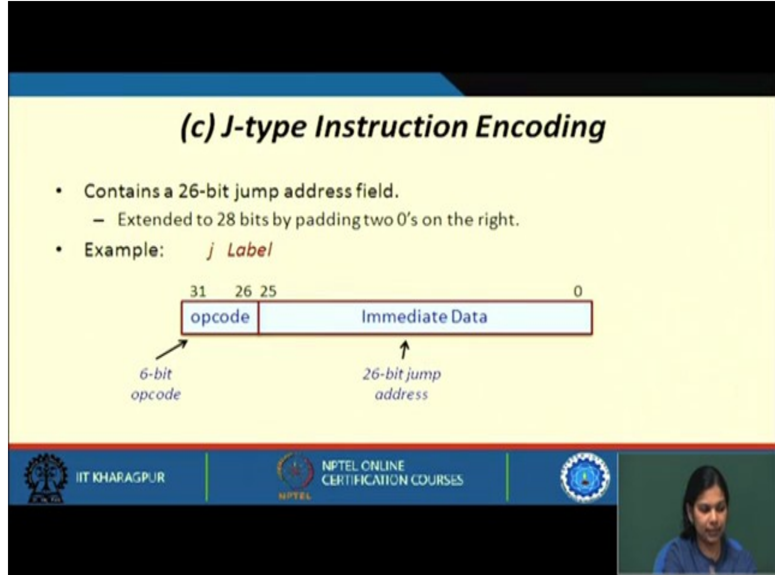
Next move on with the next type of instruction that is I-type instruction; it contains 16-bit immediate data value. Now, suppose this total 16-bit is an immediate data and you have a source register and you have a destination register, this is all you have and you have a single opcode. So, for any immediate value, we do not require other fields, but see these values are fixed, this is source, this is destination, this is opcode, but this 16-bit immediate value has changed for I-type instruction. So, this is 6-bit opcode, source register, destination register and this is 16-bit immediate data.

(Refer Slide Time: 26:16)



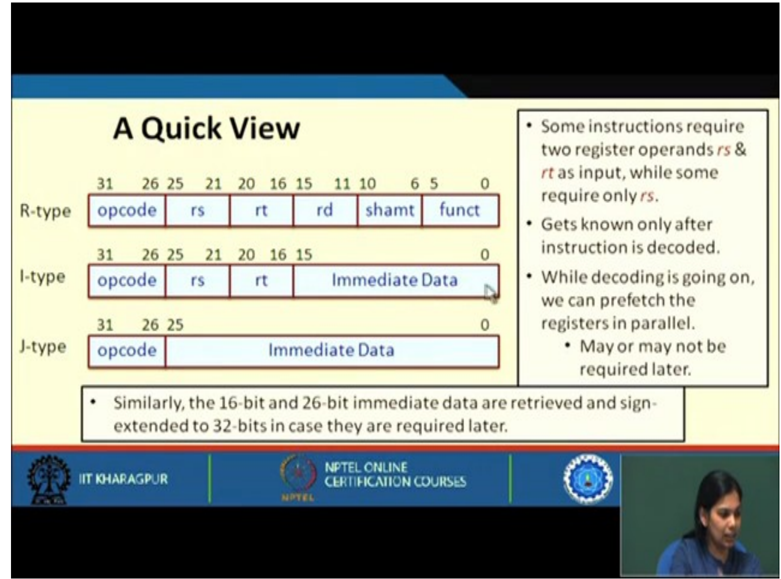
Now, you see some of the I-type instructions in MIPS32. Load word (lw), so from this memory location 50 + $s5 it will load that particular word from this memory location into here. So, all these type of instructions are I-type. So, let us see this example lw $t1, 48($s1). $t1 is the destination register and this one $s1 is the source. And 48 is the immediate value and the opcode for lw is 100010. So, let us see how this will be encoded. Now, this is the opcode that I have written here $t1 is R9 0r 01001, I have to load the word into this destination registers and $s1 is the source register, which is R17 or 10001. And this is my immediate value that is 48 is loaded here. So, this is a kind of I-type instruction.

(Refer Slide Time: 28:04)



Let us go to J-type instruction encoding it contains a 26-bit jump address field. And this 26-bit jump address field is padded with two zeros, so that it can be extended to 28 bits. And an instruction like j Label. So, when we give jump to Label, so it goes to a 26-bit address by adding two more bits it becomes 28 bit and we move there. So, these this type of instruction is jump type instruction 6-bit opcode and 26-bit jump address.

(Refer Slide Time: 28:55)



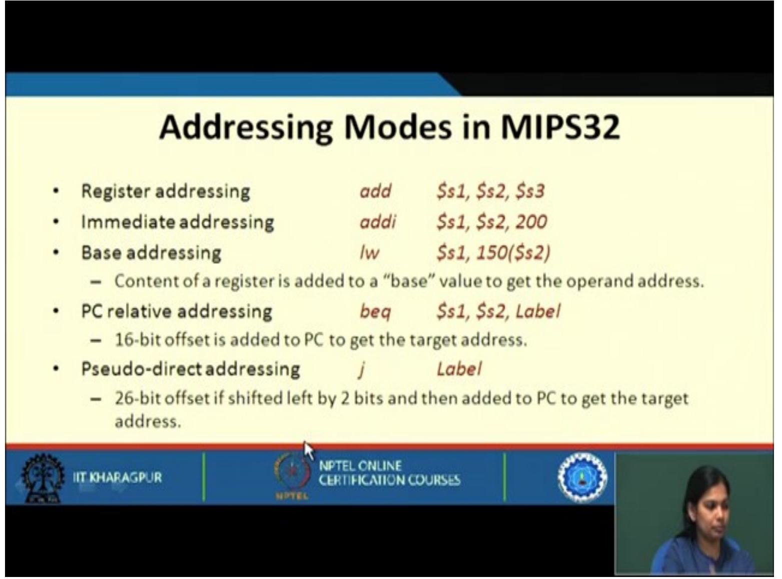
So, now you see that there is a clear place this is your source, this is another source, this destination and this can be a destination for some. And this is the immediate data and this

is the immediate data for the next one. But now if you have to check the opcode field, you can check the opcode field for all from the first part and of course, for some you have to check this. Some instruction requires two register operands like we said rs and rt as input while some requires only rs. And get to know that only after instruction decoded, after the instruction get decoded then only we will know that ok, it has got these many source register this many destination register.

And while decoding is going on we can prefetch the register in parallel may or may not be required later. So, what I am trying to say that we already know that this is a source this is a source this is a source. So, well in advance we can prefetch it, maybe we may not be requiring it for a jet you may for this j type instruction we will not be requiring this rs and rt, but in that time we can already prefetch it because for this instruction and this instruction we may might require it later. So, if you prefetch it, we may require it and we can save some time.

So, similarly this 16-bit address and this 26-bit address immediate data, these are immediate data some address. These immediate data retrieved and then we can do a sign extension to make it 32 bit. We already know what is sign extension. So, we can extend this to 32 bit and if it is required later it is fine if it is not required later it, but we can do this.

(Refer Slide Time: 31:11)



Now, what are the addressing modes in MIPS we have, we have register addressing, we have immediate addressing, we have base addressing where the content of register is added to a base value to get the operand address, we have relative addressing here and we have pseudo direct addressing. So, the 26-bit offset is shifted by left by 2 bits and then added to PC to get the target.

So, we came to the end of lecture 10 and next we will see some of the programs that we can do using MIPS.

Thank you