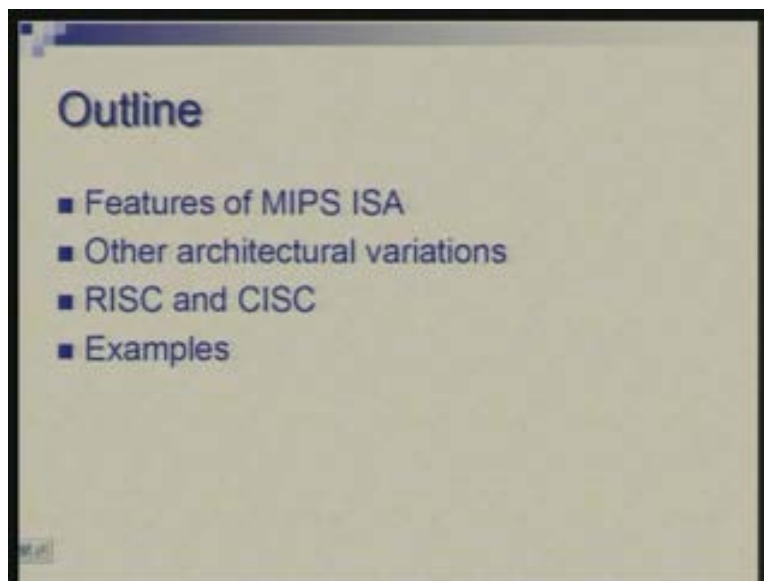


Computer Architecture
Prof. Anshul Kumar
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture - 7
Architecture Space

So far we have tried to understand instruction set architecture by taking a simple example of MIPS architecture and that we have used to look at the basic principles of how things work at a level close to the machine. We would now try to see what is beyond this simple architecture which we have discussed. We will look at what is called architectural space which means set of all architectures which are possible or which have been in existence. So we will look at main key features which we have learnt about MIPS architecture and see what variations are possible and towards the end we will look at some examples.

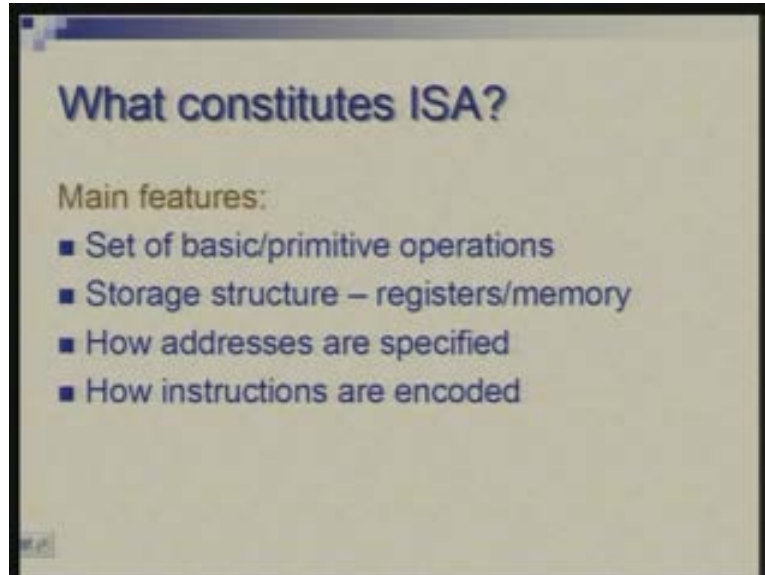
So first we will summarize the key points key characteristics of MIPS architecture at instruction set level, look at all the variations which are typically found in other machines, we will look at the these two terms RISC and CISC which stand for Reduced Instruction Set Computer and Complex Instruction Set Computer and these two represent to broadly different architectural styles and mention a few examples; I will elaborate on these examples in the next lecture but today I will just mention some of these.

(Refer Slide Time: 1:52)



So, when we are talking of this architecture versus other architecture what are the features we are talking of; what are the main things which characterize instruction set architecture.

(Refer Slide Time: 2:16)



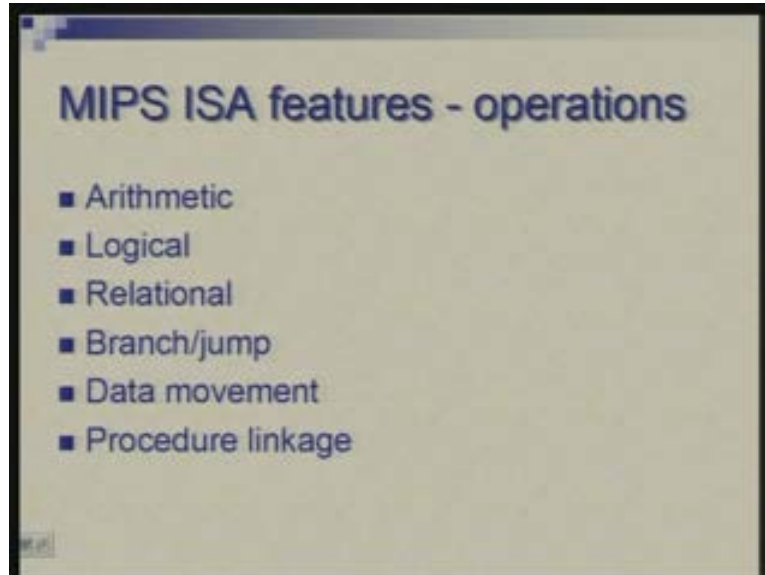
These things are predominantly what is the set of basic operations so it has to be a set of operations out of which you can build all the computation which you want to solve a different problem. So these are the basic operations or primitive operations and this is what could be different from one machine to another machine.

The second point is how storage structure is organized. Storage is in terms of registers which are part of the processor and a main memory which is outside the processor. So what is the number of registers; are there registers of different types or different sizes; is there some purpose which is specific to some registers that means the registers are general purpose registers or special purpose registers and what is the memory address space, what is the range of addresses whether it is accessed by bytes or word or one could mix it to.

Then; in an instruction how do you specify the operands, how do you specify their addresses, how many of these you have; whether you have two of two operands, three operands, less operands or more operands and so on.

And finally how do you represent instructions in terms of binary patterns; how the word which contains an instruction is divided into fields and what is the meaning of each different field. Now, in terms of these predominant features these main points how do we characterize MIPS architecture which we have discussed so far. What we have discussed is not complete MIPS architecture it is let us say the basic instructions but that still gives you flavor of what this architecture is about and we should be able to characterize this in terms of these features.

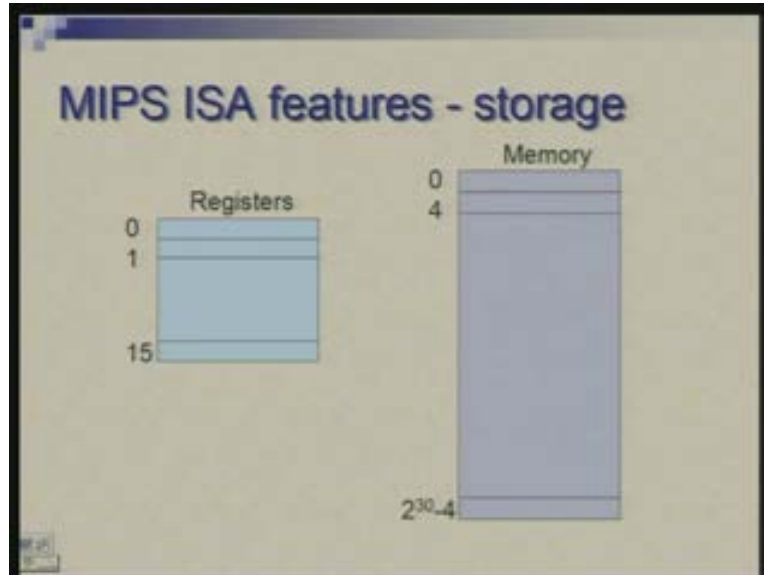
(Refer Slide Time: 4:20)



So first let us look at the primitive operations which we have studied. They are arithmetic operations; we have talked primarily about add and subtract but there are also common operation multiply and divide; logical operations AND OR, exclusive OR we have not talked of these again in detail but a small set of these operations exist; relational operation we have branch an equal branch **an equal** which does the comparison then we have the slt operation which compares for less than and the other operations have to be built using these so this is a small set of these operations available; then you have branch and jump where flow of control is changed.

Typically you go through sequentially but at some point under some condition or unconditionally you need to go to another point another instruction to actually represent the logic of the computation or to do procedure linkage you need to call a procedure or return from a procedure there are instructions for those; then there are instructions for movement of data bringing data from memory to registers or registers to memory or movement within the registers. So these are all the operations we have seen so far. There are also operations which work on non-integers that is real numbers and so on those we are going to see. But broadly this list actually indicates the class of instructions which we have.

(Refer Slide Time: 6:13)



The storage structure is shown here. You have sixteen registers. as far as instructions are concerned almost all instructions can use any register with equal ease so you have register numbered from 0 to fifteen sorry 0 to 32 which requires a 5-bit field to access them and wherever there is register field you can put any registers. So although we have seen that there are cases where some register play a specific role so one exception is register 0 which always have a value 0 you may use it as source, you may use it as destination but its value does not change it is ensured to be so by the hardware and register number 31 is used for return address by `jl` instruction. Apart from these there are no other exceptions and any register could be used for any purpose.

Of course there are some conventions which are followed to ease the task of programming particularly the procedure linkages but that is a matter of convention and given the same hardware one could follow a different set of convention. There is nothing hard and fast about that particular convention. Then apart from these registers there is one special purpose registers which is called PC or the program counter keeps track of the current instruction being executed so it always has the address of current instruction which is under execution.

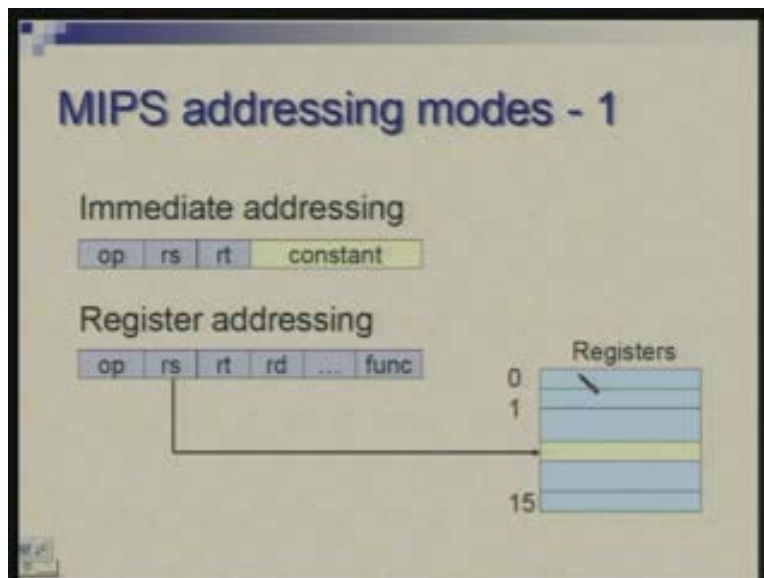
The memory, on the other hand, is an array of 2 raised power 32 bytes or 2 raised power 30 words and each word being 4 bytes and again these are numbered from 0 to..... if you are taking these as words 0 4 8 and so on with steps of 4 it goes up to 2 raised to the power 30 minus 4. So this is what we call as address space that means this is the range of addresses which can be supplied but physically the whole space may not be filled. So, in a particular system you may have, for example, 256 mega bytes of memory. It means that rest of the space is empty and nothing is there. So the memory space, for example, how much memory you require physically depends upon what type of configuration you want to have, what is specified by the architecture is the maximum you can have.

(Refer Slide Time: 8:50)

MIPS ISA features - addressing	
Purpose	Addressing modes
■ Operand sources	■ Immediate
■ Result destinations	■ Register
■ Jump targets	■ Base/index
	■ PC relative
	■ (pseudo) Direct
	■ Register indirect

The next issue is how you access the operands and this is called addressing mode. We need to access operands which participate in for example arithmetic operations. We also need to specify destination of the results of arithmetic or logical operations and we need to specify targets for branch or jump instructions. So these things are specified by one or more addressing modes which you see listed on the right column: immediate addressing mode, register addressing mode, base or index, PC relative, pseudo direct and register indirect. Let me elaborate each of these.

(Refer Slide Time: 9:50)

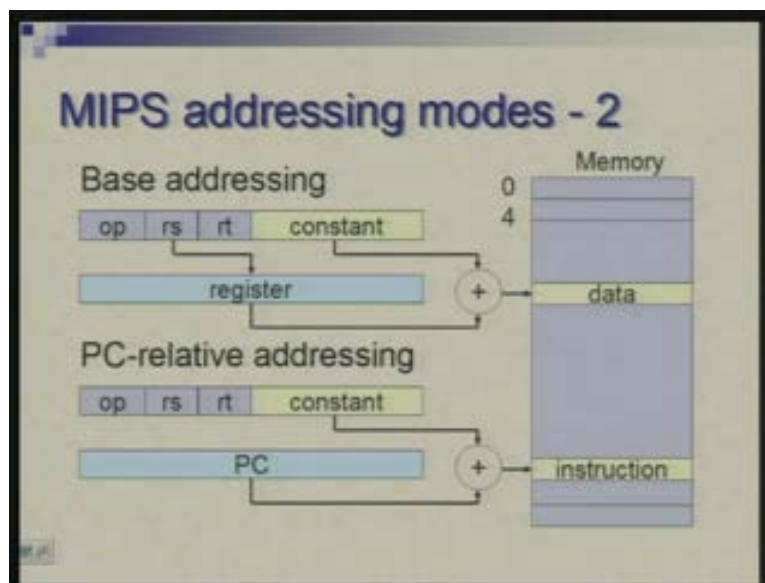


Immediate addressing mode is the one where a constant operand is put as part of the instruction and we have seen this with the number of instruction; all the immediate versions of arithmetic or logical instructions like add immediate. I mentioned there is no subtract immediate, so we have AND immediate, OR immediate, slt immediate so all these instructions can work with constant operands which are provided as part of the instruction. In the case of MIPS these constants are 16-bits uniformly.

Next addressing mode is register addressing where one of the register fields which is a 5-bit field can specify; again I have numbered 0 to 15 but we will take it as 0 to 31; a 5-bit field can specify one of these registers. So the operand or the destination of the result is in the specified register. Now, the first addressing mode which I mentioned the immediate addressing mode is applicable to only operands not for the result.

So, going back to the previous one there are three different situations where we are specifying an address either source operand or destination result or the jump target. The first one (Refer Slide Time: 11:24) is applicable only for source of operand. This is applicable for source as well as destination. The destination address is typically put in this field and source is in one of these two. So, lot of instructions such as add, subtract, multiply, divide, slt, beq, bne AND OR and so on they all refer to operand in the registers and many of these also use registers as the destination.

(Refer Slide Time: 12:21)

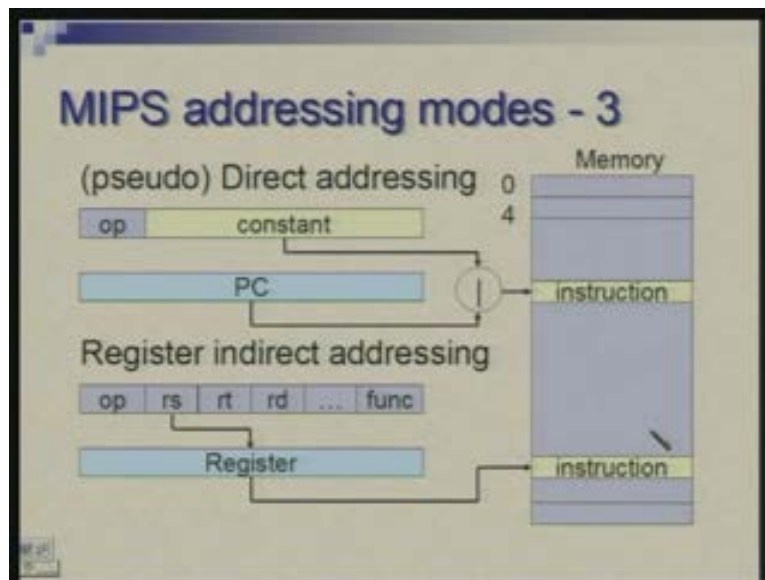


Base addressing involves two things: a register specified by one of the fields in the instruction and a constant which is another 16-bit field in the instruction. These two values are added and the resulting address refers to some memory location. So, in principle such an addressing mode could be used for source as well as destination for variety of operations. For example, one could do addition and specify one of the operands to be in memory. But in MIPS architecture we have seen that this mode is available only for load store. We have load store instruction which make reference to memory but your

arithmetic instructions or logical instructions always assume operands to be in registers or constants; this mode is not available for instructions like add or AND or OR. This is available as source as well as destination; source in case of load and destination in case of store.

Next we look at PC relative addressing which is in principle very much similar to base addressing (Refer Slide Time: 13:41). The register here is an implicit register PC so we do not have one of the register fields specifying a register here but we assume that PC is a register to which we add this constant. Another difference in these two cases is that whereas the constant in base addressing refers to a byte offset; the constant in PC relative addressing refers to a word offset. So, strictly speaking this is actually multiplied by 4 and then added to this address. because if you are saying this constant is 100 basically we mean an offset of 400 bytes; 100 words or 400 bytes so if this is a 32 byte address we need to add this constant after multiplication by 4. So, in principle these are same but there are subtle differences whereas this (Refer Slide Time: 14:48) refers to data in memory and this refers to an instruction in memory so that is the difference.

(Refer Slide Time: 14:57)



Then we come to direct addressing. the meaning of direct addressing is that the instruction specifies address of the source or destination or the target but here we do not call it direct addressing we call it pseudo direct in the sense that the address field in the instruction is not the complete address in itself it needs to borrow a few bits from program counter; this is a mode which we have in jump or j instruction or jal that is jump and link instruction.

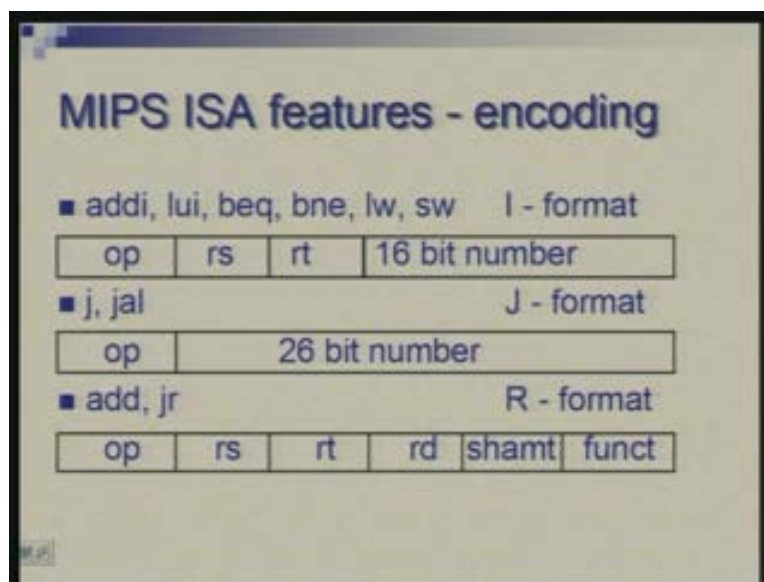
As you would recall, in these two instructions there is a 26-bit constant field which together with 4 bits taken from most significant end of PC form a 30-bit word address and that suffixed with two zeros forms a 32-bit byte address which is used to access an

instruction; the next instruction is accessed by this address. Strictly speaking, the direct address would mean that the entire address is coming from the instruction directly.

Finally we have register indirect addressing which is in jr. jr is the jump on register. A field in the instruction specifies the register and that register points to an instruction where you need to jump. So there is only one instruction which uses this in MIPS. So these are the addressing modes and you would notice that each mode is applicable to specific instructions. As we will see later there are cases where specification of mode and specification of instruction opcode can be done in a totally independent manner. That means there are a set of modes; there could be 6 or 8 or 12 or 16 and each mode is applicable with each instruction. So they are **two** completely orthogonal parts of the specification but here that is not the case.

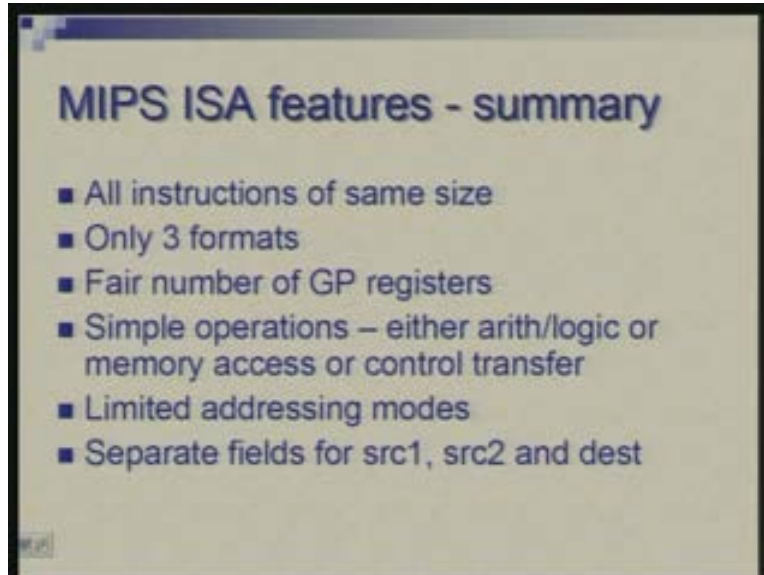
Finally the last feature we wanted to focus on was the encoding; the way instructions are represented in the machine.

(Refer Slide Time: 17:28)



We have seen exactly three formats which were called as I - format J - format and R - format. I - format has a provision of 16-bit constant, J - format has a provision of a 26-bit constant and R - format has no constant but it has three address fields. So the most common format the largest number of instructions actually follow R - format I have not listed all of them. When I say add it means all arithmetic, logical and comparison instructions they will fall here; J is very limited j and jal, I is also used by several instructions so many of the instructions which are of R - format type they have their I version not all but several of them.

(Refer Slide Time: 18:33)

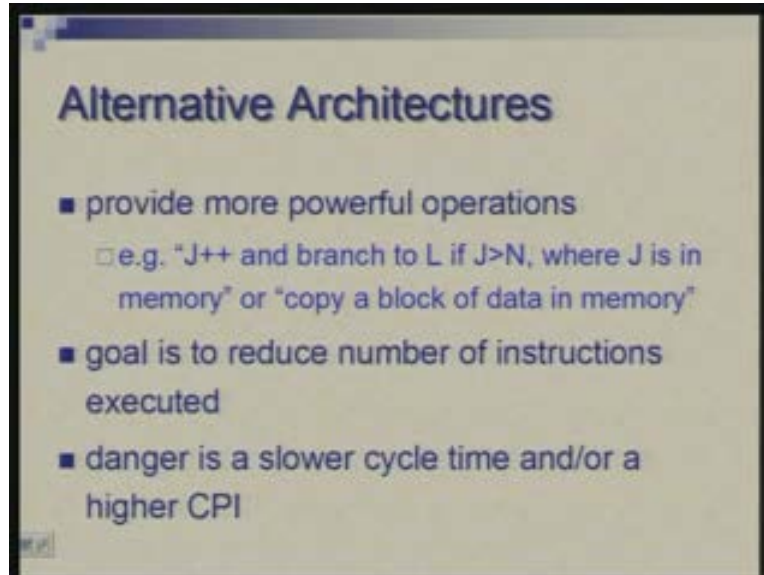


In summary what do we say about the MIPS architecture and it is important to summarize this because from here we will take this as a reference point and see what other things are available. So firstly all instructions are of same size; all instructions are 4 bytes or one word; there are very limited number of instruction formats we have just seen that there are only three formats. There is a fair number of general purpose registers 32 in our case with very small exception. The set of operations is fairly simple and the thing to be noted here is that each instruction tries to do just one thing. What I mean is that an instruction will either do an arithmetic operation or a logical operation or do comparison or do memory access or it will do control flow branch or a jump; there is no instruction which tries to do more than one of these things.

You would notice that in conditional branch instruction we had beq and bne there is some comparison being made. But as we will realize later when we discuss their implementation comparison for equality and inequality is much simpler hardware-wise as compared to comparison for less than or less than equal or greater than and greater than equal. So there is a case where some comparison is being done with branch but the comparison in this case is very simple; we are not doing any arbitrary comparison and branching within the same instruction. So deliberately each instruction performs one very simple instruction.

There are limited addressing modes so with each instruction there is a fixed addressing mode and each mode is applicable to specific instructions and there is no orthogonality. Along with a wide number of registers we have provision for specifying three fields in many of the instructions which perform arithmetic or logic operation. So source one source two the two sources can be specified independently and so can be the destination.

(Refer Slide Time: 18:33)



Now what is the contrasting architecture where do things change; if we pick up another processor where are the changes which are likely to happen?

The common feature that there is a program counter which runs sequentially through instruction is the basic idea of stored program computer which you will find everywhere. So the difference comes in some of these features which I had mentioned. So, firstly in terms of operations there are processors which define very complex operation that has single instruction. So, here are some examples.

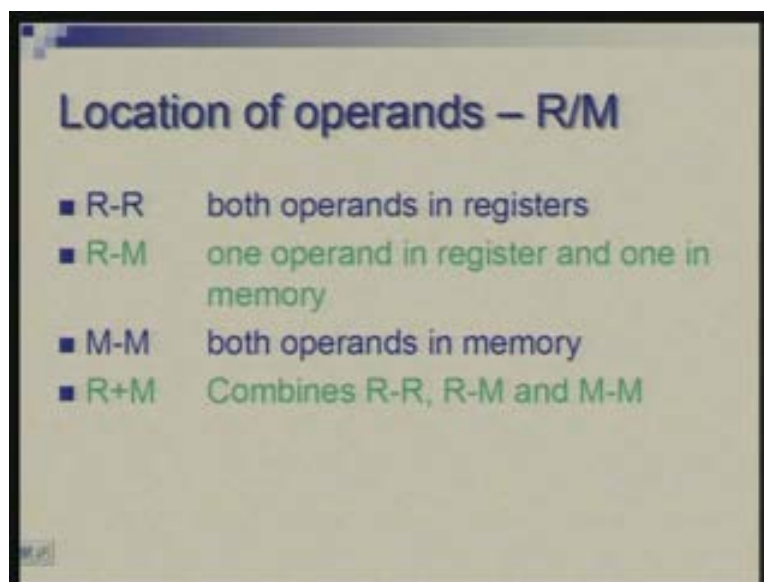
You could have an instruction which does this (Refer Slide Time: 21:58): it takes a variable in memory, increments it, compares it with some value and branches to some target if the comparison succeeds. So you have memory access, arithmetic which is incrementing comparison and branching all happening in single instruction. So it does make a logical sense to have it and with that understanding there are processors which provide such instructions. Or you could look at other operations which are commonly encountered.

For example, it could be copying a block of data from one area in memory to another area in the memory. So there are processors which have single instructions to do this and the goal of including such instructions in your instruction set is to make the program compact and shorter. You should be able to do the same computation with less number of instructions. But the danger the negative side the flip side of this is that the instructions may become slower you may not save time on the whole but on the contrary it may make the machine slower either by slowing down the clock cycle; you know that each processor works with certain clock when you say a 2 GHz Pentium that means the basic time reference is a 2 GHz periodic signal and operations occur with let us say each individual cycles of that clock.

So, trying to include more complex instructions in the set may have a negative impact on the clock frequency. So, a fast clock is possible if you have simple instruction. In addition to this or alternatively this can have impact on the number of clock cycles required for each instruction. So, we are going to see in one of the subsequent lectures that performance does depend on these two factors: the rate at which the clock ticks and the number of clock cycles which are taken by instructions on the average. So it is a tradeoff which one has to perform; you have to have instructions which are sufficient for doing any computation but fortunately that universality comes at a very small cost. As you see in the logic AND OR and NOT together can implement any logic. Similarly with very small instruction you can actually express any computation. It is not a basic necessity to have very complex instructions and any decision to include a complex instruction when you are designing an architecture should be based upon what is its overall impact on the performance; does it actually improve performance or does it make it worse.

One crucial factor is where the operands of instruction are located.

(Refer Slide Time: 25:32)

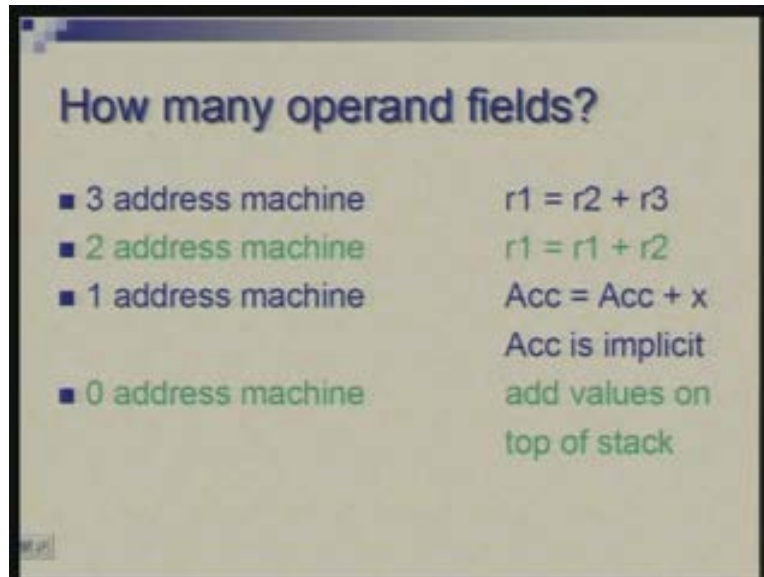


In MIPS we have seen that operands were instructions like add are always in registers. It is faster to access operands which are in registers as compared to accessing data or so in a time in memory. It takes much longer there and therefore instructions which work with register work faster that is the philosophy behind restricting arithmetic operations to register operands. But there are other architectures which do not necessarily stick to this idea. there are architectures which are called RM architecture where one operand typically comes from register one comes from memory or MM where both comes from memory and there are those which actually deal with a mixture of these which will support RR operations RM plus mm so all possibilities do exist.

RR refers to instructions where you have both operands in registers. this issue is also linked somewhere to the previous one they are not independent so restricting the

operands to register is also from the philosophy that you want to separate out memory access and arithmetic and do them with different instructions and the fact that registers can be accessed fast is the second factor which is in favor of having RR type of instructions.

(Refer Slide Time: 27:15)



■ 3 address machine	$r1 = r2 + r3$
■ 2 address machine	$r1 = r1 + r2$
■ 1 address machine	$Acc = Acc + x$ Acc is implicit
■ 0 address machine	add values on top of stack

The next question is about the number of operand fields. Well, when I say operand I mean operand sources as well as destination for the results. So, according to this number the machines or architectures get classified as 3 address, 2 address, 1 address or 0 address so the architecture we have studied is basically a 3 address architecture because the main computing instructions have three fields for specifying two operands and one destination. This means we do an operation like this: $r1$ equal to $r2$ plus $r3$ and these three registers in general could be different. You can choose for two or more of these to coincide but the architecture allow all these three to be different.

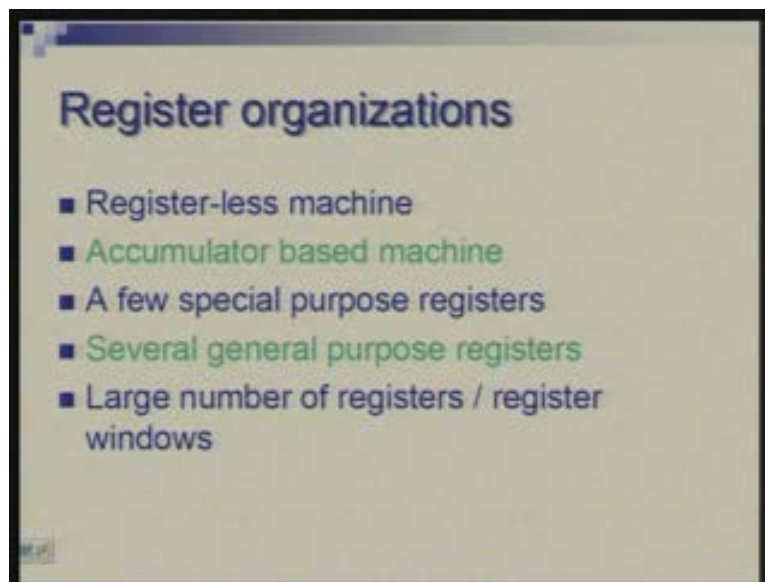
Then you have 2 address machines where typically the result replaces one of the operands. Commonly you will have $r1$ equal to $r1$ plus $r2$ that means it is basically $r2$ getting added to $r1$ that is how one could interpret. Then you have one address machines where you specify one address and the other thing becomes implicit. So Acc stands for a special register called accumulator. Many machines have a special register which is always participating in such instructions as one of the source as well as destination of the result. So the instruction needs to specify only x and Acc is assumed so therefore the instruction becomes 1 address instruction.

Finally there have been some real examples of what we call as 0 address machine where instruction does not specify any source or destination all are implicit, for example, the so-called stack machines, where the operation is always performed on operand which are lying on top of the stack and these operands are removed and the result of the operation actually is put back on the stack. So instruction does not say anything you just say add

and implicitly pick up two values from stack perform the addition and put the result back. A processor, an architecture may have actually a mixture of these instructions but when we classify the machine as a whole we go by typically what the bulk of the arithmetic instructions do.

In case of MIPS, for example, the arithmetic and logical instructions are basically 3 address instructions whereas if you look at simply a jump instruction that has need for only one address. if you look at beq instruction that again is specifying three things two operands being compared and one target address. The next point is about register organization.

(Refer Slide Time: 30:44)

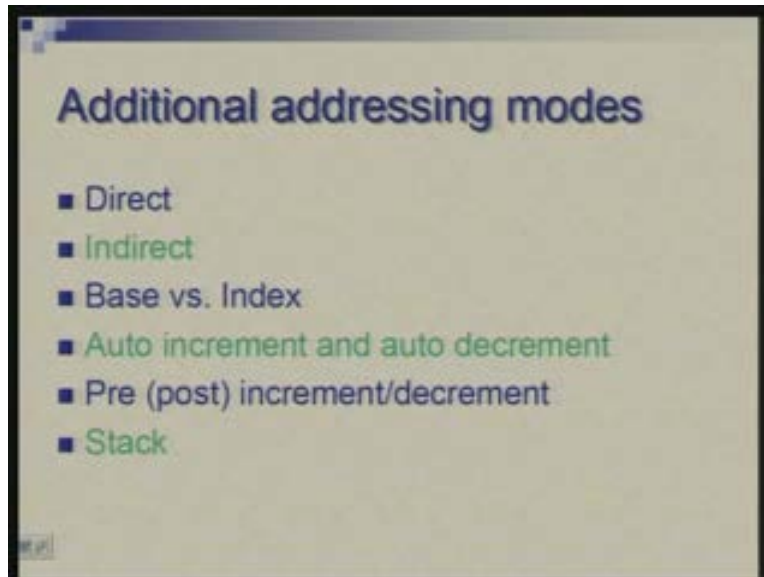


We have seen MIPS architecture which has thirty two registers which is a fair amount fair number of registers. There are those which have few registers let us say eight registers only so the number is very limited. There are also extreme cases where you have a single register which is called accumulator and you cannot really hold much of data in one register you only have the data which is currently participating in operation so everything has to basically come from memory eventually and go back to memory.

You also have registerless machine the stack organization could be of this kind. Thus, registerless means 0 registers, accumulator based machine which means one register, there are also cases when you have more than one accumulator so two or four but not very large, then machines which have not too many registers but each register has its own special purpose, then MIPS like architectures, then there are those which have much larger number of registers, for example, as many as two fifty six registers SUN SPARC is an example of that but there these registers are divided into groups each forms what is called a register window; at a time you can work with one window but you can switch from one window to another window. We will see what is the purpose of this and how this kind of switching is done.

We have seen some addressing modes. There are instances of many additional addressing modes which machines have and moreover the orthogonality between addressing modes and opcodes is also seen in many cases.

(Refer Slide Time: 32:37)



We talked about pseudo direct. For example, there are architectures which have direct addressing mode, the entire address come from the instruction. So, of course instruction size becomes crucial here. If your address is 32 bits and you want to direct address instruction the instruction size has to be more than 32.

Indirect address we have seen in case of jr instruction that is specifically called register indirect. That means the address which you are interested in is kept in a register. There are also machines which support address is being kept in other memory locations so that is simply called indirect and meaning that you are making access to memory first from where you are picking up address and then making another access. Of course it is complicated but there are examples of that.

I talked about base addressing where we have a register that can be called the base register and there is a constant offset which is added to that. In principle it is similar to..... I mentioned PC relative but there is another addressing mode which is similar to this called indexing mode. It is just a matter of the interpretation here. The idea once again is that a constant coming from instruction is added to contents of a register to get the address but the interpretation or the perspective is little different. In base addressing we are saying that the register contains the data value the base and there is an offset over that whereas in indexing mode interpretation is that constant is the base and register index is over that. So, for example, there are processors where you would expect that the starting address of the array is provided by the constant so constant field has to be large enough for that and a register contains the index into array. So if you are accessing ai the

starting address or the address of a zero is the constant part and i is in a register whereas the way we try to access an array..... in our case was that the starting address of the array was in a register which we call a base register and the offset corresponding to a constant index was in the constant field. But in our case if both are large then this does not work we have to actually do address calculation separately.

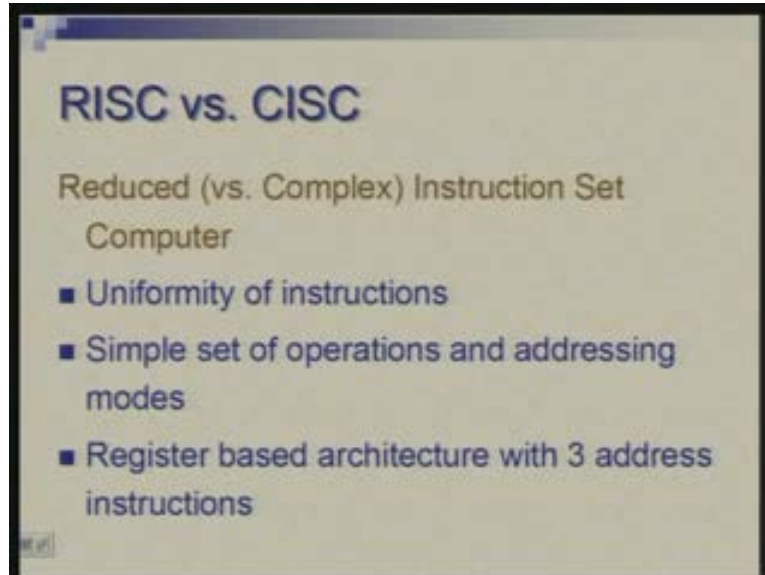
Many times you perform a sequential access to data in the memory either in increasing order of address or decreasing order. So, when you have a register which is providing the address it may be natural to automatically increment or decrement the address. This could be done by more which is called auto increment or auto decrement. What it means is that every time you make an access it is understood that the address has to be made ready for the next access either by incrementing or decrementing depending upon which way you are moving. With this there comes a variation of whether it is pre-decrement or post decrement or pre-increment or post increment that means whether you do incrementing or decrementing before making a memory access or after making a memory access, so, that variation also has to be catered for and many architecture will provide for that.

Then we have stack addressing which actually could be considered as a special case of auto increment and auto decrement. You are using a register to make memory access and also with every access you are incrementing or decrementing so there are processors which provide a stack based addressing with auto increment and auto decrement.

Now, one thing which must be noticed here is that when you provide something which is complex if you try to take care of all generalities it becomes really very complex. for example, with increment and decrement there would be an issue of whether you want to have increment or decrement of 1 when you are accessing a sequence of bytes or sequence of textual data characters for example or increment or decrement by 2 when you are accessing half words or 4 when you are accessing full words or 8 when you are accessing double words let us say for floating point numbers. So, if you try to provide all this generality and an option within the addressing mode that any of these could be specified then it becomes very complex.

In stack, for example, we have seen that it is not just increment or decrement we require with stack pointer sometime we were incrementing by 16 or decrementing by 16 depending upon how much allocation or deallocation we make on the stack. So specific provisions for common cases are helpful but complete generalization may be difficult at times.

(Refer Slide Time: 38:44)



Now, after having looked at all these variations we come to the concept of RISC and CISC; **RISC and CISC**. RISC stands for Reduced Instruction Set Computer and CISC Stands for Complex Instruction Set Computer. RISC is a term which was coined in early 80s by Hennessy and Patterson in contrast to the most popular machines of the day which existed at that time and they were called CISC because they had very complex instruction set and the argument in favor of RISC was that it this approaches one which can lead to better performance.

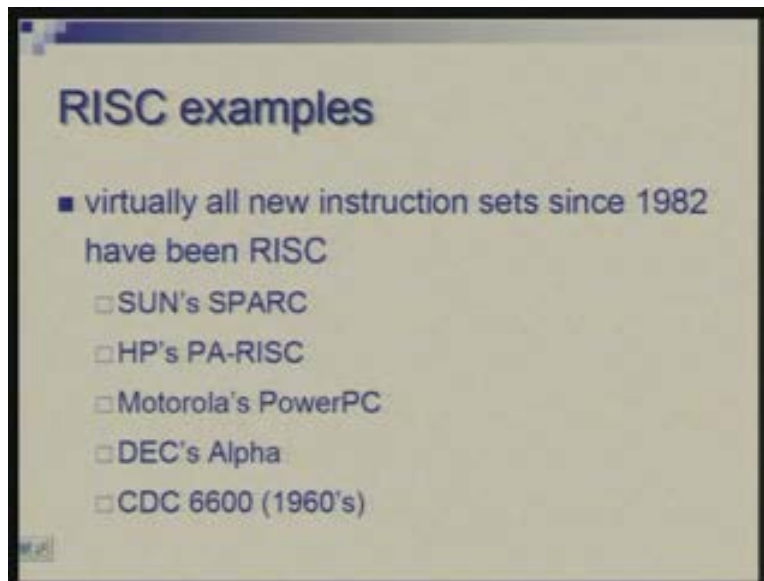
So the main features what we have seen in MIPS is that there is a uniformity of instruction in terms of sizes and a limited number of formats, simple set of operations and addressing modes and register based architecture with three address instructions.

What are the implications of these choices on hardware implementation and performance, we will see in detail later on. But these ideas were propagated basically targeting for achieving high performance at comparatively lower cost. So basically with an architecture called RISC there were RISC 1, RISC 2 and so on they were designed..... you see Berkley by Patterson and contemporarily by Hennessy an architecture called MIPS was designed and what we are studying is the MIPS architecture.

MIPS became a company later on when; they were the company which took up this architecture and there were various versions of MIPS processors and they are in general purpose computing application, also in some video games and so on. On the other hand, the basic ideas of RISC architectures developed by Berkley found their place in SPARC architecture which was taken up by SUN.

In fact beginning with 80s all the new architectural developments the new architecture which was designed were of RISC type and of course the CISC architectures do continue today and the one which is most popularly used is the Intel X 86 architecture which is of a CISC kind but because of historical and commercial reasons it is thriving. Let me mention a few examples.

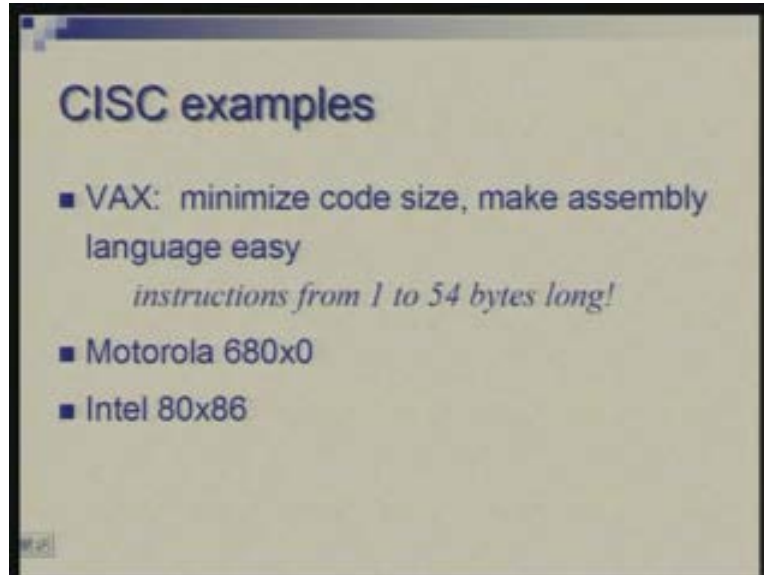
(Refer Slide Time: 41:32)



We will go into details of these in the next lecture but let me just mention. So, SUN's SPARC architecture has its roots in RISC architecture of Berkley; HP's PA RISC PA stands for precision architecture that is the name of this architecture from Hewlett Packard; Motorola developed what it called power PC and DEC which was actually a leading CISC machine manufacturer in 80s came up with alpha architecture. So all these are RISC architecture and it is not only that modern architectures are RISC but the example of this architecture goes back to 60s actually. CDC 6600 was available in around 64 or 1964; the term RISC was not coined then but many features we talked of today actually could be found in this machine which was a very high performance machine of that time.

The classical example of CISC architecture is VAX which is from this company DEC which stood for Digital Equipment Corporation. So VAX had its history in terms of most popular minicomputer called PDP level and this had very complex set of instructions; the instruction size could vary from 1 byte to 54 bytes so you could see the extent of non-uniformity here and hardware which has to interpret instruction in such a wide range is actually very complex.

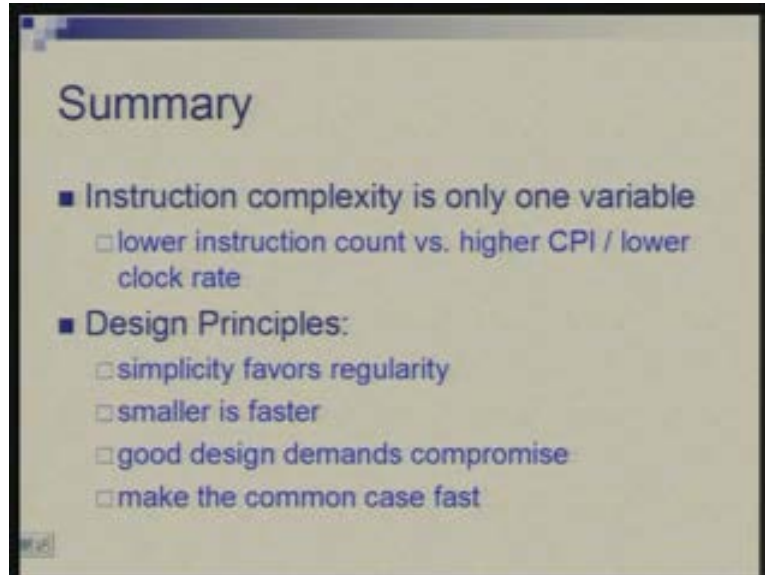
(Refer Slide Time: 42:59)



The objective of this machine was to have very compact code because at that time some crucial components of the program were often developed in assembly language. Particularly in system program operating system many critical parts of the OS typically used to be written in assembly. Of course now almost entire operating systems are written in high level language and considerations are different. So the idea was to make assembly language powerful and easy to use but at the same time what happened was that compilers found it difficult to use the entire set of..... if you have several hundreds of instructions to generate optimal program generate good code trying to make best use of all the instructions was difficult for compiler. On the other hand, with simple instruction set compilers are now able to generate very efficient code and unless we are talking of a small toy program compilers can produce machine code which is much more efficient by handwritten code when it comes to code programs of substantial size.

The other classical examples of CISC machines are the 680x0 series from Motorola. So, it is starting with 68000 and 6810 20 40 60 and so on. Intel's 80x86 has very long history and one can see that some of the features were there in 4-bit version the first microprocessor 4004 then 8-bit processors came, then 16-bit processors, now 32-bit but some features got carried over and in no place the architecture has been freshly redesigned. So carrying old baggage is being carried on which makes the architecture somewhat clumsy and hard to understand and discuss. But this compatibility of code that means you take up code which is used to run 80386 twenty years back and you can possibly still run on modern Pentium so, that compatibility has helped this commercially and more money means you can pump in more investment and technology and have high performance. So these are very high performing processors today not necessarily because there is an elegant architecture or a beautiful architecture but because the technology has been so perfected that you can make things run fast.

(Refer Slide Time: 46:45)



Let us close by summarizing a few things. We have seen that instruction set complexity is one of the main issues and this influences the performance by impacting the clock frequency and the cycles you require for each instruction execution. We are going to elaborate on this later on on how this actually effect when we go into details of the design and some good design principles which we have seen so far are that: simplicity requires that you have regularity and uniformity, smaller is faster; so, if you are accessing a smaller structure working on smaller operands it always is faster. So although uniformity is most desirable but you need to make a compromise and make a few exceptions and **good not to make too many exceptions** then it pays off to make a common case fast. So something which is done let us say 90 percent of the time if you can focus your attention on that and make that fast as compared to that remaining 10 percent then the efforts would be well spent. I will stop at that.