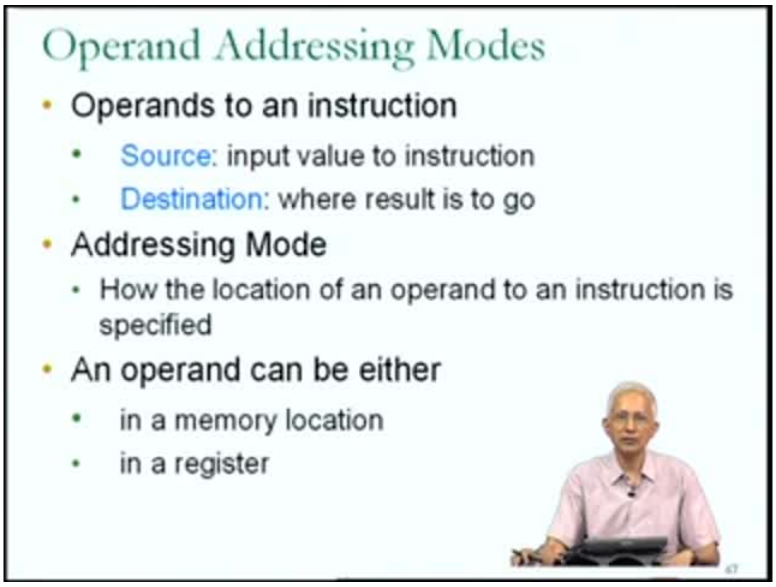
High Performance Computing

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

Module No. # 02
Lecture No. # 05

So, welcome to the 5th lecture of this, the course on High Performance Computing. We are currently trying to understand the little bit more about, what happens inside the processor, because this will be important to get an understanding of what happens, when our programs execute. And to understand about the individual instructions, that our program might get compiled into. We understood a little bit about what the different operations were in the previous lecture and also about the addressing modes, the mechanisms through which the instructions specify where the operands are coming from.

(Refer Slide Time: 00:49)

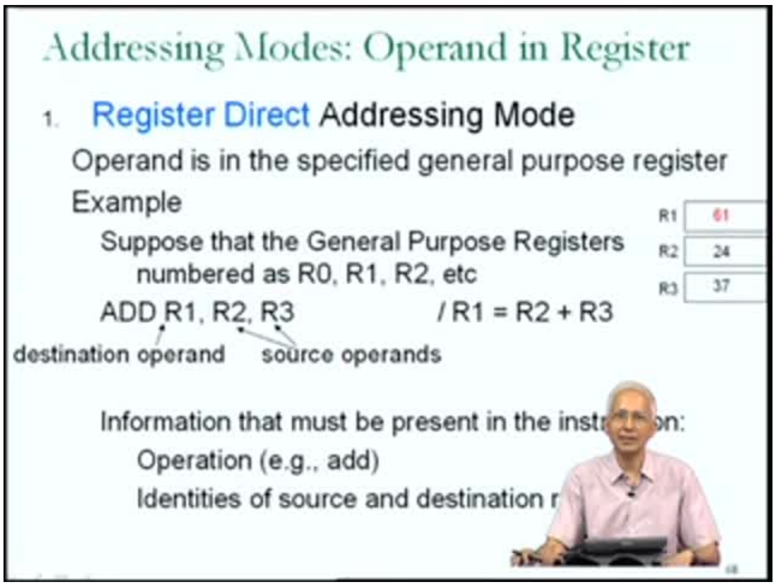


Operand Addressing Modes

- Operands to an instruction
  - Source: input value to instruction
  - Destination: where result is to go
- Addressing Mode
  - How the location of an operand to an instruction is specified
- An operand can be either
  - in a memory location
  - in a register

So, let me just move forward to what you are looking at last time, the operands to instruction are described, just either the source or destination depending on whether, they provide input values or are the destination where the result is to go; and the addressing modes are the mechanism is the basically encoding mechanism through which the location of the operand is indicated inside the instruction.

(Refer Slide Time: 01:13)



Addressing Modes: Operand in Register

1. Register Direct Addressing Mode

Operand is in the specified general purpose register

Example

Suppose that the General Purpose Registers numbered as R0, R1, R2, etc

ADD R1, R2, R3 / R1 = R2 + R3

destination operand source operands

R1	61
R2	24
R3	37

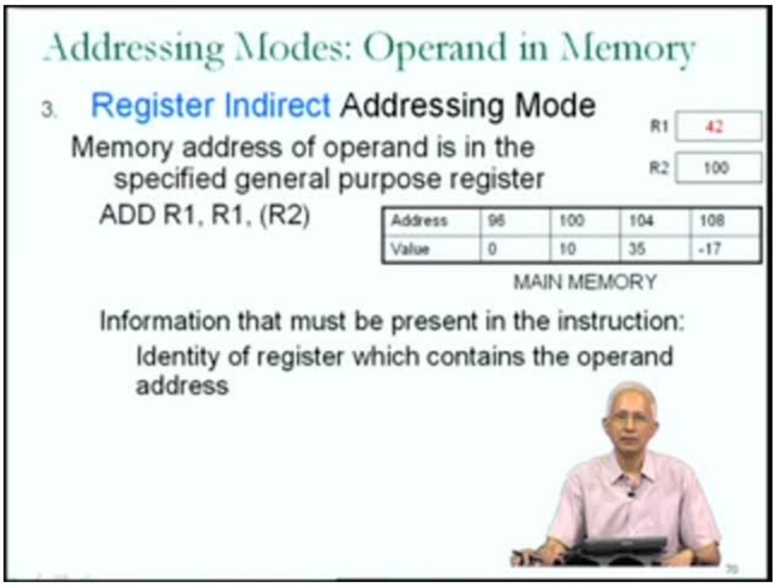
Information that must be present in the instr on:

Operation (e.g., add)

Identities of source and destination r

So, since your instruction operands can come either from registers or from main memory. There are special addressing modes for the operands coming from registers, such as register direct as well as the immediate addressing mode, all of which we saw in the previous lecture.

(Refer Slide Time: 01:30)



Addressing Modes: Operand in Memory

3. Register Indirect Addressing Mode

Memory address of operand is in the specified general purpose register

ADD R1, R1, (R2)

R1	42
R2	100

Address	96	100	104	108
Value	0	10	35	-17

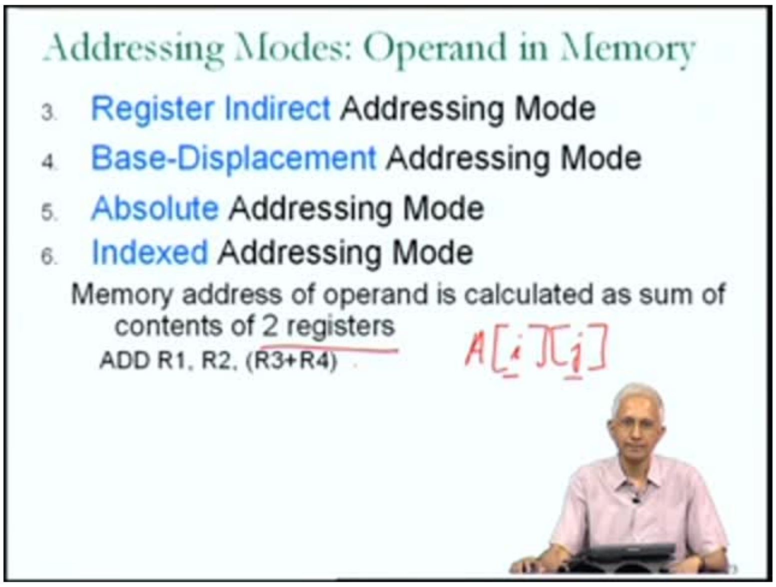
MAIN MEMORY

Information that must be present in the instruction:

Identity of register which contains the operand address

Now, we were in the process of looking at addressing modes for operands coming from memory and we saw the register indirect addressing mode in, which the address of the operand in memory is placed into a register and then the identity of that registers included in the instruction.

(Refer Slide Time: 01:51)



Addressing Modes: Operand in Memory

3. Register Indirect Addressing Mode
4. Base-Displacement Addressing Mode
5. Absolute Addressing Mode
6. Indexed Addressing Mode

Memory address of operand is calculated as sum of contents of 2 registers

ADD R1, R2, (R3+R4)

Ultimately, the operand is coming from memory. Then, we looked at the base-displacement addressing mode, which uses a computation a very small computation adding to in order to calculate the address of the operand, it adds the contents of a register to an a displacement, which is mentioned inside the instruction and then, the absolute addressing mode, where the address of the operand in memory is contained inside the instruction directly.

So, there are in fact, more addressing modes then these, it may come as a bit of a surprise, because all of these are just mechanisms to specify, where an operand is present in memory but, as I had (( )) to last time. For different kinds of C for high level language programming constructs, of a different kinds of instructions one of the other of these addressing mechanisms may be most appropriate, which is why people may use them in designing an instruction set. These are not decisions that a programmer makes, these are decisions about these addressing modes are made by the person designing a computer, the computer architect.

So, moving right along, we just mentioned that one of the some more important addressing modes is the indexed addressing modes. And the indexed addressing mode uses a slightly more complicated calculation to calculate the address of the operand in memory, and then the base-displacement addressing mode does.

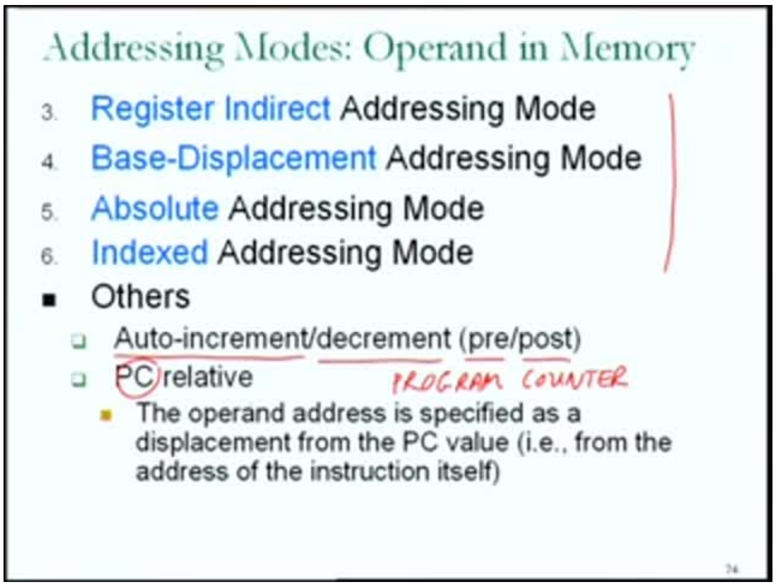
Because the addressing the address of the operand is specified through two registers and these contains these two registers when added gives you the address of the operand in memory, you turns of this addressing mode is very useful in for programs for example, which have references to arrays. For example if I have a C program in which there is a two-dimensional array called A.

That A reference to A will required two subscripts, subscript the row subscript and the column subscript and ultimately the array is going to be stored in memory; and our view of memory is linear, there are there is a memory addresses which are unsigned integers.

So, obviously, this array reference A of i j has to be converted into an address, using the starting address of the array and the actual values of i and j, which will require some computation and the indexed addressing mode is very useful for this kind of reference inside the program, which is why we often find it in instruction sets.

And so, these are the four main kinds of addressing modes for operands, which are in memory, but there are others. So, I will just mention some of them in passing.

(Refer Slide Time: 04:23)


Addressing Modes: Operand in Memory

3. Register Indirect Addressing Mode
4. Base-Displacement Addressing Mode
5. Absolute Addressing Mode
6. Indexed Addressing Mode

- Others
  - Auto-increment/decrement (pre/post)
  - PC relative
    - The operand address is specified as a displacement from the PC value (i.e., from the address of the instruction itself)

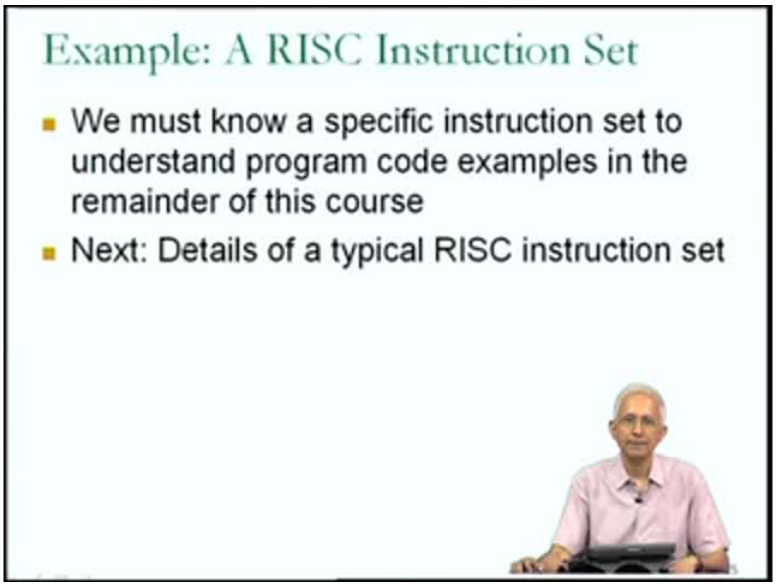
PROGRAM COUNTER
74

Can a one class of addressing mode, which one may find in slightly more complicated computer such as the SISS Computers, this addressing mode where they could be side effects of the operation of computing the address of the operand. For example, they could be addressing modes, where automatically the contents of the register, which is used in calculating the address of the operand, could get decremented or incremented, either before or after hence the pre or post, before or after the calculation of the address is done. And as you can see this is a significantly more complicated addressing mode than any of the others that we have seen. And that is why, I suggested that this might be kind of addressing mode, you would find in a CISC Architecture may be not in a RISC Architecture.

A slight more interest to us is the addressing mode called PC relative and we have seen that PC stands for the Program Counter, which is one of the special purpose registers in the CPU. The program counter at any given point in time contains the address of the instruction, which is currently being executed.

Therefore to say that, there is an addressing mode, which is PC relative means that, the address of the operand in memory is being specified as an offset from the address of the instruction, which is using that operand. And this may be convenient for certain kinds of instructions and so, the idea is that, the operand address is specified as the displacement from the PC value, in other words from the address of the instruction itself, the instruction, which is going to use this operand in it is computation. So, they have wide range of different kinds of addressing modes and as I already told last time, they go by these names and we will expect, if we look at in instruction set architecture manual, we would find the address addressing modes describe using these terms.

(Refer Slide Time: 06:19)



Example: A RISC Instruction Set

- We must know a specific instruction set to understand program code examples in the remainder of this course
- Next: Details of a typical RISC instruction set

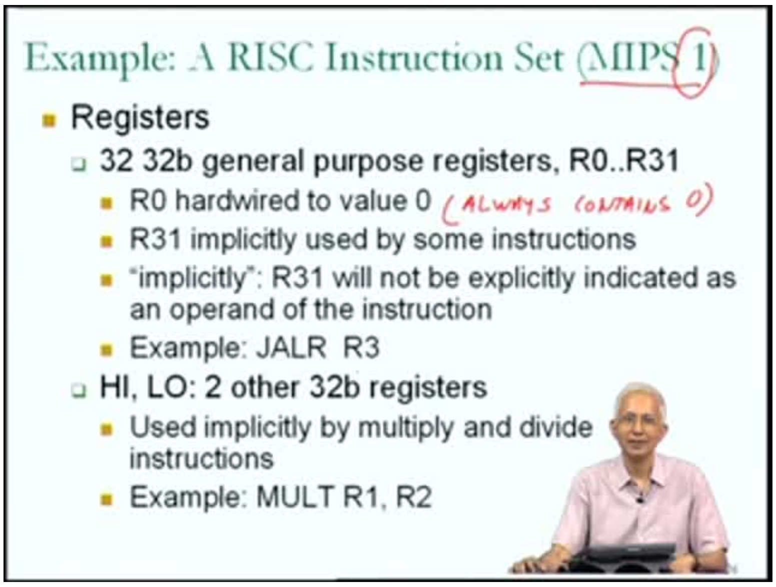
So, much for addressing modes, now that we have an idea about, what the operations the instructions, we can expect inside an instruction set and we have some idea about how the operands to those instruction might be specified using addressing modes. It is time for us to look at the specific instruction set and the example, which I have chosen to go through, is a simple RISC instruction set, reduced instruction set computer.

Properties of RISC processors are there each instruction does a single somewhat simple operation as oppose to the CISC C I S C instruction sets, where an individual instruction might do a fairly complicated operation. And so, we will be using the instruction set architecture that, I am going to discuss for the rest of this lecture, for all of the examples for in the rest of this course.

And therefore, I will be going through in some little bit of detail, because we will end up possibly writing code segments or understanding code segments written in this instruction set, to understand the other concepts of computer systems.

Right. So, the reason that, we are doing this is that we have to I will have to use examples in the rest of this course.

(Refer Slide Time: 07:29)



Example: A RISC Instruction Set (MIPS 1)

- Registers
  - 32 32b general purpose registers, R0..R31
    - R0 hardwired to value 0 (ALWAYS CONTAINS 0)
    - R31 implicitly used by some instructions
    - "implicitly": R31 will not be explicitly indicated as an operand of the instruction
    - Example: JALR R3
  - HI, LO: 2 other 32b registers
    - Used implicitly by multiply and divide instructions
    - Example: MULT R1, R2

Now, the particular RISC instruction set which, I am going to use as example is known as the MIPS 1 instruction set. Every instruction set will have to have a name, so that we can distinguish between the different instruction sets and the MIPS 1 instruction set was one of the first RISC instruction sets. MIPS was the name of the company and it still the name of a company today, a new condition of the same relative company and I am referring here to the first instruction set designed in, by that company subsequently, they have updated versions the MIPS 2 the MIPS 3 etcetera instruction sets.

We will start with the first and the simplest, which should be adequate for our purpose. Now, in order to give you an idea about the instructions, in this instruction set, as I said I will have to give you some idea about, what registers are available inside processors, which implement this instruction sets, so that is our first task. And to be brief the MIPS 1 instruction set as used that there are 32 32 bit general purpose registers, which we will refer to as a R0 through R31 and so the 32 of them, hence the names are R0 through R31.

Remember that, the general purpose registers are fast small PC is of memory inside the CPU, which are available for any purpose that the programmer or the compiler wants to use, typically they are used for storing frequently used pieces of data.

Now, in the MIPS 1 instruction set there are a few peculiarities, which we must be aware of, so let me just mention them.

The first and somewhat important peculiarity is that, even though there are 32 general purpose registers R0 through R31 not all of the them are actually available to the programmer to do, whatever he or she wants.

For example, R0 as we see in this next point is hardwired to 0 and what I mean by hardwired 0 is that, R0 always contains the value 0. There is no way to change that, if you wrote a program in the MIPS 1 language and try to change the value of R0, the instruction would not be illegal, but it would not any effect on R0, R0 would ultimately still contain the value 0.

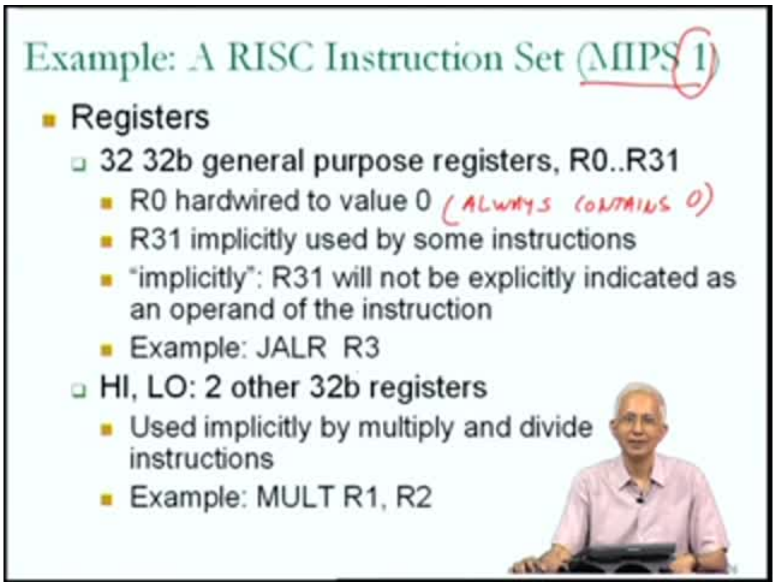
This is a very curious design decision, why it is the designers of the MIPS 1 instruction set, decide to have a register which always contain 0, as when we start writing code segments, you will realize this is actually very useful thing to have. Frequently, when you are writing a program, there will be a situation, where you want to compare something with 0.

For example, I may want to check, if the value of the variable x is equal to 0, 0 is therefore, a very frequently used constant by programs and having 0 readily available in a register, next speed of operation on 0 all though faster.

Otherwise, my program would have to have 0 as a variable, stored in a variable somewhere, over have 0 as a constant somewhere in memory, in order to do that comparison. So, this is a not that, uncommon in RISC instruction sets today, to have registers which contain fixed constants and cannot be changed in value.

In the case of the MIPS 1 R0 is one such register. So, we can use R0 as a source of the value 0 and we cannot change the value of R0. R31 the other end of the register spectrum also has a peculiar peculiarity and that is that, I have 31 is implicitly used by some instructions and what I mean by implicitly is that, if I looked at the instruction particular instruction, then I would not find the R31 mentioned as an operand inside the instruction.

(Refer Slide Time: 07:29)



Example: A RISC Instruction Set (MIPS 1)

- Registers
  - 32 32b general purpose registers, R0..R31
    - R0 hardwired to value 0 (ALWAYS CONTAINS 0)
    - R31 implicitly used by some instructions
    - "implicitly": R31 will not be explicitly indicated as an operand of the instruction
    - Example: JALR R3
  - HI, LO: 2 other 32b registers
    - Used implicitly by multiply and divide instructions
    - Example: MULT R1, R2

So, it is not explicitly used as an operand of the instruction. But it implicitly used in other words, without being stated inside the instruction and therefore, one has to be a little bit careful as a programmer or as a compiler writer, in using R31. Because, I may store, I may think that I can use R31 as the place to temporarily store, the value of my variable x from memory and then, I may call certain instructions to be executed and some of those instructions might be instructions, that implicitly use R31 as a destination in which, case the value that, I had stored in R31 will no longer be available.

So, as programmers, this is an important issue we must be aware that, R31 may not be a good register to use for storing values temporarily, unless exactly what the instructions the subsequent instructions do, if their instructions that implicitly use R31, we must take that into account.

Now, other than these peculiarities of the R0 and R31, the other 30 registers are actually freely available for the programmer or compiler to do whatever, he or she wants with. So, in other words R1 through R30 are real general purpose registers.

Now, there is one other point, which we must be aware of about the MIPS 1 registers before going ahead, that is before going ahead let me give you an example, of one of the instructions that implicitly uses R31 as an operand. And that is the instruction JALR and I give you here an example of using JALR, JALR with the operand R3.

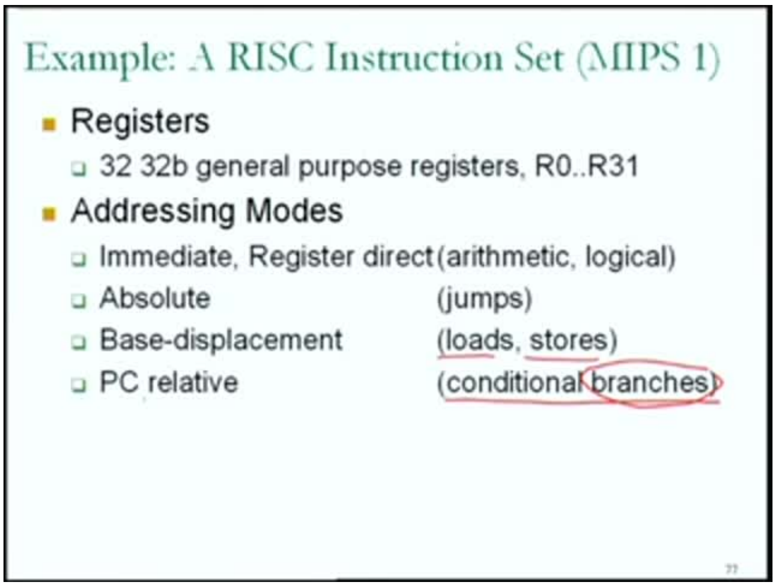
So, we will not (( )) R31 is not an explicit operand of this instruction, but this instruction thus modify R31, R31 is a destination of this instruction and we see what JALR is a little bit later.

Now, there is one other issue regarding general purpose registers in the MIPS1 instruction set, that we must be aware of and that is, that in addition to the 32 general purpose registers R0 through R31 there are in fact, two more general purpose registers, there are known as HI and LO, H I and L O and they 2 are 32 bit registers.

Now, the very fact that, they are not preferred to using a continuation of the name sequence that, was used for the other 32, in other words these are not called R32 and R33 they call by names HI and LO means that, they are actually used for certain specific purposes, though they will be available to the programmer in general as well.

So, the specific purpose that R that HI and LO are used for relates to the multiply and divide instructions. All the MIPS1 multiply and divide instructions implicitly use HI and LO and we will see this, when we talk about the multiply and divide instructions, to just sum up the, an example of multiply would be multiply R1 R2.

(Refer Slide Time: 14:34)


Example: A RISC Instruction Set (MIPS 1)

- Registers
  - 32 32b general purpose registers, R0..R31
- Addressing Modes
  - Immediate, Register direct (arithmetic, logical)
  - Absolute (jumps)
  - Base-displacement (loads, stores)
  - PC relative (conditional branches)

So, it has 2 operands R1 and R2 and it implicitly uses HI and LO as well. So, in brief then, the important general purpose registers of the MIPS 1 other 32 general purpose registers R0 through R31 of which, R0 is hardwired to 0 and cannot be modified R31 is implicitly used by some instructions.

In addition, there are the HI and LO registers, which are implicitly used by the multiply and divide instructions. Now, I do need to tell you, what addressing modes are used before, we look at the instructions, we must be aware of all these things.

So, the use of addressing modes, when the MIPS Instruction set is somewhat simple, the immediate and register direct addressing modes are used by the arithmetic and logical instructions, we had seen that there are four categories of instructions to expect. The arithmetic and logical instructions, the data transfer instructions, the control transfer instructions and the special instructions. So, we are learning now then, all of the arithmetic and logical instructions use the immediate and register direct addressing modes.

You look all that, the immediate and register direct addressing modes, from the previous lecture, if we learned these are the two addressing modes, which take their operands out of registers. The register direct addressing mode takes, operands out of general purpose registers and the immediate addressing mode take such operands out of the instruction register, because the operand is available inside the instruction.

So, this is going to mean, that all of the arithmetic and logical instructions will be able to execute without having to access operands from memory and therefore, they will be able to execute much faster than would have been the case, if they had memory operands.

So, once again this looks like a good design decision by the architects, arithmetic and logic will be as fast as possible. Now, the absolute addressing mode is used but, only by a jump instructions, you recall that, I am using the word jump to refer to those control transfer instructions, which are unconditional, you must know there is no condition.

So, sort of relative to the goto statement of C program and at the absolute addressing mode is not used anywhere else. Next, the base displacement addressing mode is used by load and stored instructions and you will call that, the base address base-displacement addressing mode is the one way, the operand is coming out of memory but, the address of the operand is calculated as the sum of a base register, the contains of a base register and a displacement, in other words the small signed value, which is contained inside the instruction.

So, after adding these two values, one gets the address of the operand in memory and this is used only for loads and stores. Remember that, a load instruction is an instruction, which can be used to copy a value from memory into a register. And the store instruction is an instruction, which can be used to do the reveres.

In other words to copy the value from a register into a memory location, so these are the two kinds of MIPS1 as we are going to see, the two kinds of MIPS1 instructions, which deal with operands out of main memory and both of them use to base-displacement addressing mode.

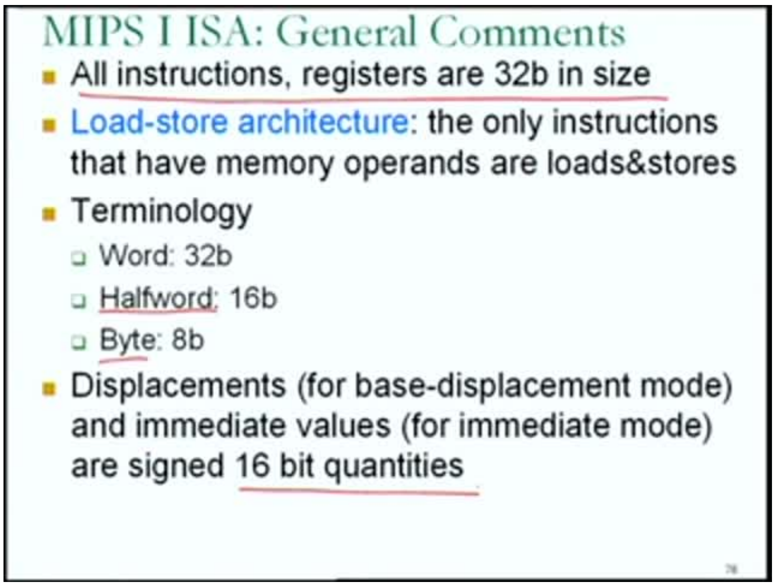
Finally, the PC relative displacement, which I spoke about a few minutes ago, is also used in MIPS 1 instruction set, in particular it is used by conditional branches and I had indicated in the previous lecture that, I will use the term conditional branch or just branch, to refer to a control transfer instruction, which has a condition.

So, if the condition is true, then the branch control transfer takes place and for such, MIPS 1 instructions the PC relative addressing mode is used, where the address of the operand, in other words the target of the branch is indicated as a displacement from the PC value over by, which we understand to that means, a displacement from the address of the branch instruction itself.

Now, these are the only addressing modes used in the MIPS 1 instruction set. There is no index addressing mode, none of the addressing modes that, I talked about I used, so things are little bit simpler than, they could have been and this will allows us to quickly understand the examples, that I see that, they are going to see. And if we had used the MIPS 2 MIPS 3 or MIPS 4, we would have found most sophisticated addressing modes used.

But now since, you understand what the addressing modes mean, once you have understand the MIPS 1 instruction set, you will quickly be able to understand MIPS 2 MIPS 3 MIPS 4 programs without much difficulty.

(Refer Slide Time: 19:28)



MIPS I ISA: General Comments

- All instructions, registers are 32b in size
- Load-store architecture: the only instructions that have memory operands are loads&stores
- Terminology
  - Word: 32b
  - Halfword: 16b
  - Byte: 8b
- Displacements (for base-displacement mode) and immediate values (for immediate mode) are signed 16 bit quantities

So, moving right along, I will start by making a few general comments on the MIPS 1 instruction set architecture, before we actually look at the specific instructions. Now, the first comment, which first thing is which we should be aware of, is that in the MIPS 1 instruction set, all of the instructions just like all of the registers are 32 bits in size.

So, every single instruction is 32 bits in size. If you have looked at or may be aware of CISC instruction sets, they are many CISC instruction sets, where because there are some instructions, which are very complicated and other instructions, which are very simple different instructions might be have different sizes.

You might have some instructions which have 8 bit in size, other instructions which are 32 bits in size and so on. So, that complexity is eliminated in the MIPS 1instruction set because, all instructions are exactly the same size, 32 bits.

In other characteristic of the MIPS 1 instruction set is that, while it is as I had mentioned a RISC instruction set, you may also see it describes as a load-store instruction set or a load-store architecture. And it is given this name, because the only instructions in this instruction set, that take operands out of memory or the load and store instructions.

All of the other instructions take operands out of registers and this is going to help to make the instruction execute faster, there is no need for those instructions to wait for an operand to come out of main memory at two orders of magnitude slower speed than the processor, as we had seen a few lectures ago.

So, the MIPS 1 instruction set is a RISC load store instruction set. Now, some of the terminology that is used in the MIPS 1 manuals will be of importance to us. So, let me just outline this. Now, in the MIPS 1 instruction set the word, w o r d, a word is used to refer to a 32 bit quantity.

In other words, the word size of the MIPS 1 instruction set is 32 bits. However, certain data values might be 16 bits in size, in other data values as we have seen might only be 8 bits in size. For this reason, they have a term for 16 bit quantity half word and they have a term for the 8 bit quantity, which is byte as you would have expected.

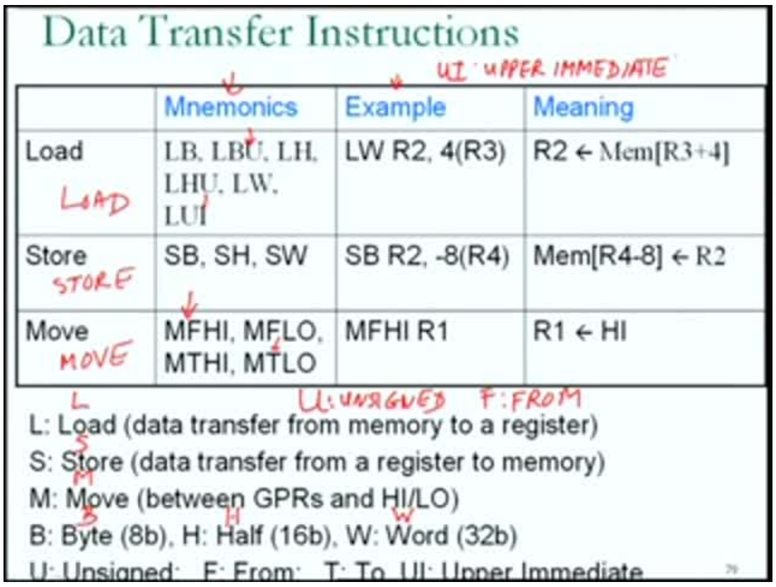
So, the three terms in the MIPS 1 instruction set relating to the size of something a word half word and byte corresponding to 32 bits 16 bits and 8 bits. And in the case of subsequent instruction sets in this series, such as the MIPS 3 MIPS 4 instruction sets. If they had data values, which were larger than 32 bits, they would have a word or the term to the corresponding to that, but that is not the problem for us in the MIPS 1 instruction set.

Now, another important thing to note in the case of the MIPS 1 instruction set is that, we saw that the base-displacement addressing mode is going to be used to specify the operands of load and store instructions and we also saw that, the arithmetic and logical instructions can have immediate operands.

In addition, that the load-store instructions have a displacement. Now, these values the displacements and the immediate values are going to be constants, that are present inside the instruction and therefore, we need to know how big those constants can be, they are all going to be signed integer constants, it turns out that they are all of size 16 bits.

And again, this may not be too much of a surprise, because I have mentioned earlier that all the other instructions are of size 32 bits. Therefore, it was clearly impossible to have a constant inside an instruction, which is of size 32 bits that would have taken up the entire instruction. And we now learned that, the size of the constants inside the instructions is that they could be 16, they are signed 16 bit quantities.

(Refer Slide Time: 23:53)



Data Transfer Instructions

UI : UPPER IMMEDIATE

	Mnemonics	Example	Meaning
Load LOAD	LB, LBU, LH, LHU, LW, LUI	LW R2, 4(R3)	R2 ← Mem[R3+4]
Store STORE	SB, SH, SW	SB R2, -8(R4)	Mem[R4-8] ← R2
Move MOVE	MFHI, MFLO, MTHI, MTLO	MFHI R1	R1 ← HI

U: UNSIGNED F: FROM

L: Load (data transfer from memory to a register)
S: Store (data transfer from a register to memory)
M: Move (between GPRs and HI/LO)
B: Byte (8b), H: Half (16b), W: Word (32b)
U: Unsigned; F: From; T: To, UI: Upper Immediate

Now, with this few general comments on the MIPS 1 instruction set, we can actually go ahead and look at the MIPS 1 instructions one by one. I am going to start by looking at the data transfer instructions, so in the tables that I am using to describe the instructions, I am describing a lot of instruction, that the whole category of data transfer instructions on this one slide. And you notice that, I have one column, where I have labeled it as mnemonic and in this column, I have included the names of a name, which could be used to describe that instruction.

Obviously, I since, we are talking about machine instructions, I should be showing you the bit pattern corresponding to that instruction, but if I show you the bit patterns, it would be difficult for us to understand program. So, I am going to raise the level of discussion a little bit by abbreviating the bit pattern into a mnemonic and easy to use, easy to remember a word as a replacement. Next I am showing you an example of using an instruction of that category and then a description about the meaning of that instruction is.

So, there are three categories of instructions in this table, there are the load instructions, there are the store instructions, then at the bottom I have something called the move instructions. And remember that, the load instructions are used to load values or to load a value from main memory into a general purpose register, store instructions are used to store a value from a general purpose register into main memory and move instruction is used for some other form of data movement.

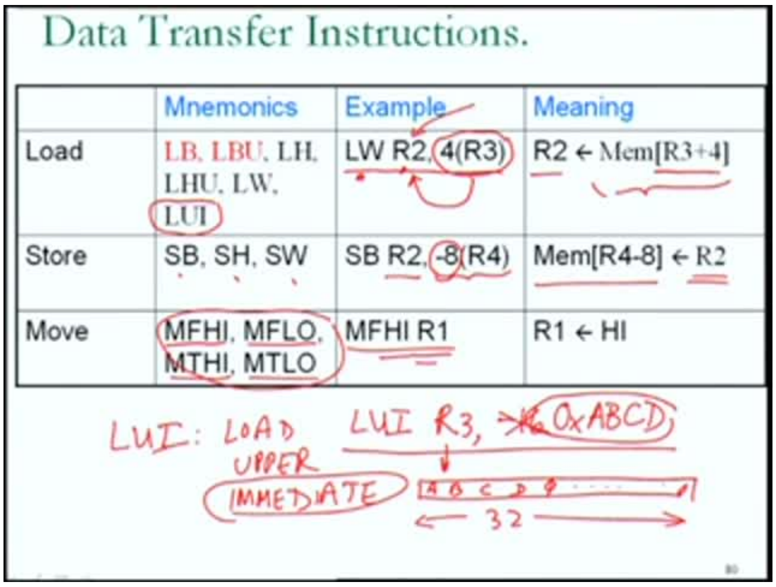
Now, before actually going into the instructions one by one, I am going to put up some information, which is generally applicable to understand the terminology, which is used in these mnemonics. So, basically whenever, there is an instruction that starts with the letter L in the in these tables, it is a load instruction, whenever there is an instruction that starts with the letter S it is a store instruction and instruction that starts with letter M is a move instruction.

Whenever, in this table we see a B it stands for movement of a byte, whenever we see an H it is stands for movement of a half word or a 16 bit piece of data, whenever we see a W it is stands for movement of a 32 bit value. And in addition to this, some of the other terminology, which is present in this table, you will notice in some places there, is U.

For example, over here and in general U stands for Unsigned in some cases there is an F an F stands for From, so move from T is also present, T is the opposite of from, T stands for To and then finally there is UI which stands for Upper Immediate.

So, with this terminology explanation L stands for Load, S for Store, M for Move, B H and w stands for Byte, Half Word and word, Unsigned from to an Upper Immediate. We can look at the instructions one by one. So, here we are.

(Refer Slide Time: 27:11)



Data Transfer Instructions.

	Mnemonics	Example	Meaning
Load	LB, LBU, LH, LHU, LW, LUI	LW R2, 4(R3)	R2 ← Mem[R3+4]
Store	SB, SH, SW	SB R2, -8(R4)	Mem[R4-8] ← R2
Move	MFHI, MFLO, MTHI, MTLO	MFHI R1	R1 ← HI

LUI: LOAD UPPER IMMEDIATE

LUI R3, 0xABCD

So, let us just try to understand what is up here. So, using the terminology that, we just saw LB will stand for a Load Byte, LBU which stands for Load Byte Unsigned, LH will stand for Load Half, LHU for Load Half Unsigned, LW for Load Word, LUI for Load Upper Immediate, SB for Store Byte, SH for Store Half word, SW for Store Word, MFHI will stand for move from HI and so on, MFLO move from LO move to HI and move to LO.

So, let us just look at the example over here. So, this is an instruction, which is load word R2 for R3. Now, that the load word instruction is going to, it is a load instruction. So, it is used to copy data from main memory into a register. The register is being specified in register direct addressing mode in this case. So, R2 is the destination operand of this load word instruction and I am following the convention of putting the destination operand first in this particular case.

Now, the rest of this instruction must let us know from, where a memory in other words the address of the memory location, which is going to be copied into register R2. And thus we just saw, the load and store instructions all use the base-displacement addressing mode to indicate the memory operands. And what we have over here is the base-displacement operand, the base register is specified as R3 and the displacement value is 4 or plus 4.

So, what this instruction means is as indicated in the meaning column, the address of the operand has to be calculated as the sum of the contains of R4 plus the contents of the value R3 plus the constant 4. So, once this calculation has been done that particular word of memory can be read and that will be the new value of the register R2 destination register.

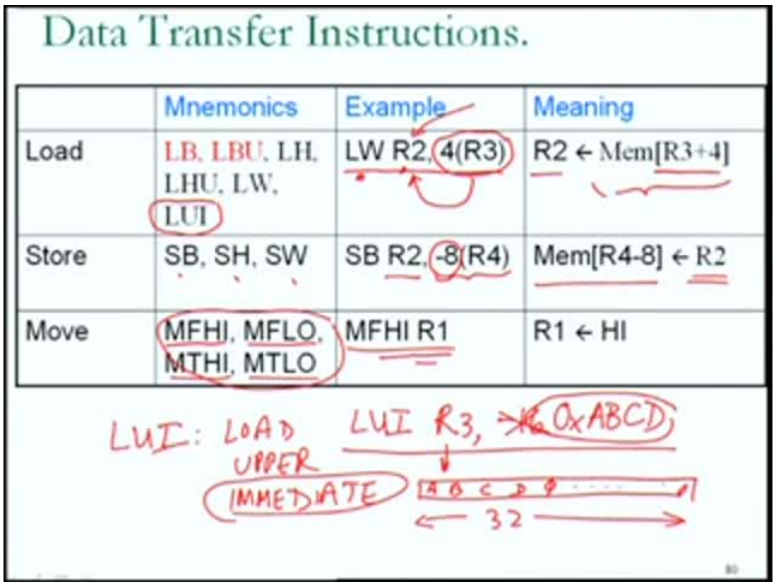
So, the contents of memory address as specified by 4R3, I copied into register R2 and that is the meaning of the instruction. Similarly, if there was a load half instruction, it would load into the register R2 not 32 bits, but 16 bits is half stands for half word and this is similarly, with load byte load byte will look like load word.

It will have the destination register and it will have a memory address specified, but instead of copying 32 bits, which is would have happen in the case of the load word instruction, the load byte instruction will cause only 8 bits to be copied.

I will make a comment about LUI a little later. Now similarly, they are the store instructions there store byte store half and store word and as seen in this example, the example is store byte R2 comma minus 8 R4.

So, the store instructions in general, I use to copy a value from a register into a memory location and as can be seen in this example, the register I want to copy the byte from is R2 and the destination address in memory; where the value is to be copied is specified by the base-displacement addressing mode minus 8R4, which is why the meaning is take a contents of the register R2 and copy them into memory location, which is calculated as contents of R4 minus the displacement value 8.

(Refer Slide Time: 27:11)



Data Transfer Instructions.

	Mnemonics	Example	Meaning
Load	LB, LBU, LH, LHU, LW, LUI	LW R2, 4(R3)	R2 ← Mem[R3+4]
Store	SB, SH, SW	SB R2, -8(R4)	Mem[R4-8] ← R2
Move	MFHI, MFLO, MTHI, MTLO	MFHI R1	R1 ← HI

LUI: LOAD UPPER IMMEDIATE
LUI R3, 0xABCD
A B C D 0
32

Next, we have the move instructions and these move instructions are used to transfer values from or to the HI and LO registers, in other words, I can copy a value from register R1 into the register HI using move from HI R1.

So, using this collection of four instructions, I can use the HI and LO instructions for any purpose that I want as a programmer, because I could have values which are in registers general purpose registers and then transfer them to HI or LO using these instructions. Now, finally, before moving ahead let me just make a comment about the LUI instruction as mentioned the LUI instruction stands for LOAD UPPER IMMEDIATE.

And this instruction an example of this instruction would be something like load upper immediate.

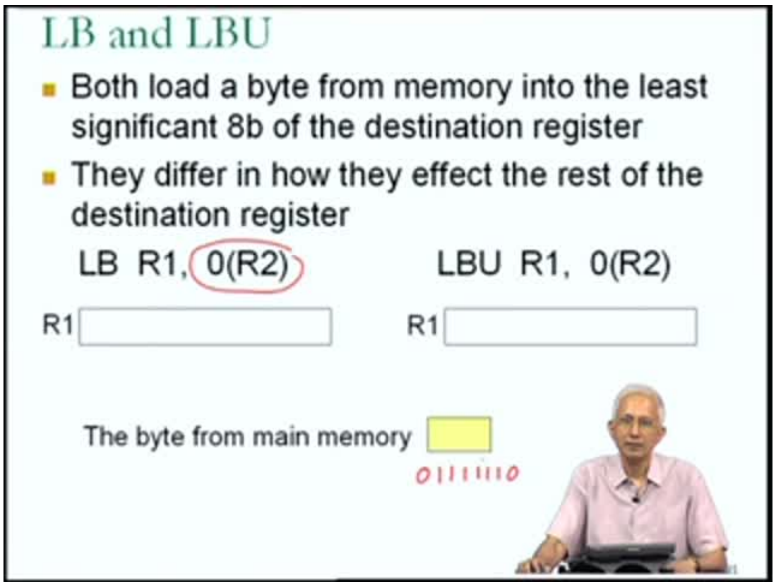
So, as with any load instruction, the load upper immediate instruction is one that, can be used to copy a value into a register. So, it must have a register as it is operand. The other operand is going to be an immediate value and in this case the immediate value is going to be a 16 bit immediate value.

For example, the immediate value might be minus 16 or let me just use a better example than that, when I am going to do is rather than specifying the example in decimal, as specified the example in hexadecimal, that is why you will get used to seeing hexadecimal values. And I will use a typical hexadecimal value, let us say hexadecimal ABCD, remember hexadecimal base 16 therefore; the digits can go beyond 9 to ABCDE and F. So, what is load upper immediate R3 ABCD do? Now, you have to remember that R3 is general purpose register of the MIPS 1 instruction set and it is therefore, a 32 bit register, it is of 32 bits in size.

What load upper immediate thus is, it causes it is 16 bit immediate operand to be loaded into the upper a most significant bits of the destination register. So, the ABCD gets loaded into the most significant 16 bits of R3 and the less significant bits get values of 0.

So, this is the very special kind of instruction, we see by this is useful again a little later in to lecture. So, with this we sort of understood the load store and move instructions you notice that, I have highlighted the LB and LBU instructions, because I have not actually indicated, what the difference between these 2 instructions are.

(Refer Slide Time: 33:51)



LB and LBU

- Both load a byte from memory into the least significant 8b of the destination register
- They differ in how they effect the rest of the destination register

LB R1, 0(R2)   LBU R1, 0(R2)

R1   R1

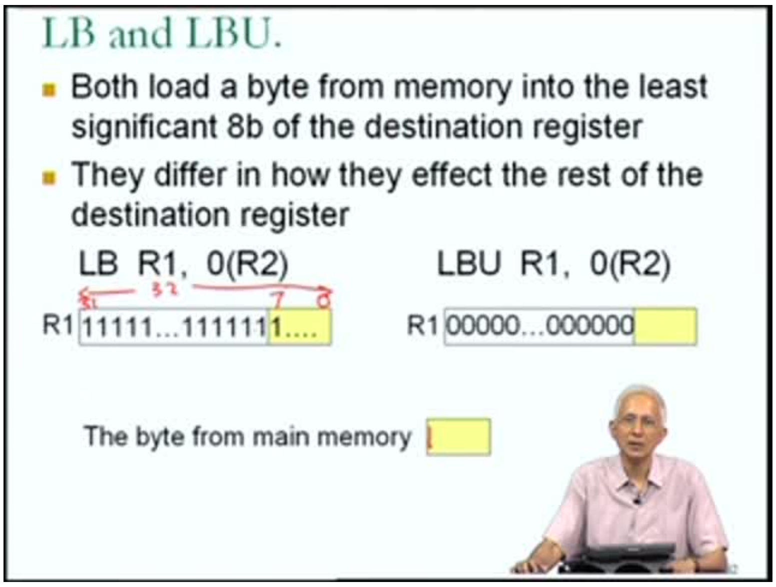
The byte from main memory

So, let me do that next. So, LB and LBU, LB as we understand is load byte, LBU is load byte unsigned and we need to clearly understand how they are different. So, that LB can be used to transfer a byte from memory into a given register, LBU can also be used to transfer a byte from memory into a given register, how do they differ? Both do the same functionality loading a byte from memory into the least significant 8 bits of the destination register, how do they differ.

They differ in how do they effect rest of the destination register, remember the each of the MIPS 1 registers, it is a 32 bit register, if you load a byte into the register that, tells you, how 8 of the bits of the register are modified what about the remaining 24 bits. Now, the 2 instructions differ and how they affect the remaining 24 bits. Let us, look at two examples, I have on the left I have load byte R1 0 R2, so this is some memory address in base-displacement and on the right, there is load byte unsigned R1 0 R2.

So, in both cases, they are destination register is R1 and let us just suppose that the byte, which is going to be copied from memory is the byte showed in yellow down here. But, when I say that I am showing you the byte in yellow just (( )) we could take on any value for example, the value could be 01111110, that is an 8 bit quantity.

(Refer Slide Time: 35:28)


LB and LBU.

- Both load a byte from memory into the least significant 8b of the destination register
- They differ in how they effect the rest of the destination register

LB R1, 0(R2)

R1 11111...1111111....

LBU R1, 0(R2)

R1 00000...000000

The byte from main memory

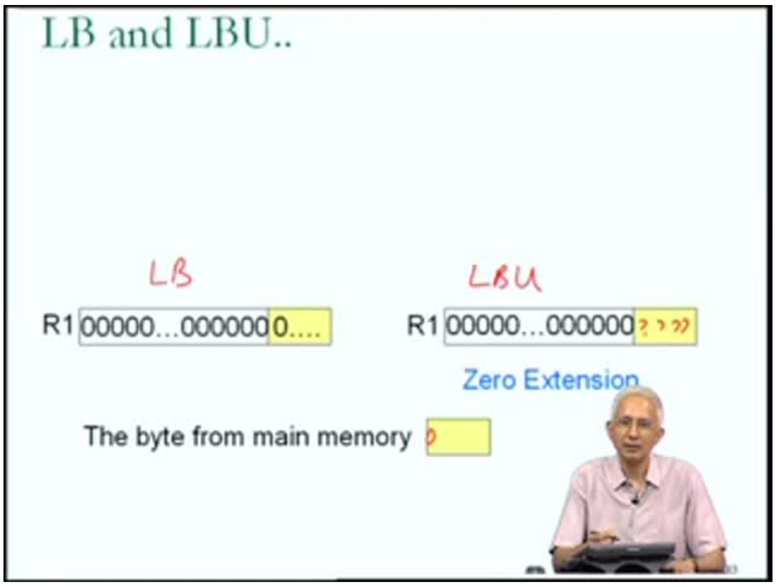
So, I am just showing you a yellow example of what the byte might be. So, for both L load byte and fore load byte unsigned. The effect of the instruction is that, the byte from memory goes into the least significant 16 bits, least significant 8 bits of the register, remember the register is of size 32 bits.

So, the byte from memory whatever, it is occupies the least significant bits in other words, the bits from 0 through 7, if I number the bits from right to left, the most significant bit is what I would call bit 31, to both of them do this they load the byte into the least significant byte.

The difference between the two is that, load byte unsigned always fills the rest of the register with 0’s, whereas, load byte sign, a load byte will fill the rest of the bits of the register in different ways, depending on what the most significant bit of the byte, which has been loaded from memory is.

For example, if the most significant bit of the byte, that has been loaded from memory is a one, I shown in the example of here, then what the load byte instruction will do is, it will fill the remaining 24 bits of R1 with once.

(Refer Slide Time: 36:41)



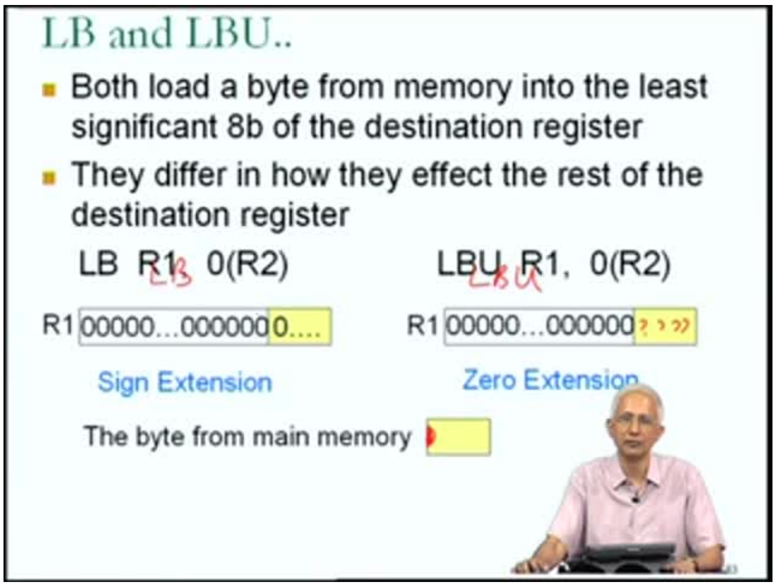
LB and LBU..

Similarly, if the most significant bit of the byte, that is loaded from memory is a 0, then this load byte instruction, this is what load byte does, this is what load byte unsigned does, what the load byte instruction we do is to fill the remaining bits of register R1 most significant bits of register R1 with 0.

If you think about this, what is actually happening is then the case of load byte unsigned regardless of what the bits that are loading from memory are, the remaining bits are always filled with 0 and this is what is known as 0 extensions.

So, the byte that has been loading from memory is extended into 32 bits using 0, whereas, in the case of the load byte instruction, the way that the remaining bits are filled or extended depends on what the most significant bit of the byte that was loaded is.

(Refer Slide Time: 37:48)



LB and LBU..

- Both load a byte from memory into the least significant 8b of the destination register
- They differ in how they effect the rest of the destination register

LB R1, 0(R2) LBU R1, 0(R2)

R1 00000...000000 0.... R1 00000...000000 ? ? ??

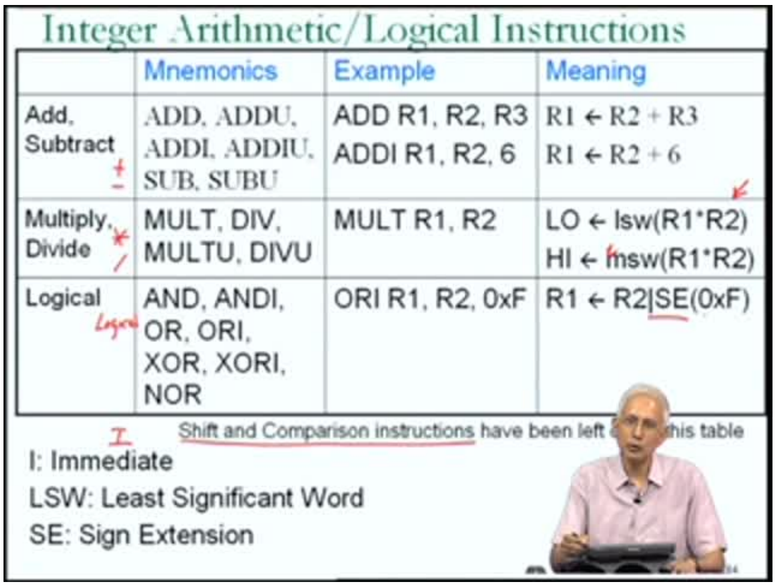
Sign Extension Zero Extension

The byte from main memory

Now, the most significant bit as we have seen, if we are talking about the two complement value, the most significant bit gives you some information about the sign of the value, if it was a 0, that the value if interpreted, thus a signed integer is the positive value whereas, if the most significant bit was 1, that the value would as a represents a negative value.

Therefore, the operation which is done by the load byte instruction is goes by the name sign extension, if the most significant bit of the byte is a 0 the rest of the register gets filled with 0. So, if most significant bit of the byte was a 1, the rest of the register get filled with 1. And that is the difference between the load byte and the load byte unsigned. So, they are, they have an common that the load a byte from memory into the least significant 8 bits of the destination register but, otherwise 1 does 0 extension the other does sign extension.

(Refer Slide Time: 38:25)



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

I

Shift and Comparison instructions have been left [illegible] this table

I: Immediate

LSW: Least Significant Word

SE: Sign Extension

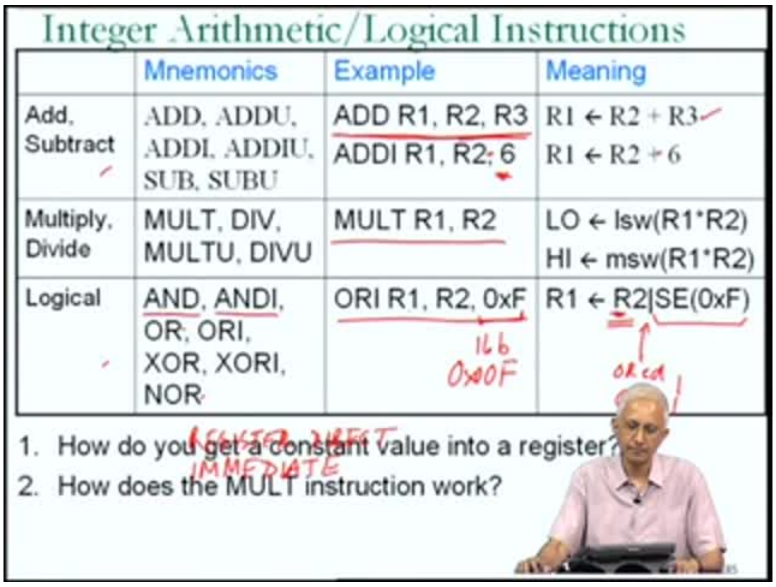
Now, moving right ahead, I am we are next look at the arithmetic and logical instructions of the MIPS 1 instruction set. And in this table I am showing a subset of the instructions as indicated in this comment over here at the bottom, I have left out two classes of instructions the shift instructions and some comparison instructions,

Since, if necessary I will explain them, when examples, which use them come up. So, in this particular table the top row has add and subtract instructions, the second row has our multiply and divide instructions and the third row has a collection a of logical instructions. And I am once again before going through the instructions one by one, let me just put down any terminology, which is used which might help in our reading of the table. First of all, in some of the, we have already seen that, I stands for Immediate as in the case of the load upper immediate instruction of the previous table, I stood for Immediate.

Now, in the terminology used in the meaning column over here, I have used LSW and MSW, which I would write like it to read us least significant word LSW and most significant word MSW, and finally, I have used SE to stand for sign extension.

In other words, the operation that, we just saw that the load byte instruction does in the hardware, sign extension is the idea of that a value can be extended to more bits by just replicating the sigh bit or the most significant bit further remaining bits that, have to be filled.

(Refer Slide Time: 40:20)



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?

So, with these three additional pieces of notation, we can understand the contents of this table. So, as we go through this table we see that there is an add instruction.

I should at this point mention that, the arithmetic instructions that I am showing you in this table, relate to integers and the MIPS instruction sets have separate instructions for real a floating point values and again as we come across the floating point operations in examples, I will introduce the floating point instructions.

For the movement just real interpret everything that, you see here as relating to integer values not to floating point values. So, this is an add instruction, which can be used to add integers ADD you, which we understand stands for add unsigned.

So, from which we infer that add must refer to adding of signed integers ADDI that, I stands for immediate, so this is ADD immediate, ADDIU which must be ADD immediate unsigned and then corresponding subtract instruction such as, subtract and subtract unsigned there is a multiply, there is a divide multiply U, which must be multiply unsigned, divide U divide unsigned.

Then, we are use logical operations and AND, AND immediate OR, OR immediate exclusive R exclusive R immediate NOR.

So, a representative collection of logical instructions, if you look at the table, I give you a few examples in the add and subtract line, then I have give you two examples, which use the add basically the signed integer addition instructions.

The add R1 R2 R3 and the ADD immediate R1 R2 R6 you will recall that, I mentioned that, all of the arithmetic and logical instructions take operands in only two addressing modes. The two addressing modes, which are used by the MIPS 1 instruction set for arithmetic and logical instructions are register direct and immediate. In the register direct addressing mode the operand comes out of a general purpose register and the immediate addressing mode, the operand is specified inside the instruction.

So, we see an example of an ADD instruction which takes all it is operands out of registers first, add R1 R2 R3, so it takes it is two source operands out of R2 and R3 does the addition using sign, signed integer adder and puts the results into the destination register R1 as indicated by the in the meaning on the right.

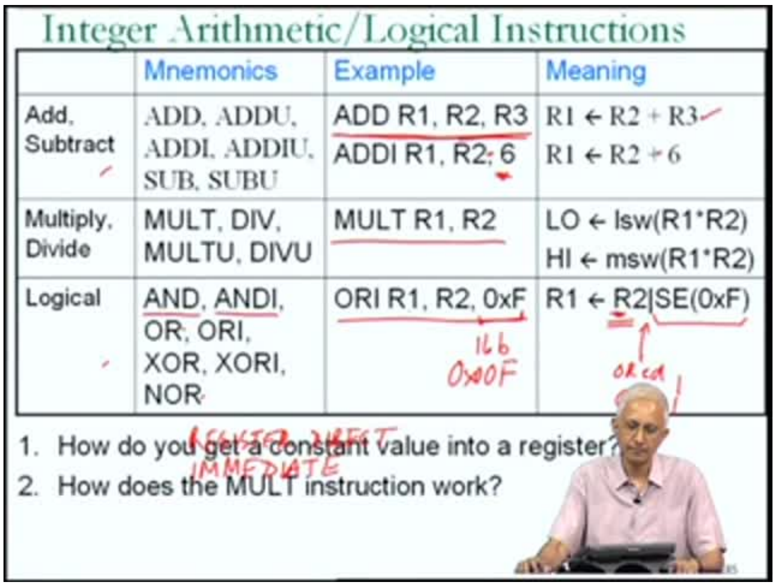
There is also an ADD immediate instruction, which can be used to add a value, which is specified inside the instruction such as, 6 in the second example to the contents of a register with result going into a another register.

So, these the ADD and ADD immediate instructions can be used to do addition on signed integer values, which give you some indication that the 6, that we have over here could, just is well have been a minus 6 and then this would have been adding minus 6 to R2. And that would, sort of motivate the reason for not having a subtract immediate instruction.

There is no need to have a subtract immediate instruction. Since, if I had been a subtract immediate instruction, it would have just had the positive value of 6 as the second operand and I can achieve the same effect by having a minus 6 is the second operand for the animated instruction.

So, the designers will avoid having instructions, which are not necessary in the sense that, if the functionality of subtract immediate can be met by the ADD immediate, then there is no need to include the subtract immediate instruction, if you just be a waste of the hardware, this the hardware would have to be a little bit more complicated, because there is one more instruction.

(Refer Slide Time: 40:20)



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 16 b 0x0F	R1 ← R2|SE(0xF) OR'ed

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

How does unsigned arithmetic differ from signed arithmetic? That is a most subtle question and we know that, there are signed integers and there are unsigned integers and that, the there are some situations, where we use one and some situations where we use the other and very clearly the arithmetic on the to must be mildly different.

For example, it is conceivable that over flow may be issues a mechanism through which they differ, but they are separate instructions meaning that there is actually separate hardware to deal with arithmetic on integers, which are signed, I supposed to arithmetic and integers, which are unsigned.

Now, as with the add and subtract instructions, the logical instructions have variance in which there are all operands coming out of registers and one operand coming out of a immediate value such as, shown in the OR immediate, example where the value in register R2 is OR'ed with an immediate value which in this case is the value F.

Now, the notation, which is used here tells you that since R2 is a 32 bit register, when the OR operation is done the second operand must also be a 32 bit value. Whereas, the operand which has been specified over here in actuality this is an 8 bit operand, but that all immediate is in the MIPS 1 instruction sets are 16 bits in size. So, this is actually a 16 bit value, we can think of this has being the same as 0 X 0 F, it is as many 0are as required to make into a 16 bit value. So, the question is how can you, when how do you generate a 16 bit value out of the, how do you generate a 32 bit value to do the OR out of the 16 bit immediate, which has been specified in the instruction and the answer is that is done using sign extension.

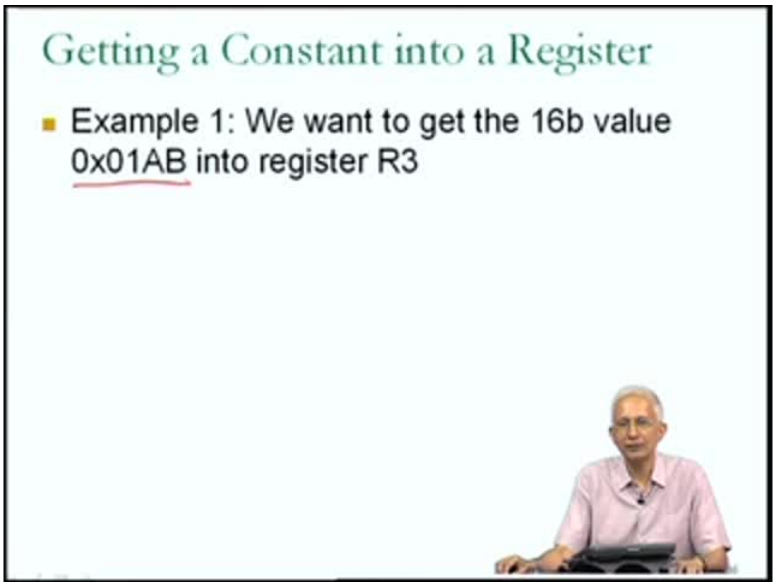
So, once a sign extension is done by the hardware, the hardware then does the OR using simple OR circuit and I am using the same operative for OR, that you are used to seeing from C, the C OR operator is that pipe which is what is used in this example.

And. So, with this we have a some out of an understanding about the add subtract and the logical operators, logical instructions, I have left the multiply and divide instructions for a separate slide, because they are somewhat curious.

The first thing, you will notice about the multiply and divide instructions is that, the multiply instruction seems to have any 2 operands, now we are used to seeing arithmetic instructions, where there are 2 operands, which are operated on using an arithmetic operator and the result goes into a destination.

So, there is a third operand the destination operand, the multiply and divide instructions are different, they have only 2 operands, this requires separate comment. But in fact, before coming to that, I wanted to say a few words about how to use these instructions for a specific objective. The objective which, I am going to target is let suppose, I want to get some constant value into a register, how do I do that, that is the question which we will address.

(Refer Slide Time: 47:24)



Getting a Constant into a Register

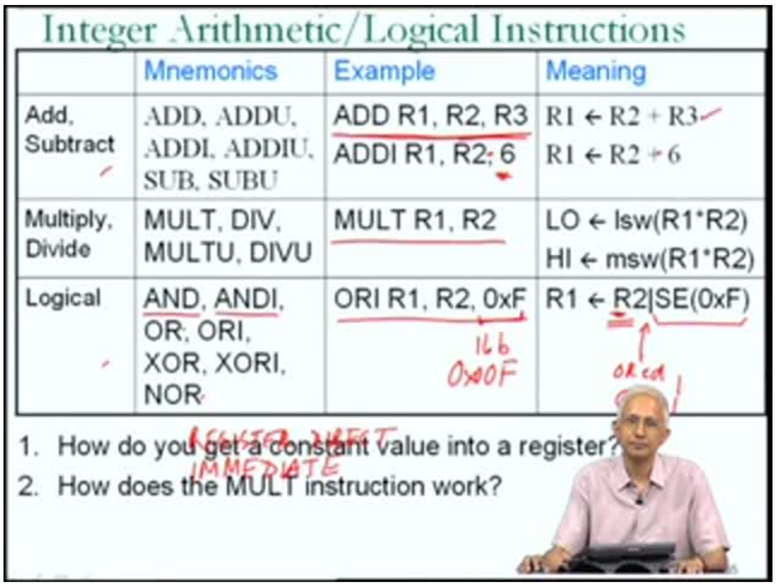
- Example 1: We want to get the 16b value 0x01AB into register R3

So, the question of getting a constant into a register using the MIPS 1 instruction set, we look at to 2 separate cases, first of all let suppose that, I want to get a 16 bit value, the specific 16 bit value X 01 AB into register R3.

So, I know that, I can do this, by putting the value one way of doing this is i could write a program where this value is declared as a constant and I can then make that, available in a memory location, I suspect that the compiler will make that available into a memory location and then, I could load that, constant into the register using a load half instruction.

That is one way that, this could be done, but the question, which I want to address here, address here is how could I write a sequence of instructions using which, I can get the 16 bit constant value, in other words, the constant value appears in one of the instructions into the register R3.

(Refer Slide Time: 48:26)



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

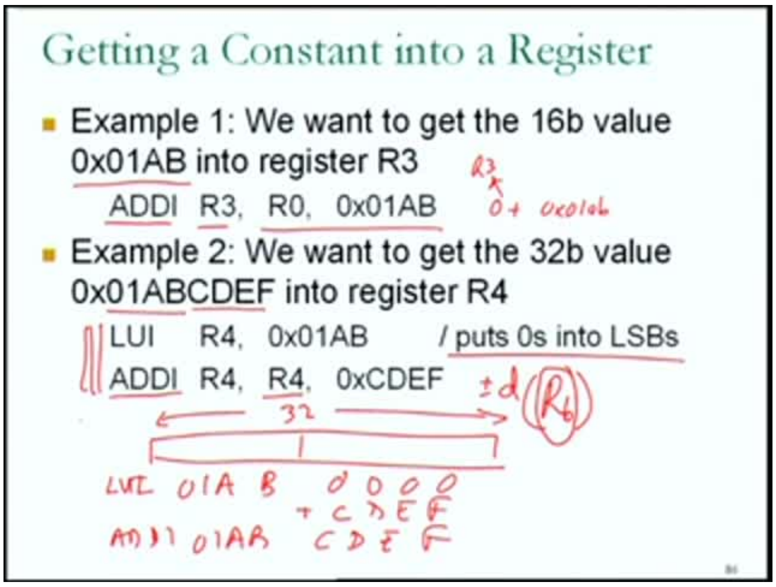
Multiply, Divide	MULT, DIV, MULTU, DIVU	MULT R1, R2	LO ← lsw(R1*R2)
			HI ← msw(R1*R2)

Logical	AND, ANDI, OR, ORI, XOR, XORI, NOR	ORI R1, R2, 0xF 16b 0x00F	R1 ← R2|SE(0xF) ORed

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

Now, in order to do this, I will have to use one of the arithmetic instructions from the previous table and there are many possible ways of doing this.

(Refer Slide Time: 48:39)



Getting a Constant into a Register

- Example 1: We want to get the 16b value 0x01AB into register R3

  ADDI R3, R0, 0x01AB

- Example 2: We want to get the 32b value 0x01ABCDEF into register R4

  LUI R4, 0x01AB / puts 0s into LSBs

  ADDI R4, R4, 0xCDEF

R3
0 + 0x01ab
±d
R4
32
LUI 01A B 0 0 0 0
+ C D E F
ADDI 01AB C D E F

I am going to suggest in, which I used the ADD immediate instruction, we call that in the ADD immediate instruction, I can specify a 16 bit immediate value. What I have here is the situation, where I want to deal with a 16 bit immediate value.

Therefore, I will use the ADD immediate instruction to add this 16 bit immediate value to something and then I will use R3 is the destination register. Thus since, my objective is to get that 16 bit immediate value into R3 without modification, I know that the thing that, I want to added to is. In fact, the value is 0 where can I get the value 0 from fortunately, I have the value 0 always available in register R0.

And therefore, with the single instruction ADD immediate R3 R0 0X01AB, I can achieve my objective, remember the R0 in this instruction ADD immediate R3 is the destination R0 and X01AB are my two source operands. R0 always contains the value 0 therefore, what this instruction is doing is adding 0 to the X value 01AB and putting this into register R3, in other words, it is putting this constant value into register R3 which is what I wanted to do.

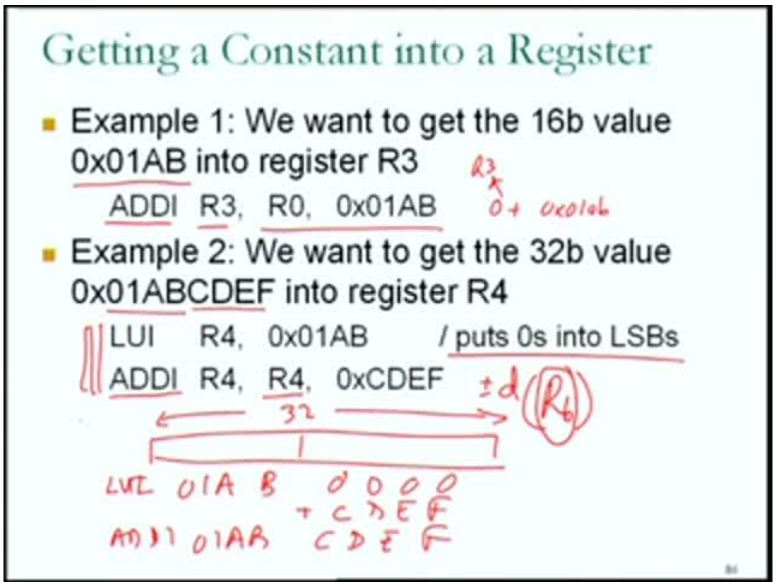
So, when you read the MIPS 1 programs, you will frequently find the ADD immediate instruction do being used for a purpose like this. If you look back at the table you notice that, I could just as easily have used the AND immediate or the OR immediate. Several other instructions which could have been used with inside the different ways, but this is possibly the easiest way of doing it, Now, that we see how I could get a 16 bit value into register using the MIPS instruction set, how do I get a 32 bit value into a register in the MIPS instruction set.

Now, 32 bit values, 32 bit constant values are always going to be a problem in the MIPS instruction set, because they can never be contained inside in instruction, the most information that, I the most concept information that I can contain in instruction is 16 bit signed value.

Therefore, very clearly in order to achieve this objective of example two, I will have to use a least two instructions. It will not be possible to do this with one instruction, unless I goes through the path of using load word to load this constant which, I know is available at some particular memory location. In short of that, I will have to use these two instructions and this is where the load upper immediate instruction is going to be useful.

If I use a load upper immediate to get the up most significant 16 bits, in other words, 01AB into the most significant 16 bits of R4 and I recall that, from my description of the load upper immediate instruction, it put 0 into the least significant bits.

(Refer Slide Time: 48:39)



Getting a Constant into a Register

- Example 1: We want to get the 16b value 0x01AB into register R3
  - ADDI R3, R0, 0x01AB
- Example 2: We want to get the 32b value 0x01ABCDEF into register R4
  - LUI R4, 0x01AB / puts 0s into LSBs
  - ADDI R4, R4, 0xCDEF

R3
0 + 0x01AB
± d (R4)
32
LUI 01A B 0 0 0 0
+ C D E F
ADDI 01AB C D E F

Then the effect of the first instruction is going to be that, I have 01AB in the most significant bits of register R4 and 0 in the less significant bits of R4.

So, all that remains to be done is to get CDEF into the least significant bits of R4, which I can do using an ADD instruction. So, I add two R4, which already contains in this most significant bits 01AB and so, the way to look at this is of my 32 bit register after executing the load immediate, I have 01AB in the most significant bits and 0 in the least significant bits; the ADD immediate instruction is adding CDEF to this. So, that the net effect is that after the ADD immediate instruction, I have 01ABCDEF in the register, now you notice that, I have used a ADD immediate which is a signed integer addition.

But, I need not worry about carry in this case because, when I add CDEF to 0 0 0 0, there is no possibility of carry and therefore, this two instruction sequence can be used to load of 32 bit value into a register.

Now, this is actually a very important sequence of instructions, we are going to see this sequence of instructions very often, because we frequently need to get a 32 bit value into a register. And let me, give you one very simple example with this may be necessary, you will be call that, all of the load and store instructions in the MIPS instruction set use the base-displacement addressing mode. And in the base-displacement addressing mode there is a base register to, which some displacement is added, the displacement is a signed quantity could be positive or negative.

Therefore, in order to specify the location of operand in memory using the base-displacement addressing mode, I must be able to get an address into a register, I as a programmer or as a compiler writer, I must have a mechanism to get an address into a register. An addresses are typically 32 bits and for the movement, we will assume that the addresses are at least 32bits in size. So, I have to have a mechanism by which, I can get a thirty two bit known value a constant into a register. And this is the mechanism that is going to be used in the code segments that, we write to do this which is why I talked about it this early.

Now, with this we have seen the MIPS1 data transfer instructions in other words, the loads stores and the moves. We have seen the MIPS1 arithmetic and logical instructions, we have looked at some very simple examples, after this we have to looked at the MIPS1, control transfer instructions and after that, we will be in a position, where the entire MIPS1 instruction set and can start looking at more and more complicated examples, it was our objective of actually figuring out how we can make our programs run faster.