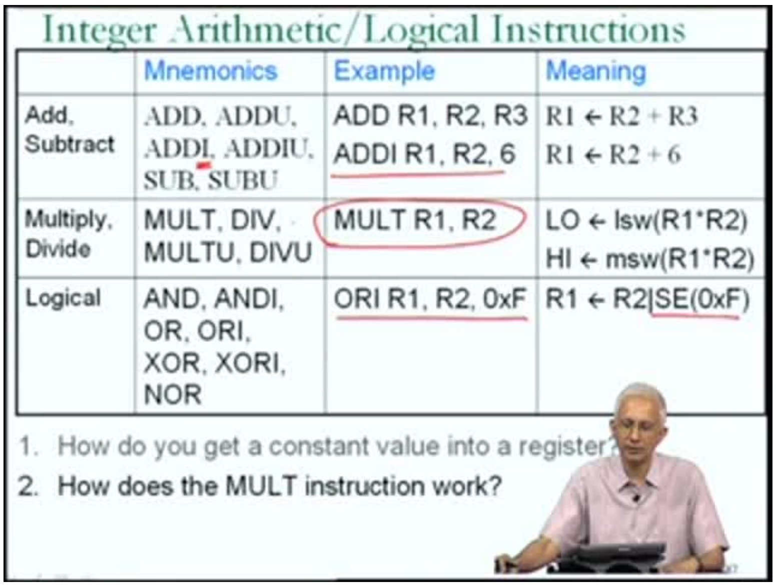
High Performance Computing
Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture No. # 06

Welcome to lecture 6 of a high performance computing. We are in the process of looking at the MIPS 1 Instruction set in some detail. In the previous lecture, we saw that the MIPS 1 Instruction set is the 32 bit; in other words, 32 bit worth size. Load store in other words, the only instructions that take memory operands are the load and store instructions, and it is a RISC Instruction set. We saw that the 32 general purpose registers R0to R31 of which, R0 is always contains the value 0, and R31 is implicitly used by some instructions.

(Refer Slide Time: 00:52)



Integer Arithmetic/Logical Instructions

	Mnemonics	Example	Meaning

Add, Subtract	ADD, ADDU, ADDI, ADDIU, SUB, SUBU	ADD R1, R2, R3
		ADDI R1, R2, 6
			R1 ← R2 + R3
			R1 ← R2 + 6

Multiply, Divide	MULT, DIV, MULTU, DIVU	MULT R1, R2	LO ← lsw(R1*R2)
			HI ← msw(R1*R2)

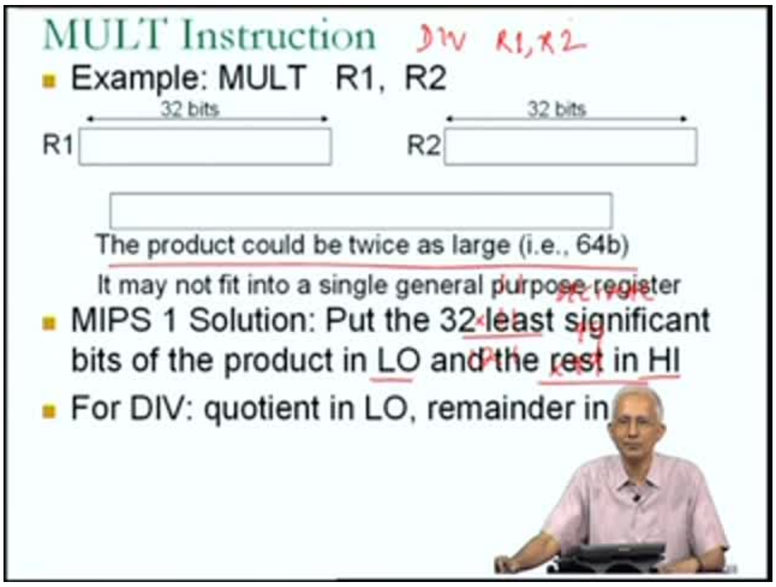
Logical	AND, ANDI, OR, ORI, XOR, XORI, NOR	ORI R1, R2, 0xF	R1 ← R2|SE(0xF)

1. How do you get a constant value into a register?
2. How does the MULT instruction work?

We saw that there are 2 other additional registers called HI and LO. We saw the addressing modes which are used by the instructions, and also looked in detailed at two of the categories of instructions - the data transfer instructions, in other words, loads, stores and moves to and from the registers HI and LO as well as the arithmetic and logical instructions. We are in the process of actually looking at the table of arithmetic and logical instructions.

Let me remind you that there are add and subtract instructions which can also take immediate operands, such as, the second example, and the logical operations also have the possibility of immediate operands. The immediate is are all signed 16 bit quantities which may have to be sign extended if they have to be operated on with 32 bit operand. All that means to be understood in this table, other multiply and divide instructions which have the audit that they are the only instructions in the MIPS 1 Instruction set, which do not seem to have a destination operand specified which we need to decipher.

(Refer Slide Time: 02:01)



MULT Instruction DIV R1, R2

- Example: MULT R1, R2

32 bits
R1
32 bits
R2

The product could be twice as large (i.e., 64b)

It may not fit into a single general purpose register

- MIPS 1 Solution: Put the 32 least significant bits of the product in LO and the rest in HI
- For DIV: quotient in LO, remainder in

So, let me just go through an example. Here is an example of multiply instruction, multiply R1 R2. So, we suspect that R1 and R2 are the source operands of this instruction, and that what the instruction is supposed to do is to take the 32 bit value out of R1 multiplied by the 32 bit value out of R2, and the only question is what is it supposed to do with result? There are first thing we need to understand is that the result is conceivably larger than 32 bits. There in result could in fact be as large as 64 bits, and I guess all of you, many of you may have an understanding of this from the decimal equivalent.

If I have to, let us suppose I am talking about decimal, if I have to two digit numbers and I multiply them together, you know that the result is in this case three digits, but in general could be as high as four digits. So, when I multiply to two digit numbers, I could get a number which is as big as 4 digits.

When I multiply to two digit numbers, the result could be less than four digits. Which is why in this comment appear, I mentioned that when I multiply to 32 bit values, the result could be as large as 64 bits. This is the problem for the MIPS Instruction set, because you will remember that the size of all the general purpose registers is only 32 bits, and that is why when I do a multiplication of 2 register values, there is no guarantee that the result can fit into one of the general purpose registers, and this is why the HI and LO come into the picture.

So, what happens in the case of the MIPS 1 Instruction set is that, the result of the multiplication. In other words, the product is put into the registers HI and LO, and as the name suggest, the least significant bits of the product going to the register LO and the most significant 32 bits of the product go into the register HI, and you know that both HI and LO from the previous lecture, what I told you? Both HI and LO are 32 bit registers, and therefore, the full 32 bits of the product can be stored and made available to the program.

And typically you would expect that the program will have to check to see whether high is equal to 0, and if high is equal to 0, then that the full product is contained within the register LO and proceed accordingly.

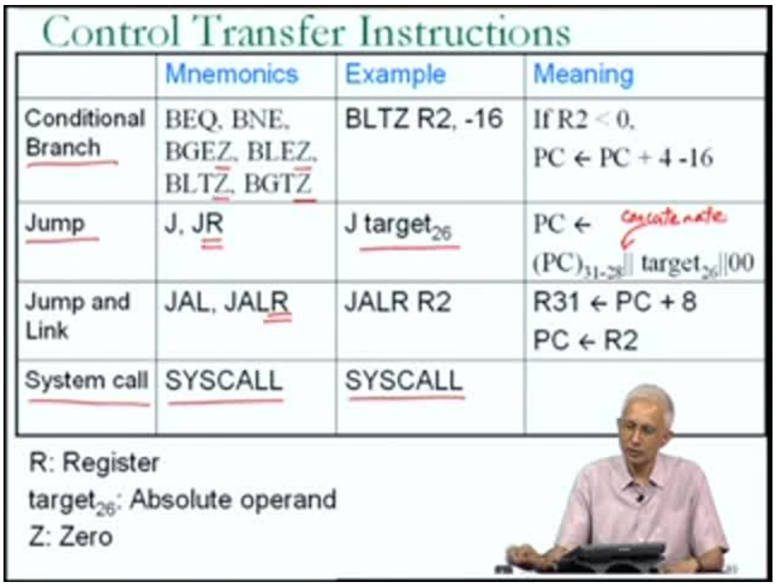
Now, what about the divide instruction? The divide instruction like the add instruction has only 2 operand, for example, I might divide R1 by R2, but in the case of divide, you know that the result a quotient, but there is also an other result which is the reminder.

How big could the quotient be? We know the, there is no problem with the size of the quotient; there is no problem that the size of the reminder; both of them can only be as large as 32 bits, only questions where should those 2 go? So, this, if I had a divide instruction in the MIPS 1 Instruction set, problem would then be where do I, how do I specify 2 destination operands; in other words, 1 destination operand for the quotient; the other destination operand for the remainder in a single instruction.

And therefore, once again to bypass this problem, they use the HI and LO registers. The quotient goes into LO and the reminder goes into HI, and this is basically all we need to know about the multiplying divide instructions.

So, in effect to do a multiplication or division, one can use the single instruction but they will be follow up instructions to transfer the result, whether it be the quotient or the 32 bits of the product into one of the general purpose registers for the subsequent computation by the program.

(Refer Slide Time: 05:57)



Control Transfer Instructions

	Mnemonics	Example	Meaning
Conditional Branch	BEQ, BNE, BGEZ, BLEZ, BLTZ, BGTZ	BLTZ R2, -16	If R2 < 0, PC ← PC + 4 -16
Jump	J, JR	J $target_{26}$	PC ← concatenate $(PC)_{31-28}$ || $target_{26}$ || 00

Jump and Link	JAL, JALR	JALR R2	R31 ← PC + 8
			PC ← R2

System call	SYSCALL	SYSCALL	

R: Register
$target_{26}$: Absolute operand
Z: Zero

Now, with this, we have actually completed our quick look at the MIPS 1 arithmetic and logical instructions, and we will now move on to the control transfer instructions. In some sense, this is the largest and the most important category of instructions, because these are the instructions which are going to be used to create the control flow of the c program and is the part of programming which is probably the most challenging.

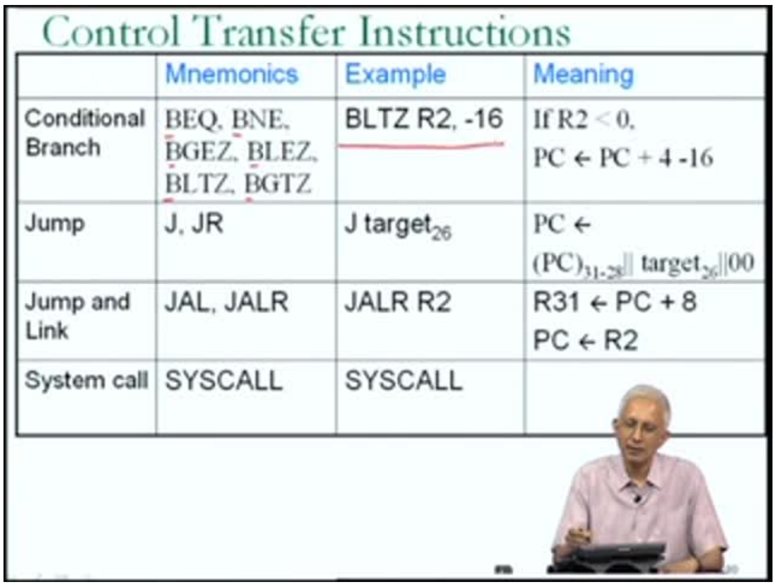
Now, as your expect, in this table, had we want you that I am going to have to talk separately about the conditional branch instructions or branch instructions, the unconditional jump instructions. A family of instructions which are going to be used to implement function calls, which are call the jump and link instructions and some other instructions. I have actually introduce the system call instruction at the bottom and I am not going to say anything about it today.

But we will come back to this instruction after 7 or 8 lectures well into the course. Therefore, please do remember that I talked about the system call instruction, and with that, I will not talk about any more today.

Now, in terms of a terminology used in this table, one piece of terminology that is used is the letter R is used in this table to stand for register. In some of the notation, I have used target sub 26, and target sub 26 is referring to an absolute operand of 26 bit size.

Some of the instructions have a Z in the mnemonic and Z stands for zero, and the any other notation which I have used here is that in this particular meanings column, I have used a two pipe operator by which you can understand that I mean the operation, catenate or concatenate whichever you are more used to.

(Refer Slide Time: 07:59)



Control Transfer Instructions

	Mnemonics	Example	Meaning
Conditional Branch	BEQ, BNE, BGEZ, BLEZ, BLTZ, BGTZ	BLTZ R2, -16	If R2 < 0, PC ← PC + 4 -16
Jump	J, JR	J target_{26}	PC ← $(\text{PC})_{31-28} \| \text{target}_{26} \| 00$

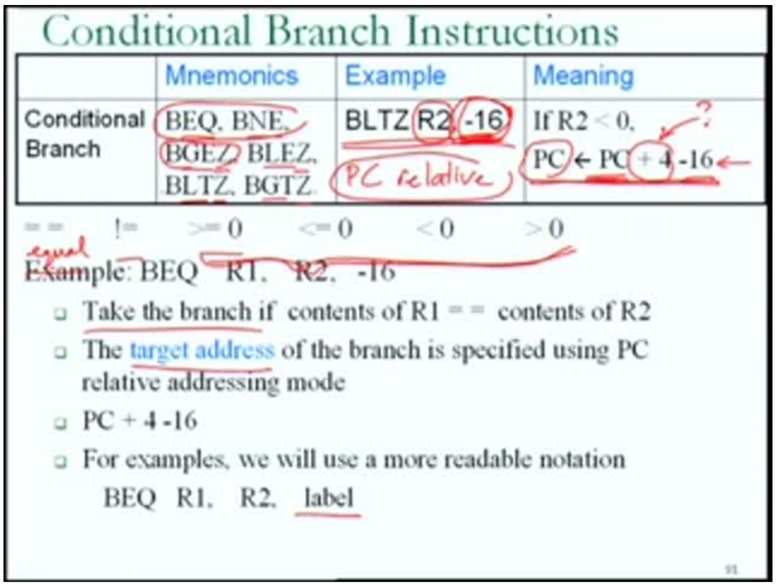
Jump and Link	JAL, JALR	JALR R2	R31 ← PC + 8
			PC ← R2

System call	SYSCALL	SYSCALL	

So, with this quick overview of the terminology, we can try to understand these instructions one by one. So, let me just remind you the conditional branch instructions or branch instructions, and B stands for branch in all of these instructions will be used to transfer control of the program depending on whether a particular condition is true or not. I suppose to the jump or unconditional branch instructions which transfer control unconditionally.

So, if you look at this particular example, BLTZ R2 minus 16 and look at the explanation, it looks quite mysterious.

(Refer Slide Time: 08:39)



Conditional Branch Instructions

	Mnemonics	Example	Meaning
Conditional Branch	BEQ, BNE, BGEZ, BLEZ, BLTZ, BGTZ	BLTZ R2, -16 PC relative	If R2 < 0, PC ← PC + 4 -16 ?

== != >= 0 <= 0 < 0 > 0

equal

Example: BEQ R1, R2, -16

- Take the branch if contents of R1 == contents of R2
- The target address of the branch is specified using PC relative addressing mode
- PC + 4 -16
- For examples, we will use a more readable notation

  BEQ R1, R2, label

91

So, just still I need to explain this little bit more detail. So, let me separately talk about the conditional branch instructions. So, we now have the same example branch if less than 0 and the meaning of it, but let me just go through these mnemonics one by one.

So, we have BEQ, we need to be able to read this in a slightly friendly of fashion in BEQ, B and E, BGEZ, etcetera. We know that Z stands for zero and we expect that EQ stands for equal that the condition which we which is being tested is a test for a quality. Using that is a hint, we would guess any e stands for not equal, GE stands for greater than equal, LE for less than or equal, LT for less than and GT for greater than; which means that the kinds of conditions that are being tested by these instructions are equal which you understand in c by the equal operator. NE which is not equal greater than or equal to 0.

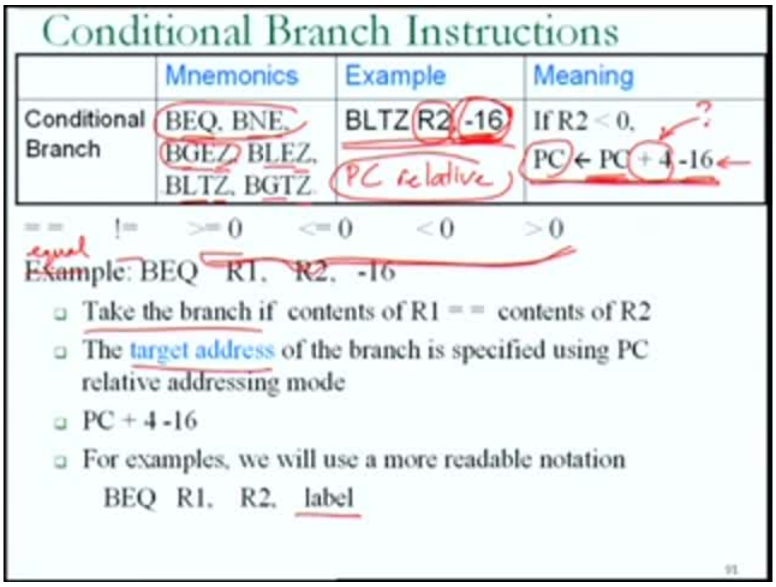
Remember that the Z at the end of BGEZ stands for 0, less than or equal to 0, less than 0 and greater than 0. So, these mnemonic suggest that these are the conditions which are being tested whether two things are equal, whether they are not equal, whether something is greater than are equal to 0, less than or equal to 0, less than 0 or greater than 0. From this example, we understand that for the comparison with 0 instructions, the value inside the register is being compared with 0.

So, this instruction basically means branch if less than 0 R2, which you could read us branch if R2 is less than 0 2 minus 16.

Now, in the lead up to this, in the in the previous lecture in talking about the different addressing modes used by the MIPS 1 Instruction set, I had indicated that the branch instructions use a PC relative addressing mode. And in the case of a branch instruction, the first operand is very clearly being specified in the register director addressing mode, and therefore, the PC relative addressing mode must be referring to the second operand, this minus 16. Which is why we are not too surprised to see in the meaning column that the meaning of the minus 16 is that the way that the program counter is change. In other words, the way that the control flow of the program is modified is by taking the whole value of the program counter and subtracting 16 from it. At this point, we are not too sure why this plus four is there. That will become clearer when we talk about how the hardware to implement this instruction set could be implemented.

But for the movement, we must understand that this is the meaning of the instruction. Then if I execute an instruction branch if less than 0 R2 minus 16, the meaning is that if the current value of the register R2 is less than 0, then the program counter will be modified. In other words, there will be a change of control to the program counter of the branch instruction plus 4 minus 16, and the minus 16 is because of the fact that this is PC relative addressing.

(Refer Slide Time: 12:01)



Conditional Branch Instructions

	Mnemonics	Example	Meaning
Conditional Branch	BEQ, BNE, BGEZ, BLEZ, BLTZ, BGTZ	BLTZ R2, -16 PC relative	If R2 < 0, ? PC ← PC + 4 -16

==   !=   >= 0   <= 0   < 0   > 0

equal

Example: BEQ R1, R2, -16

- Take the branch if contents of R1 == contents of R2
- The target address of the branch is specified using PC relative addressing mode
- PC + 4 -16
- For examples, we will use a more readable notation

  BEQ R1, R2, label

91

So, something similar is going to happen for all of the instructions which compare with 0, but what about the instructions that do not compare with 0, in other words, the branch if equal and the branch if not equal?

Once again, if there is a condition which is being checked and the a check of equality is being done, then two things must be being checked for a quality. So, the question is what is a BEQ instruction look like, and the answer is there. In the case of the BEQ and the BNE instruction, there are actually 2 operands. In fact, an example would be BEQ R1 R2 minus 16, and this instruction basically reads branch to minus 16 if R1 is equal to R2, a branch if equal R1 R2 minus 16, and similarly, for the branch if not equal.

So, this family of instructions is adequate for many of the conditions that you will encounter, but if there are conditions which do not directly fall under the set of six comparisons, then one must find the way of achieving those conditions using these six instructions which is the programming challenge or the challenge to the compiler.

So, the idea of the BEQ R1 R2 minus 16 is, that the branch will be taken if the contains of R1 are equal to the contains of R2, if the contains of these 2 general purpose registers are equal.

Now, in terms of terminology, you should note that take the branch is the same as saying that the program counter will be modified to something that it is not just what it would have been if the branch are not been taken, and in terms of terminology, one often talks about the target address of a branch. The target address is the branch; I mean the address to which control will be transferred if the branch is in fact taken as indicated by the PC relative addressing mode.

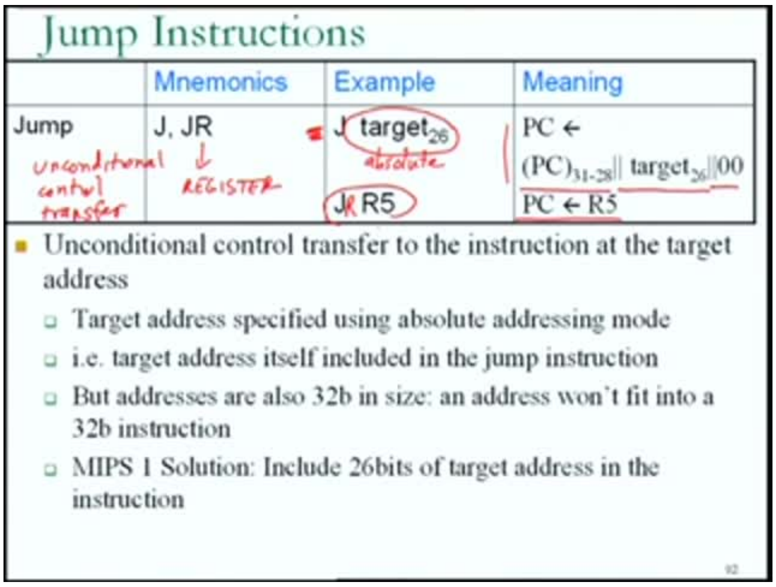
So, from now I will refer to target address of a branch or even the target address of a jump, because even for a jump, there is an address to which control will be transferred, and in general, we will refer to this as the target address, and so, this particular example the branch if less than 0 example, the target addresses PC plus 4 minus 16, and as I said we do need to understand what the plus 4 is. That will happen a little bit later.

Now, in general, for the examples that we write, it will be somewhat inconvenient to write minus 16 and then, to actually count down in the program or up in the program to see which is the instruction, which is other particular address.

So, rather than planning on doing that for writing examples in this course, I will rather go for the alternative of using a c like notation for the specification of the PC relative addressing mode, and I will explicitly just label instructions of the program, and instead of putting the PC relative address of the target into the instruction, I will just put the label corresponding to the target in the instruction.

This would make a lot easier for us to read and understand code sequences in the rest of this course. So, I am postponing only one thing in this discussion and that is to explain to you in more detail with this plus 4 is coming from; otherwise, we have understood how these conditional branch is operate.

(Refer Slide Time: 15:26)



Jump Instructions

	Mnemonics	Example	Meaning
Jump unconditional control transfer	J, JR $\downarrow$ REGISTER	J target$_{26}$ absolute JR R5	PC $\leftarrow$ (PC)$_{31-28}$ || target$_{26}$ || 00 PC $\leftarrow$ R5

- Unconditional control transfer to the instruction at the target address
  - Target address specified using absolute addressing mode
  - i.e. target address itself included in the jump instruction
  - But addresses are also 32b in size: an address won't fit into a 32b instruction
  - MIPS 1 Solution: Include 26bits of target address in the instruction

So, let us move on to the jump instructions. So, in general, j stands for jump, and I had indicated that R stands for register. In general, jump is unconditional control transfer.

So, we have two examples - jump to target and the target is specified in the instruction as a twenty six bit absolute. Remember that in our notation slide, I talked about target 26 is being a absolute, something which is specified absolute addressing mode. In other words, it is inside the instruction. The second example is an example of JR, I am sorry, it should be in R; here, JR R 5.

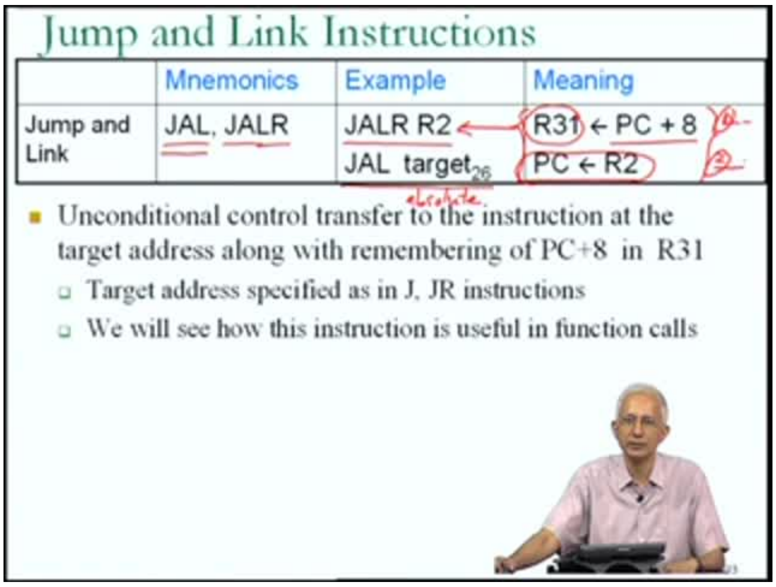
So, what is these instructions do? The second instruction very clearly changes the value of the program counter, so, the program counter now contains whatever value is present in R5, but the first example is little bit more complicated as can be seen from the meaning.

So, how is the program counter manipulated in the case of the first example? Now, the problem that we are seeing over here is as any jump instruction is an unconditional control transfer. In both these examples, the objective is to transfer control to the instruction at the target address, and the target address is specified using the absolute addressing mode.

The target address itself has to be included inside the jump instruction. Since the target address is the size 32 bits. Since the target address is going to be the address of an instruction into conceivably has to be 32 bits in size, but obviously I cannot include a 32 bit target address inside a 32 bit instruction. We saw this problem in the previous lecture, and therefore, the MIPS 1 solution is that you allow to specify a 26 bit target address, and the hardware uses this 26 bit target address to construct the 32 bit address by adding two 0's as least significance bits of the final target address, and the remaining bits of the target address I just taken from the current value of the program counter. In other words, the value of the program counter associated with the jump instruction itself.

So, this is the way that they overcome the problem of not being able to specify a 32 bit target address, and it may be a little bit restrictive in the range of addresses they can be used as targets for the jump instruction, but if that is the case, then one can always use the jump register instruction, in which case the 32 bit target address can be placed into the register so much for the jump instruction, and the jump register instruction.

(Refer Slide Time: 18:14)



Jump and Link Instructions

	Mnemonics	Example	Meaning
Jump and Link	JAL, JALR	JALR R2	R31 ← PC + 8 ①
		JAL target$_{26}$	PC ← R2 ②

- Unconditional control transfer to the absolute instruction at the target address along with remembering of PC+8 in R31
  - Target address specified as in J, JR instructions
  - We will see how this instruction is useful in function calls

With this, we will move on to the next category of MIPS 1 control transfer instructions, the third line in our table. These are the jump and link instructions of which once again there are two, and now, we can read this as jump and link and this as jump and link register.

So, what are these instructions to do? First of all let me just give you an example of each. Jump and link register we saw as in the case of the jump register instruction from the previous table would have a single operand which is a register operand. Whereas, the jump and link instruction has single operand which is a specified in absolute addressing mode and the meaning example over here applies to the first example not to the second example.

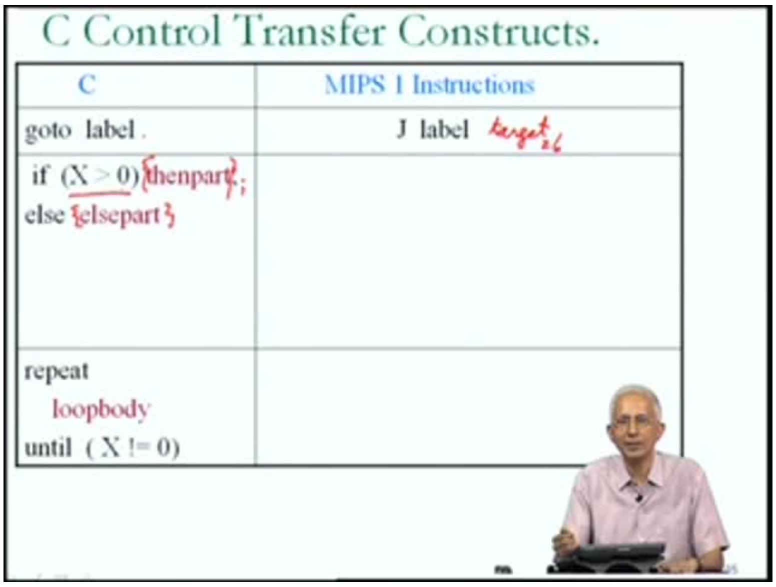
Now, the meaning that we see over here is little bit interesting. This is the first instance that we are seeing of a MIPS 1 Instruction, in which, the meaning of one Instruction involved two operations. I could know all of the situations that we had the meaning was a single line or if you there was one situation where the characters would not fit on to a single line, but here there are two separate operations which are happening as part of the jump and link instruction; which means that this is the very special kind of an instruction, it does more than one simple thing, and we look at this in more detail. We see that it does two things - one of them is to manipulate R31. It puts PC plus 8 into R31, even though R31 is not an explicit operand of this instruction.

And then, it modifies the program counter to now contains the value of R2. This is a somewhat curious sequence of instructions, and so, it seems to unconditionally transfer control to that, to that instruction at the target address. That is what the second step is doing. While at the same time, remembering PC plus 8 in register R31, which is what the first instruction is doing, which is what the first operation in the sequence was doing.

So, this is the first example that was seeing in the MIPS 1 RISC Instruction set of an instruction which is doing something which is not a really primitive operation, because in order to describe this, I have to use two primitive operations, and therefore, this must be really important; otherwise, I could not have included it, is the deviation from the RISC. In some sense, it is a deviation from the RISC design principle of trying to have every each instruction doing only one single operation.

So, as we will see, I believe in the next lecture, possibly the lecture after that, this instruction is critical for the implementation of function calls and that is why it had to be included, and how it is useful in the implementation of function calls, we will see shortly.

(Refer Slide Time: 21:11)



C Control Transfer Constructs.

C	MIPS 1 Instructions
goto label .	J label target_{26}
if $(\underline{X > 0})$ {thenpart}; else {elsepart}	
repeat loopbody until (X != 0)	

Now, before actually moving to looking our programs, in general, I want you to just try to relate control transfer as you know it from your c programming experience to control transfer as we must view it now in the MIPS 1, well. Now, I was here, in this table, I have three examples of control transfer from the c, from your c world, there is a go to statement; there is an if then else construct, and there is a repeat loop.

The repeat loop is one example of a loop. I could just as well I have included a while loop or a for loop, and after having gone through the repeat loop, you will see the, yourself will be able to fill up the table with additional entries for a while loop or for a for loop without any great difficulty.

So, the purpose of this table is to give us a good understanding of what we can expect to see in a MIPS 1 program in place of a go to. In our original c program, if there was a go to, what can we expect to see in the MIPS 1 program? If you can, if in our original program, there was an if than else what can we expect to see in the MIPS 1 program and so on.

So, let us start by thinking about the go to. So, go to in a c program is an example of a unconditional control transfer. Control is to be transfer to the statement which has that particular label unconditionally, no need to check any condition, and so, we suspect them in the MIPS world. The same effect is going to be achieved using an unconditional control transfer instruction. Therefore, either a jump instruction or a jump registers instruction.

Since in our current terminology, I am using labels in my MIPS 1 programs, but we will actually see is that go to label will map into a jump instruction to the same label.

Now, there is a remote possibility that you will recall, you will recall that under this notation, label will be specified using the target 26 absolute addressing mode inside the jump instruction. Therefore, as I had suggested in the slide on jump instructions, there is a possibility that if the target address of this jump. In other words, there are label, the statement which is label by the go to is very far away from the jump instruction.

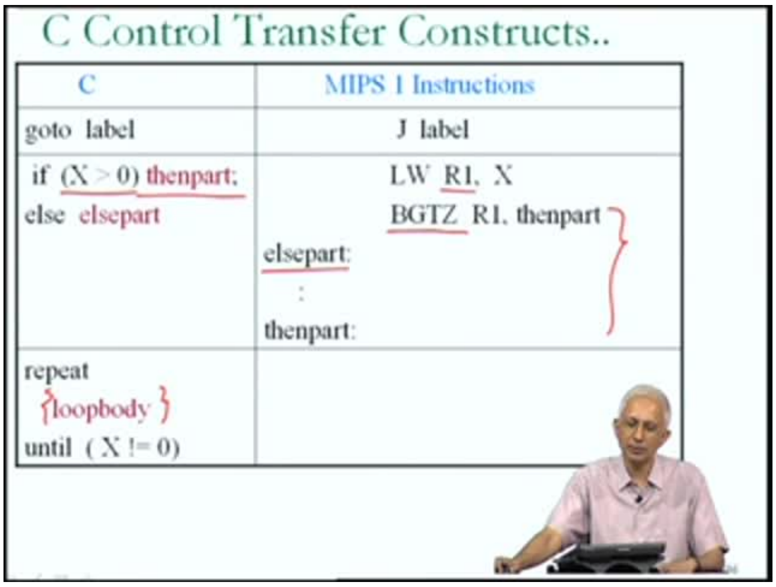
Then in may not be able to specify it appropriately using a 26 bit target address, and you could work out the kind of constraints which this places on your the distance between the go to statement in your c program and the label, and try to figure out if it is really that much of a constraint after all. Remember that this is 26 bits which is the actually a fair distance in terms of instructions even if each instruction is 4 bytes in size.

So, the go to a label will simply map into a jump to a label in almost all cases that you will encounter.

Next, let us consider the, if X greater than 0, the if then else construct. Now, I am using a generic kind of example here. In general, they will be some condition, I am using the example of the condition is X greater than 0, X is a variable. In the then part, they could be any number of statements; so, I am just indicating it by then part. In general, if there is no other one statement, you would have to enclose that by braces, and then, there is the else part. This could be once again any number of c statements.

So, very clearly in the MIPS 1 equivalent, I will specify to this example, I will have to start by loading the variable X into a register, because in the MIPS 1 branch instructions, actually I have to use a branch instruction, a conditional branch instruction to check this condition. The MIPS 1 branch instructions all take their operands out of registers. Therefore, I will have to start by loading the value of this variable X into a register, then I will have to compare it is value to zero, and that I can do using one of those six branch instructions, and appropriately, the target of the branch will be a label which will relate to the then part of the else part.

(Refer Slide Time: 25:23)



C Control Transfer Constructs..

C	MIPS 1 Instructions
goto label	J label

if (X > 0) thenpart;
else elsepart
	LW R1, X
	BGTZ R1, thenpart
	elsepart:
	:
	thenpart:

repeat
{loopbody}
until (X != 0)	

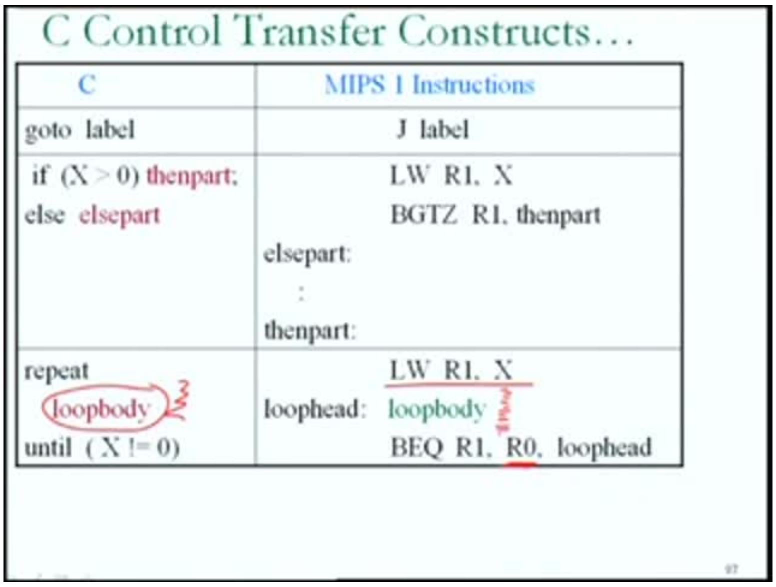
So, the way I am suggesting to work it out is as follows. I will start by getting the value of the variable X which is related to this condition into a register, let us say R1. Then in this particular case, I want to transfer control to the then part if X is greater than 0. I have a way to check if X is greater than 0 in the MIPS 1 instruction set using branch if greater than 0.

So, if R1 is greater than 0, then I transfer control to then part, and I have a label called then part later in my program. What if R1 is not greater than 0? Then I want to transfer control to else part. Therefore, in this example, I have put my else part as a label immediately following the conditional branch instruction, and therefore, this sequence will achieve what was required by the if then else.

Now, we need not be too concerned that then part appear before the else part in the c program, and in the MIPS program, the else part appears before the then part. That is just inconvenience if one is, that is not during that the inconvenience, because alternately one does not really read a typical c programmer, does not even read the machine language code.

Finally we will go to the repeat loop, and I am again using a somewhat generic example. Repeat any number of statements which are inside the repeat loop until this condition becomes true; in other words, until x comes not equal to 0.

(Refer Slide Time: 26:57



C Control Transfer Constructs…

C	MIPS I Instructions
goto label	J label

if (X > 0) thenpart;
else elsepart
	LW R1, X
	BGTZ R1, thenpart
	elsepart:
	:
	thenpart:

repeat
loopbody
until (X != 0)
	LW R1, X
	loophead: loopbody
	BEQ R1, R0, loophead

Once again X is the name of a location in memory. It is the name of a variable. Therefore, I will have to start by loading X into a register. Subsequently, I can use a conditional transfer instruction which will transfer control to the beginning of the loop body. Therefore, the way this could be mapped is I have an instruction to load X into a register. Then I have the loop body and I have associated with the loop body a label called loop head. So that at the end of the loop body I check if r one is equal to 0, R1 contains the variable X. Therefore, if R1 is equal to 0, that means that this condition is not true, and therefore, I have to go back to loop head, and therefore, this achieves the effect of this conditional transfer of this loop.

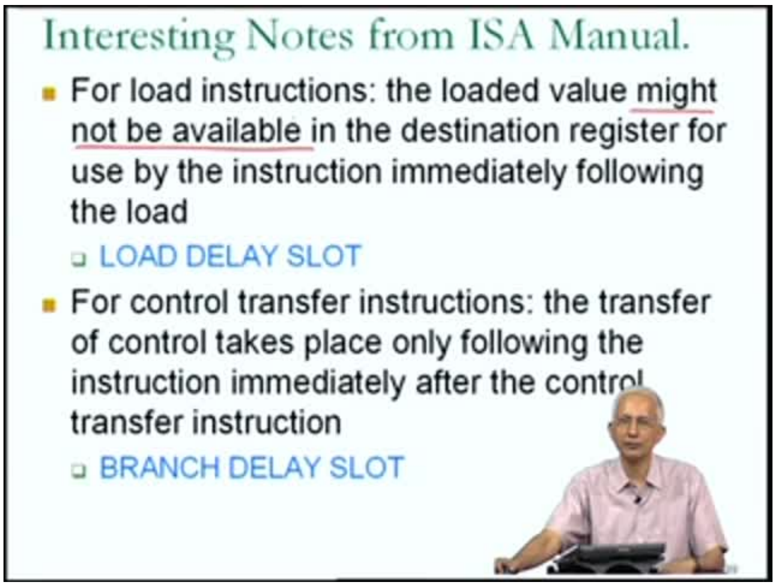
So, just note that even though the condition was not equal to 0, I ended up having to use a condition of equal to 0 because the not equal to 0 was the exit condition for the loop, and therefore, the continue condition for the loop was equal to 0.

And note here once again that I am comparing R1 with 0 by using register R0. I have to do this because I am using the BEQ instruction which has two operands. Now, just to make sure that notation here is clear. Inside my c statement, there was a loop body which contained any number of c statements.

Now, those c statements would get translated by the compiler into some collection of machine MIPS 1 instructions, and therefore, the number of instructions in the loop body will depend on the number of statements in the loop body over here and would depend how the compilation is done. But with this, we understand that essentially the importance c control transfer constructs that we are aware of can be achieved using the conditional and unconditional transfer instructions of the MIPS 1 instruction set.

So, with this, we have actually seen the data transfer instructions, the arithmetic and logical instructions in the control transfer instructions, and we are ready to actually look at real examples of MIPS 1 programs. Now, as it happens if I had made available to you a MIPS 1 instruction set architecture manual, then you would have read about the details of each instruction separately. Each of these instructions would have been mentioned on a separate page and on the page they would have given you all the information that I have gone through over here but some additional information as well, and some of the additional information that would have been included in the in the instruction set manual would have been some relevance, and therefore, as important aside, I am going to mention two kinds of important comments that one might see in an instructions set architecture manual relating to some of the instructions that we have come across.

(Refer Slide Time: 29:52)



Interesting Notes from ISA Manual.

- For load instructions: the loaded value might not be available in the destination register for use by the instruction immediately following the load
  - LOAD DELAY SLOT
- For control transfer instructions: the transfer of control takes place only following the instruction immediately after the control transfer instruction
  - BRANCH DELAY SLOT

So, I would label this slide as interesting notes from the instruction set architecture manual. Now, for some RISC Instruction sets, you find comments like this associated with load instructions in the instruction set architecture manual. The loaded value might not be available in the destination register for use by the instruction immediately following the load. This is obviously a comment which was included on the page of the load instruction as a warning to the programmer, because by default, the programmer might assume that if there is a load instruction that the value which is to be loaded into the destination register will become available as soon as the instruction has been executed, and therefore, the instruction immediately following the load instruction can definitely use the value in the destination register.

This is an explicit warning that, that is not the case, and it is present in apparently in the instruction set architecture manual of many MIPS 1 processes, which is why I have included it here. If at this warning is so important that it is given a name, it is given the name of load delay slot - indicating that the programmer must be aware that load instructions have the delay associated with them. They do not cross the load destination register to become updated as soon as you would imagine. There is also this element of uncertainty. If you look at this description, it says that the loaded value might not be available.

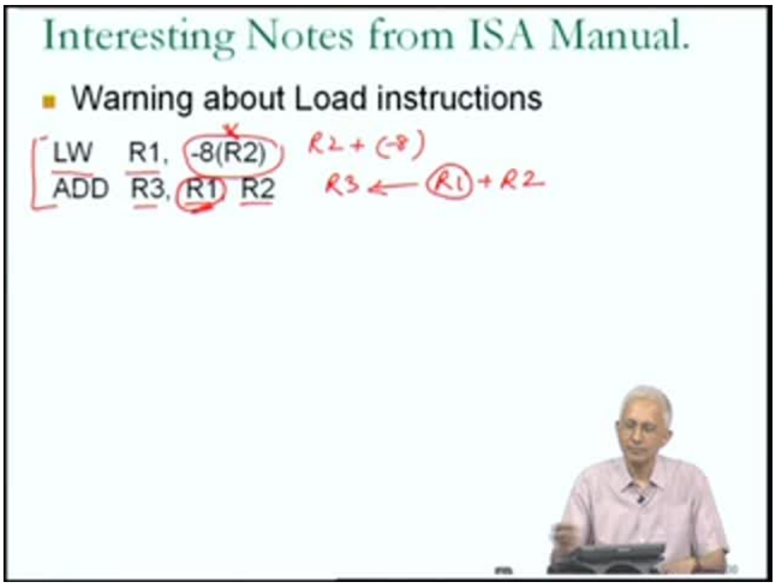
So, just think that in some implementations of the MIPS 1 Instruction set, the loaded value might be available, or for some programs when they execute for some reason, the MIPS the loaded value might be available, but in general, the programmer cannot assume that the loaded value will be available.

So, this is one kind of warning which you might see, and other kind of warning which you might see and in instruction set architecture manual for a RISC like architecture like the MIPS 1, relates to control transfer instructions, and might say something like this - the transfer of control takes place only following the instruction immediately after the control transfer instruction. Now, typically you would expect that the transfer of control will take place as part of executing the control transfer instruction, and that therefore, the instruction following, the, could control transfer instruction is clearly covered.

But here, we have explicit mention that the transfer of control takes place only after the instruction immediately after the control transfer instruction has been executed. Therefore, definitely a very important warning to a programmer particularly if he is program he or she is programming in this language, in the machine language, and once again, this is such an important warning that is available in on the page of the instruction set architecture manual, and it is given a name - the branch delay slot. I warning that control transfers are also delayed, they do not actually happen us fast as you would have thought.

So, from the programmers perspective, what is this mean? Let us just try to understand what I as a programmer should do if I see these kinds of warning in a instruction set architecture manual.

(Refer Slide Time: 32:57)


Interesting Notes from ISA Manual.

- Warning about Load instructions

```
LW   R1, -8(R2)     R2 + (-8)
ADD  R3, R1  R2     R3 <- R1 + R2
```

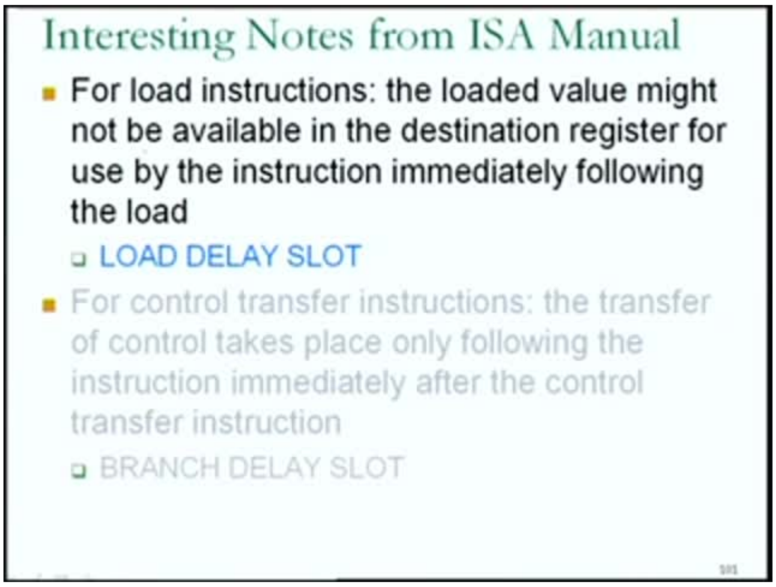
Let us start for with that warning about load instructions. So, the warning was something to do with when the destination will, when the loaded value will be available in the destination register.

So, let us suppose that I have would written a program and there is a load word instruction. Remember, this is a load word instruction, you saw this not too long back. A load instruction in general causes a value to be copied from main memory, from avoidable in main memory into a register. In general, this is an instruction which will do such a load of a word size or 32 bit size quantity. The destination is to a register R1, and the source is specified as a base-displacement addressing mode, base-displacement addressing mode. The address of the memory operand is calculated by taking the contains of register R2 and adding to them the immediate constant which in this example is minus 8.

So, this, this competition results in the address of the operand which can be, which can then be send to memory so that the data can be fetched. Now, if I have written a program which uses such an instruction. I may have also assume that, I mean typically I would have written such a this instruction in my program because I want to use the value of this variable out of the register R1. Therefore, it is not to uncommon that the next instruction in my program would be an instruction that uses R1. The next instruction in this particular example is an add instruction which adds the contents of R1 to the contents of

R2 and puts the result into the register R3. Remember that in our notation, the destination register is written for us, and then, the two source registers. So, the add instruction that I have shown you here is a register that uses the value of R1, and it is possible that my intend in writing a program containing these two instructions one after the other was that I want, let us say a value to be loaded from memory, let suppose that minus 8 R2 corresponds to my variable X.

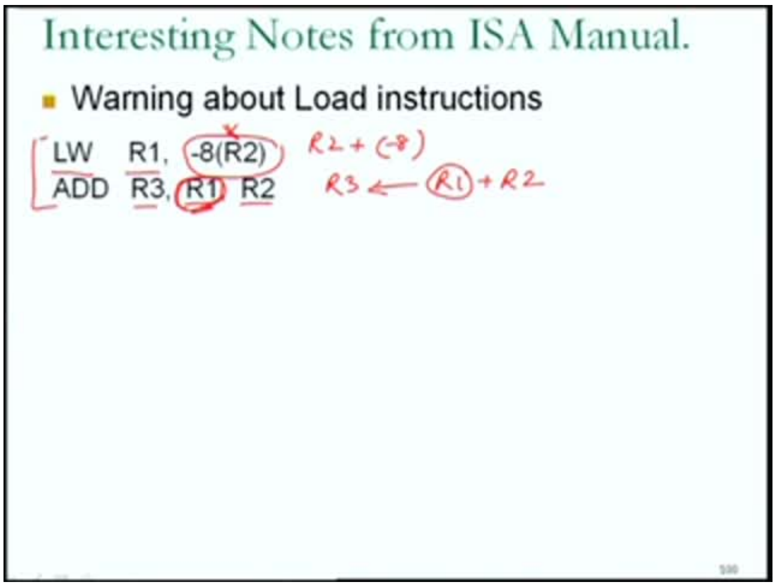
(Refer Slide Time: 35:02)



Interesting Notes from ISA Manual

- For load instructions: the loaded value might not be available in the destination register for use by the instruction immediately following the load
  - LOAD DELAY SLOT
- For control transfer instructions: the transfer of control takes place only following the instruction immediately after the control transfer instruction
  - BRANCH DELAY SLOT

(Refer Slide Time: 35:14)

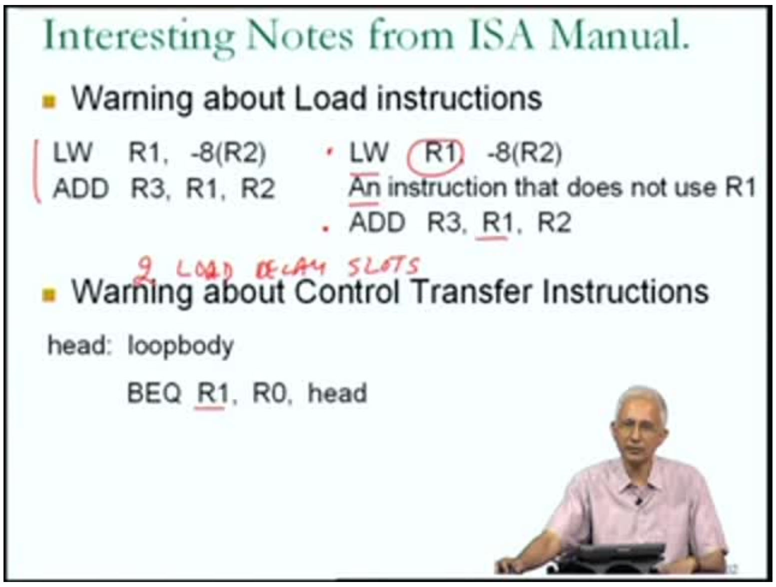


Interesting Notes from ISA Manual.

- Warning about Load instructions

```
LW   R1, -8(R2)      R2 + (-8)
ADD  R3, R1, R2      R3 ← R1 + R2
```

(Refer Slide Time: 35:27)



Interesting Notes from ISA Manual.

- Warning about Load instructions

```
LW   R1,  -8(R2)        • LW  R1  -8(R2)
ADD  R3,  R1,  R2         An instruction that does not use R1
                        • ADD  R3,  R1,  R2
```

9 LOAD DELAY SLOTS

- Warning about Control Transfer Instructions

```
head:  loopbody
       BEQ  R1,  R0,  head
```

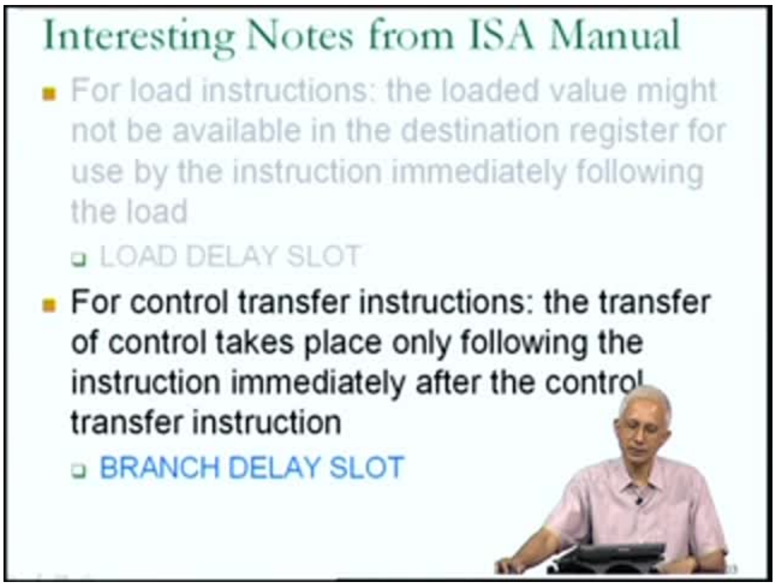
So, I wanted the value of X to get loaded into the register R1 and then I wanted to use that value in the addition. That might have been why I did the load followed by the add. Then, I must take into account the warning, which tells me that for a load instruction, the loaded value might not be available in the destination register for use by the instruction immediately following load. Therefore, a clearly the situation that we have here is dangerous. There is no guarantee that the add instruction will get the value that was loaded. Therefore, I as a programmer must or as a compiler writer must suggest for this warning, and the way that I could do it is by making sure that the load instruction and the instruction following it which uses the load, the loaded value are not consecutive got separated.

Therefore, example if I take a modified version of the same code segment where there is a same load instruction and the same add instruction, but I put between the 2 an additional instruction, such that the instruction in between does not use register R1 for it is own sake. Then I find out that the add instruction even under that the warning which had been provided in the instruction set architecture manual will safely receive the loaded value as loaded into R1 by the load instruction, and the program would gone as I had expected or as I had planned originally.

So, this is the kind of warning which one must very clearly take into account as the machine language program or as a compiler writer, and in this particular example, the situation was that the instruction set architecture manual warned me that there was this 1 LOAD DELAY SLOT. That load instruction and the instruction using the loaded value have to be separated essentially by at least one instruction but that could have been a most serious warning, for example, that they are is a need for 2 LOAD DELAY SLOTS. In which case, I would actually I have to write a program in such a form that there were at least two instructions between load instruction and the next instruction using the loaded value, and the warning would have been written in a form that would be understandable to us in this slide.

In a moving on from this to the next warning, the warning about control transfer instructions. Now this will relate to our program. I will have to give you an example where there is a use of a control transfer instruction. So, I will use the example which we have already seen the example of the repeat until. Remember that I implemented the repeat until using a label on the beginning of the loop body followed by a conditional branch instruction - the BEQ.

(Refer Slide Time: 37:42)

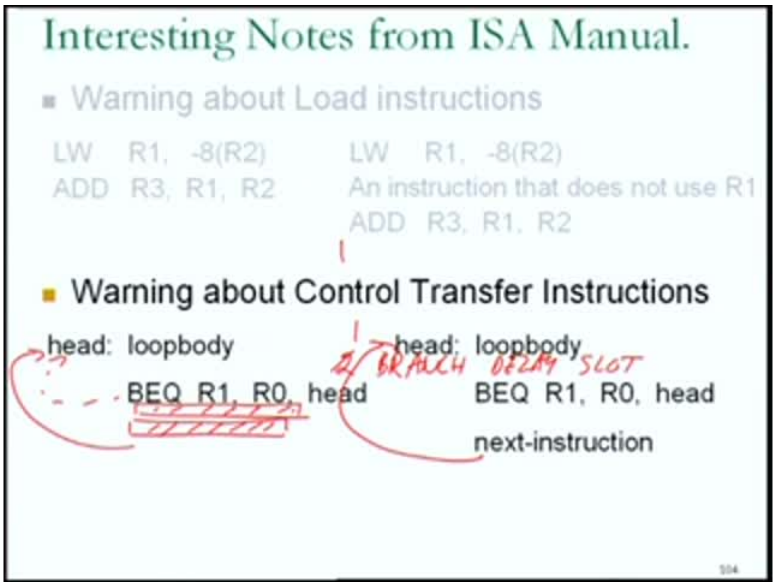


Interesting Notes from ISA Manual

- For load instructions: the loaded value might not be available in the destination register for use by the instruction immediately following the load
  - LOAD DELAY SLOT
- For control transfer instructions: the transfer of control takes place only following the instruction immediately after the control transfer instruction
  - BRANCH DELAY SLOT

The condition is related to register R1. Now, that, the situation that I have here relating to the warning, just, let us just remember what the warning was. The warning was for the conditional, for a control transfer instruction, they could transfer of control takes place only following the instruction immediately after the control transfer instruction.

(Refer Slide Time: 37:55)



Interesting Notes from ISA Manual.

- Warning about Load instructions

```
LW    R1,  -8(R2)        LW   R1,  -8(R2)
ADD   R3,  R1,  R2       An instruction that does not use R1
                         ADD  R3,  R1,  R2
```

- Warning about Control Transfer Instructions

```
head:  loopbody          head:  loopbody
       BEQ  R1, R0, head        BEQ  R1, R0, head
                                next-instruction
```

BRANCH DELAY SLOT
104

So, how does that relate to the code segment that we have over here? Now, according to that warning, the transfer of control to head, which is the control transfer that I am interested in, that is the correct implementation of the repeat until will not happen as a side effect, will not happen just after the branch instruction, but will happen after the next instruction in the program has been executed. In other words, the diagram that I should have in mind is something like this. Given the warning that we have just read from the MIPS 1 instruction set architecture manual.

In other words, if I had written the program assuming that immediately after the branch, you people instruction control is transfer to head, so, it is by the dotted arrow, then I would have misunderstood what was happening. In fact, the statement immediately is following the branch whatever it is, this also part of the repeat loop. It will be executed along with loop body regardless of what else may be present at the program. Therefore, thanks to our warning from the instruction set architecture manual. I must account for my repeat until implementation by possibly taking the last instruction from loop body if that is correct and putting it after the branch if equal instruction.

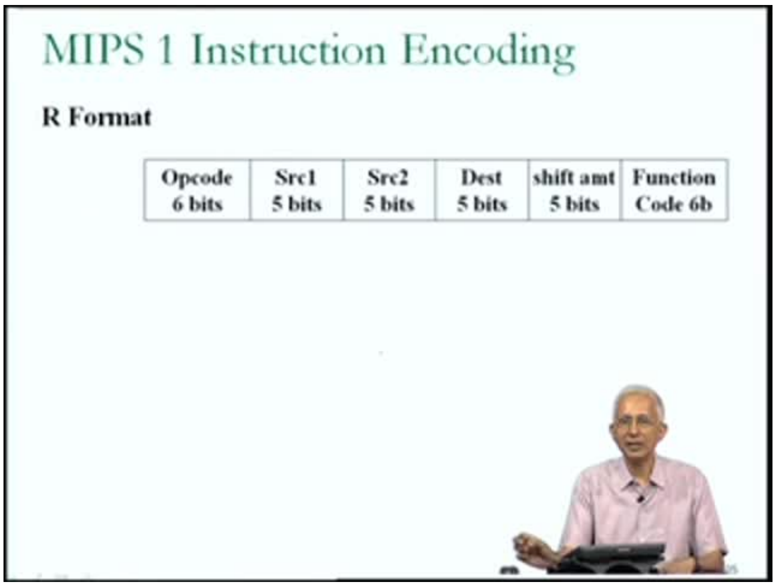
So, this is very important warning and it so important. That, in fact, it is refer to as the BRANCH DELAY SLOT even in the MIPS manual, and once again, it is conceivable that for other processors, you may see that there is a warning of 2 BRANCH DELAY

SLOTs which essentially tells you that two instructions after the control transfer instruction will be executed before the control transfer takes place.

So, these are clearly very important, and we will take both of these warnings into account. In fact, I think, for most of the examples, I will overcome that the assumption there is 1 BRANCH DELAY SLOT and 1 LOAD DELAY SLOT in writing the code.

So, that this, this idea of having to be aware of warnings of this kind will sink in, and so, the a correct implementation would have head, loop body branch instruction, and then, the next instruction with realization that the control transfer happens only after the next instruction.

(Refer Slide Time: 40:09)

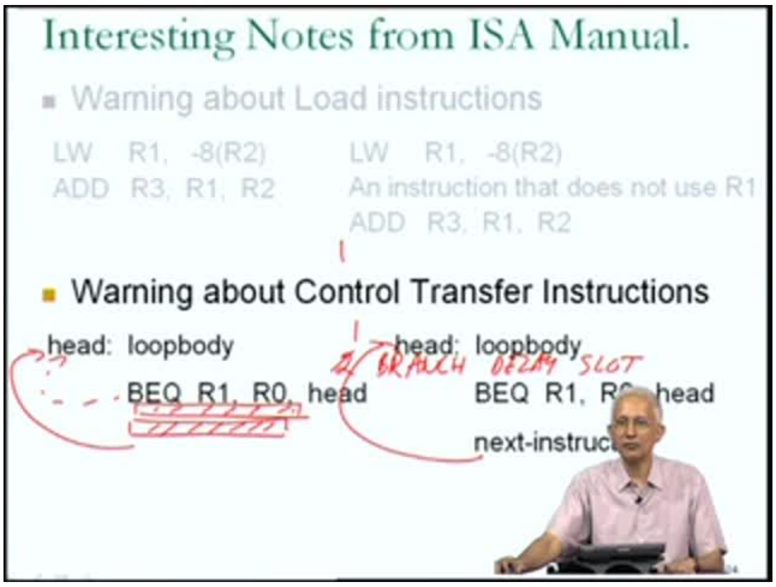


MIPS 1 Instruction Encoding

R Format

Opcode 6 bits	Src1 5 bits	Src2 5 bits	Dest 5 bits	shift amt 5 bits	Function Code 6b

(Refer Slide Time: 40:37)



Interesting Notes from ISA Manual.

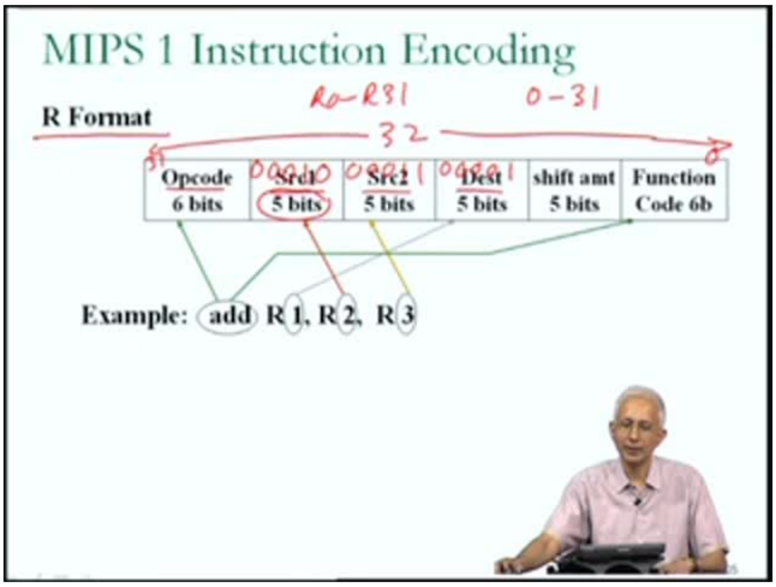
- Warning about Load instructions

```
LW    R1,  -8(R2)          LW    R1,  -8(R2)
ADD   R3,  R1,  R2         An instruction that does not use R1
                           ADD   R3,  R1,  R2
```

- Warning about Control Transfer Instructions

```
head:  loopbody                    head:  loopbody
       BEQ  R1,  R0,  head                BEQ  R1,  R0,  head
                                          next-instruc
```

(Refer Slide Time: 40:50)



MIPS 1 Instruction Encoding

R Format

Opcode 6 bits	Src1 5 bits	Src2 5 bits	Dest 5 bits	shift amt 5 bits	Function Code 6b

Example: add R 1, R 2, R 3

Now, that we have seen all the MIPS instructions. We can actually move to the last part of the instruction set architecture manual which is information about what each instruction looks like. Now, remember, the instructions of your program as generated by the compiler, end up in main memory your program executes out of main memory and if the instructions are to be in main memory, they are going to be represented in binary, and therefore, up to now I have been talking about instructions, such as, branch equal using very readable notation. In reality, that instruction is going to be in the form of some binary sequence - 32 bit binary sequence - what is the 32bit binary sequence look like? That is what we learn in the portion of the instruction set architecture manual which tells us about the encoding of instructions.

Now, the MIPS 1 instruction set has three different format for instructions - the first format or the first instruction format is what is call the R format and it describes the diagram that we have over here, describes what? A 32 bit instruction which is in the R format would look like. In other words, what the different bits from the least significant bit to the most significant bit are used for, and you notice this that they are many different fields in this instruction.

There is an op code field which we suspect must be some encoding of the operation - op code probably stands for operation code. There is a field label SRC 1 which is specification of the first source operand, second source operand, destination operand and various other fields.

Now, if you look at the size of each of these fields, we notice that the each of the source operand fields is a 5 bit field, what is it mean to have a 5 bit field? A 5 bit field means a five bit binary value can take on values between 0 and 31, and this is in consisting with our understanding that in MIPS 1 instructions, if there is a register direct operand, one has to specify the identity of that register.

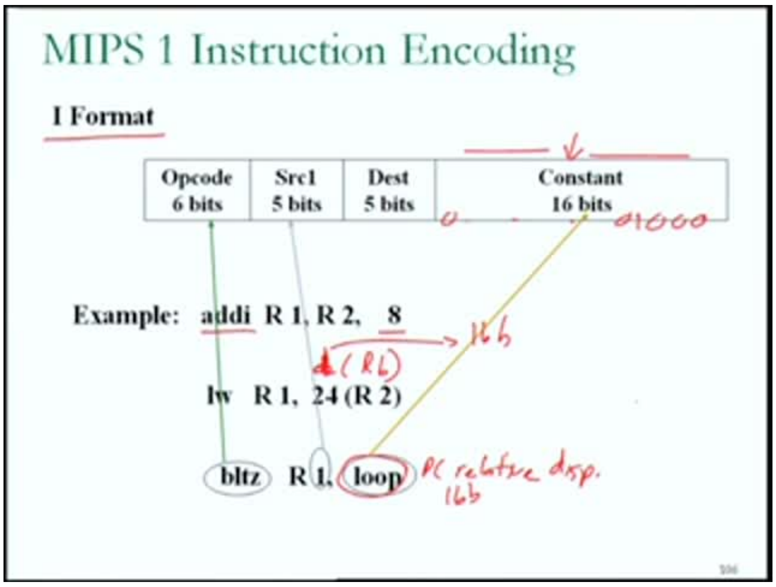
So, any register between R 0 and R31 can be specified using the appropriate 5 bit sequence. So, the fore most question you remind at this point will be what are the different instructions? Which will be encoded using the R format? Very clearly any of the arithmetic or logical instructions which does not use an immediate operand can be encoded using this format.

For example, consider I add R1 R2 R3, so, how will the, add the fact that this is an add operation, how will that be encoded? That would be encoded using the op code bits as well as possibly using the last 6 bits which are label this function code. To the, there are 12 bits available for encoding that the fact this is an add instruction. The fact that the destination operand is R1 and that the source operands are R2 and R3 would be encoded using the SRC 1 field, SRC 2 field and the destination field.

So, what exactly would one see in the SRC 1 field for this particular example? This is a 5 bit field. What I want to see over there is an indication that R2 is the first source operand, and therefore, I would see the value 2 in that five bit field. What would the value 2 look like in the 5 bit field? It would look like 0 0 0 1 0.

So, that is the exactly what I would expect to see in that 5 bit field. Similarly, for R1 over here I would expect to see 0 0 0 0 1, and for R3 I would expect to see 0 0 0 1 1. So, ultimately, when all the bit fields are filled up, we could understand exactly what the add R1 R2 R3 instruction looks like in 32 bits. You can find out what the different fields are, the different values, the exact values of the op code then the function code bits for the add instruction by looking up the appropriate section in the MIPS 1 Instruction set architecture manual if you needed to.

(Refer Slide Time: 44:28)



MIPS 1 Instruction Encoding

I Format

Opcode 6 bits	Src1 5 bits	Dest 5 bits	Constant 16 bits
			0 . . 01000

Example: addi R 1, R 2, 8 → 16b

d (R2)

lw R 1, 24 (R 2)

bltz R 1, loop PC relative dsp. 16b

So, basically the arithmetic and logical instructions which use only destination operands and source operands could be encoded using the R format. The second MIPS 1 Instruction set format is the I format or immediate format, because it has a 16 bit field inside the instruction. In addition, there is the 6 bit op code field, a 5 bit source operand field and a 5 bit destination operand field, and thus would imagine any instruction of the MIPS 1 instruction set which has a 16 bit displacement or immediate value can be encoded using this format.

In fact, will be encoded using this format, and they are several examples. For example, the arithmetic and logical instructions which have immediate operands. The immediate operand is the 16 bit field, requires the 16 bit field.

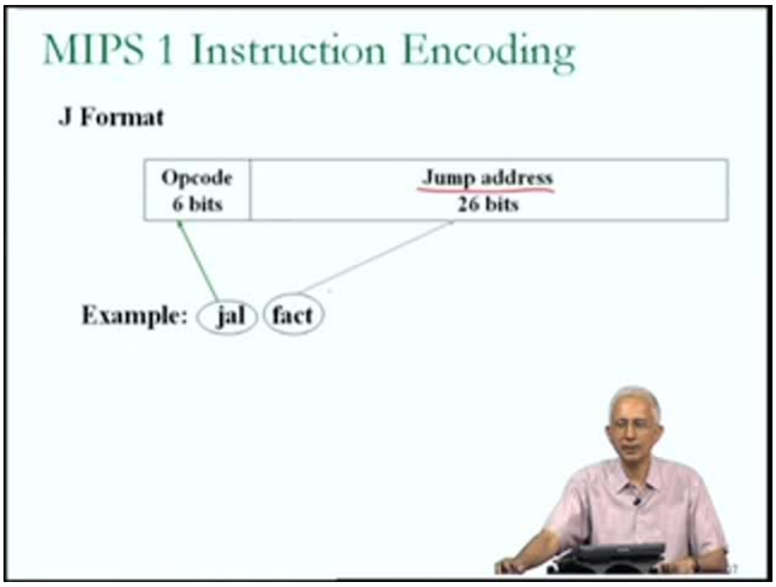
In this particular case, it requires only a 3 bit field, but in general, the immediate operand would be a signed 16 bit value. Therefore, the operation would be encoded in the op code field. The destination register would be encoded in the destination field. The first operand which is the source operand in register direct addressing would be encoded in the SRC 1 field, and the 16 bit immediate operand would be encoded in the constant field.

So, in this particular example, I would see the value 8 in 2's complement in the 16 bit field. In other words, I would see a lot of 0's followed by 0 0, I mean lot of 0's followed by 1 1 1 0 0 0 which is the 16 bit 2's complement value, a representation of 8. What other instructions would be encoded using the I format? Think about the load instructions or the store instructions. All of these instructions have their operand specified in base, I am sorry, base-displacement addressing mode. They have to specify a base register and destination and displacement is the 16 bit sign displacement.

So, once again, this I format could be used for this purpose. The fact that this is the load word could be encoded in the op code field, the destination can be encoded, I am sorry, the displacement can be encoded in the constant field. The base register can be encoded in the S1 SRC 1 field and the destination in the destination field.

Anything else? Yes, even the conditional branch instructions can be encoded using this format. Recall that for the case of the conditional branches, the target is specified as a PC relative displacement, and the size of the displacement is 16 bits. So, once again there is the situation where the 16 bit field can be used for this purpose. The op code would be used to indicate that this is branch if less than 0 field. The R1 would be encoded in the source 1 field, and the 16 bit PC relative displacement in the constant field.

(Refer Slide Time: 47:33)



MIPS 1 Instruction Encoding

J Format

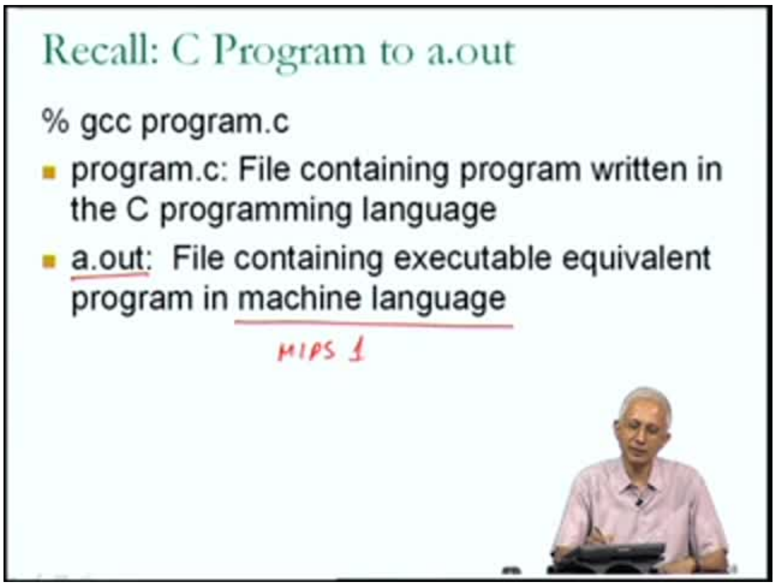
Opcode 6 bits	Jump address 26 bits

Example: jal fact

So, many different instructions actually find the encoding using the I format. There is 1 other format in the MIPS 1 instruction set and that is the J format. The J format has a large 26 bit field, and as you would suspect that this is necessary for the instructions which have a target 26 absolute addressing mode of which there are a few - there is the jump instruction and there is the jump and link instruction.

So, for example, if there was a jump and link instruction in your program, in our notation I will represent the target by a label, but in the instruction, the label would have to be encoded by it is actual twenty 6 bit address using the mechanism that we had seen in the slide about control transfer instructions.

(Refer Slide Time: 48:29)



Recall: C Program to a.out

% gcc program.c

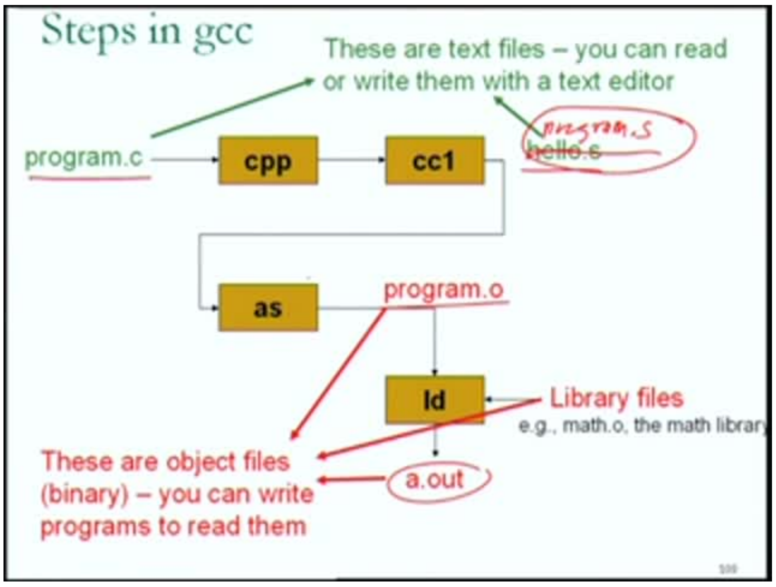
- program.c: File containing program written in the C programming language
- a.out: File containing executable equivalent program in machine language

MIPS 1

So, with this, it turns out that only these three formats are necessary, and all the instructions that we have seen and other instructions which we have not seen like the system call instruction etcetera can all be encoded using these three formats. With this, we are actually in a position. We have, we can move forward to start looking at programs and in order to do this, I just wanted to remind you a little bit about what happens when you write a c program and how it moves towards being a program in the machine language. I will quickly recall something from one of the earliest lectures. Remember that when you write a program c, you have to actually compile it using a step called the compilation step, and that the net result if doing this is that your c program file which was the file containing your program, which you typed into a file using a text editor or something like that was compile translated into an equivalent program and the default name of the output file of gcc is a dot out.

So, the output of gcc is the file called a dot out. It is the file containing and executable equivalent program in machine language. In other words, it could be it is a machine that you are working with is the MIPS 1 is a MIPS 1 machine, then this would be an equivalent program in the MIPS 1 machine language.

(Refer Slide Time: 49:39)



Steps in gcc
These are text files – you can read or write them with a text editor
program.c
cpp
cc1
program.s
~~hello.s~~
as
program.o
ld
Library files
e.g., math.o, the math library
a.out
These are object files (binary) – you can write programs to read them

Now, we saw that these translation happens to a series of steps. So, your program dot c goes through cpp, then you goes to cc1, and along the way a temporary file call hello dot, I am sorry, this would be program dot s was created which was used by one of the other steps, and other temporary file called program dot o was created which was merge with other files to generate an a dot out. We saw this in earlier lecture.

In this lecture what I would like to point out is, that is, there are these different files - program dot c, program dot s, program dot o, library file a dot out. All of which are potentially available to you to look at as I had mentioned earlier. Now, some of these files are going to be easy to look at. For example, you are, you can always look a program dot c, and you may be able to find a mechanism, by which, you can look a program dot s. Now, both program dot c and program dot s are text files, and you will be able to look at these files; you can read them or write them using a text editor. Whatever text editor you are used to using, whatever mechanism you use to could a the additional program dot c. On the other hand, the other files that I have mentioned on the slide - program dot o a dot out and the library files - are not text files, but they are object files; in other words, they are binary files. They contain information which you cannot easily edit using a text file that this not mean that you cannot write programs to open those files to read them or write them, but you would have to go through the effort of actually writing a program to open and read an a dot out file, our program dot o file or it have to use something else like and octal dump program to do the same thing.

So, some of these files that you encounter in the process of compilation or friendly and easy to read. You could therefore actually take a program find the way by which c c c would give you program dot s, and you could even edit - read and write and edit - the program dot s file and then pass it to the rest of the steps of compilation. So, with this thought in mind, we will move forward in the next lecture towards actually seeing what happens to a c program in terms of ending up in an a dot out file which contains MIPS 1 instructions.

Thank you.