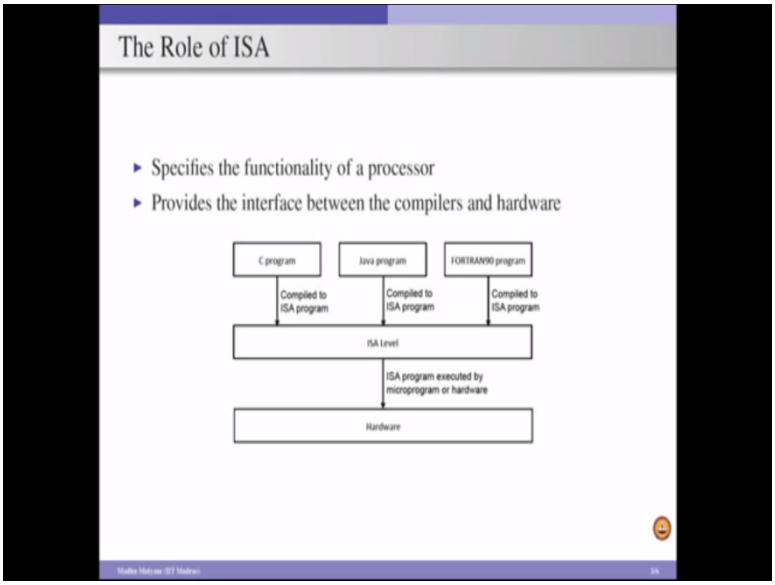
**Computer Architecture**
**Prof. Madhu Mutyam**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Madras.**

**Lecture - 03**

So, in the last module, we discussed the quantitative principles of computer design and in this module we are going to discuss instruction set principles. So, we know that computer systems are very complex and in order to deal with the complexity so, we follow the layered view of computer design, where each layer abstracts the layers below it and supplies this abstraction view to the higher layers. So, one such layer in the computer design is architectural layer. This architectural layer provides the instruction set architecture.

(Refer Slide Time: 00:54)



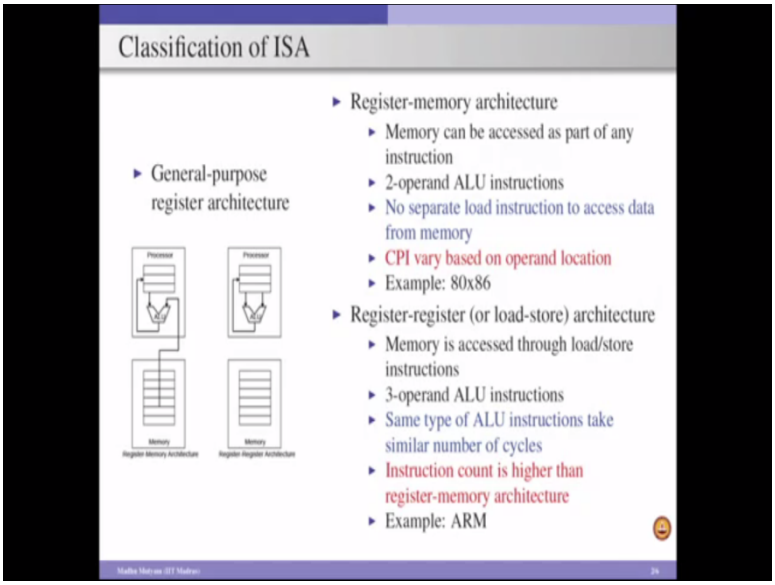

So, this instruction set architecture specifies the functionality of a processor in terms of what type of instructions it executes, what are the instruction representations, what are the addressing modes, memory addressing, registers and etcetera. And actually this instruction set architecture is the interface between compilers and the hardware. So, as a result it provides to the programmer, the functionality of the entire processor so that, the programmer can write a program which can be efficiently executed on the hardware.

If you see this diagram, this consists of multiple programs written and different programming languages and there is a layer which is ISA level and below to that is a hardware layer. So, program written in a particular language will be compiled using the corresponding compiler.

And after compiling we get an intermediate form of the language that is an ISA level program, which is nothing but represented in machine language, because the computers will only understand the machine languages, which are in terms of 0's and 1's.

So, program written in C language can be compiled to come up with an ISA program. Similarly, a program written in Java can be converted to an ISA program and similar to that of Fortran 90 program. So, this gives the programmer flexibility, so that he can choose any programming language and compile his program.

(Refer Slide Time: 02:45)




ISAs can be classified into 4 groups - 1 is stack based ISA, the another one is accumulator based ISA and the third set is General purpose register architectures which further divided into 2 groups one is a register-memory architecture and other one is register-register architecture. Because the current ISAs typically follow the general purpose register architecture, we are going to discuss only the general purpose register architecture are also called GPR architectures.

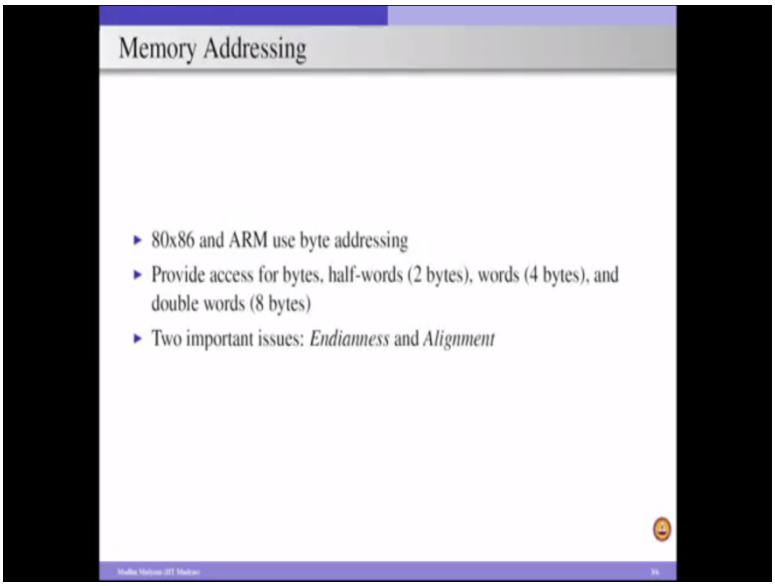
In the case of register-memory architecture, we have 1 operand supplied by the processor register and the other operand can be from the memory. And in the case of register-register architecture our operations will take operands from the processor registers and we go to the memory only for memory accesses which are through a specified instruction called as load and store.

So, in register-memory architecture, memory can be accessed as part of any instruction and typically the instruction formats consists of 2 operands, where 1 operand can act as both the source and the destination and because any instruction can access memory so, there is no need of a separate instruction to access memory. So, that will reduce your instruction count in the program, but the disadvantage is cycles per instruction can vary depending on the location of an operand.

For example, if I consider an add instruction where 1 operand is a memory location and the data can be available in the cache memory versus a similar instruction where operand can be available in the main memory. So, both takes a different number of cycles to execute. So, this is a problem with register-memory architecture. An example ISA which follows this type of thing is 80x86. And the second type of GPR architecture is the register-register architecture where the memory is accessed through separate instructions called as load and stores.

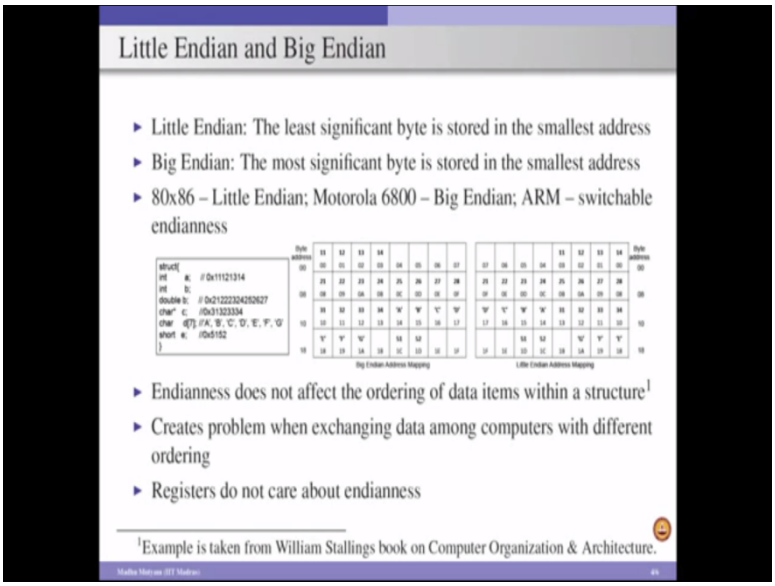
And as a result this architecture is also called as load-store architecture. All ALU operations will be performed on register operands and any time if we require some data to be accessed from the memory, we use load and store instructions. And the instruction format consists of typically 3 operand instructions where 1 operand is the destination and 2 other operands are source. Since, we perform operations on register operands the same type of ALU instructions take the similar amount of time to execute, because the access latency from the register file is same, whether we are accessing register 1 or register 10 in the register file set. But, since we use a separate instruction for accessing memory, the instruction count increases as compared to the register-memory architecture. The example ISA for this is ARM.

(Refer Slide Time : 06:30)




So, now let us discuss the memory addressing. And again we are going to focus on the architectures which are currently used predominantly so, such as the 80x86 and ARM. So, these ISAs use byte addressable memory locations. So, any memory location can be addressable and we can get an access to bytes, half words, words, full words and so on, but there are 2 issues with accessing data from the memory. Those are Endianness and Alignment. So, we first discuss what are the issues with the Endianness and what is an Endianness and so on.

(Refer Slide Time: 07:27)



So, there are 2 types of Endianness used in different ISAs. One is little endian the other one is big endian. So, in little endian, the least significant byte is stored in the smallest address. The exactly the opposite happens in the case of big endian where the most significant byte is stored in the smallest address. To understand this we can consider an example, but before that. So, we will see which ISA uses what type of Endianness, 80x86 uses the little endian, Motorola 6800 uses big endian and ARM and RISC processor use switchable endianness.

Now, consider an example, consider a simple structure which consists of set of elements, int a, which has a value 11121314, int b, which is not initialized with any value, double b, character pointer c, character array and short e. And if I am going to store this using big endian and little endian formats, we will see how these elements will be stored in the memory. Generally in the case of big endian, we typically consider this format of address mapping, where byte addresses are start from left to right, top to bottom.

And whereas, in the case of little endian we start from right to left, top to bottom. When we consider big endian format so, int a requires 4 bytes of space and as we mentioned in the definition the most significant byte is stored in the smallest address. The smallest address is 00. So, 11 is stored in the smallest address location that is 00. And after that the second one 12 byte will be stored in the second byte address and so on. And 14 is the last byte of the int a which is stored at address 03.

And the second integer variable is not initialized at all. So, as a result we keep the empty space there. The next 4 bytes are empty. Then to store b which is a double variable so, it requires 8 bytes of space and again 21 is the first byte so, which is stored at the smallest address which is 08 and so on. And character pointer c so which has 4 bytes of space and that also again starting from 10 address all the way up to 13 we store the character pointer data.

Next, the character array which consist of seven characters in the array a, b, c, d, e, f, g which are stored starting from address 11 all the way up to 1A. And finally, the short variable e which requires 2 bytes of space which is stored in 1c and 1d locations. Remember the character array d which is actually requiring seven characters and we left 1 character space there. So that, variable e is aligned with particular address and so on and we will see what is alignment in the next foil. The same thing if we want to represent in a machine which supports little endian format then it will look like this. As I mentioned earlier, the addresses are from right to left and top to bottom. So, that the least significant byte of a particular

variable is stored at the smallest address. If I consider the first variable, an integer variable a, whose least significant byte is 14 so we are storing that at the smallest address which is 00.

So, that the first byte of integer variable a is stored at the address 03. And similarly, the other elements storage can be explained. But if we observe one particular thing which is character array d, so character a in the character array d is stored at the address fourteen 14 in the case of big endian, but the same thing is true in case of little endian. By the way these addresses are presented in hexadecimal and so on.

So, except the character array d, all other things are strictly following the little endian or big endian definition, but in the case of array the ordering is not changed among the elements of that array. So, character g of d is stored at address 1a in both the little endian and big endian.

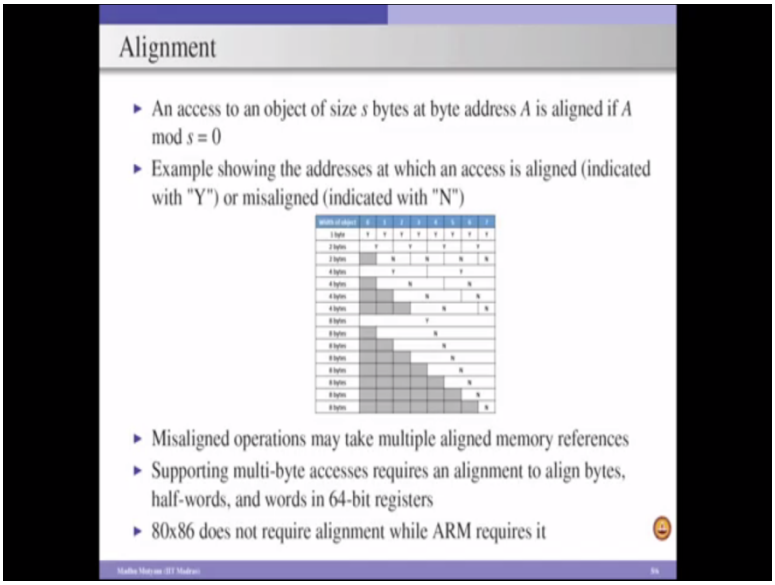
So, this actually says that, endianness does not affect the ordering of data items within a structure. Even if you see the example, the entire example, which is a struct, consist of collection of variables of different data types and we have not changed the order among these elements of the structure. The same thing is true even with the arrays because array is now, in this particular case it is a character array. Each character takes 1 byte of space. So, as a result at the byte level the Endianness will not change. So, as a result, the character array is stored the same address locations in both the endianness. So, Endianness creates a problem when we are transferring some data from a computer of, which is using a particular Endianness to another computer which uses a different Endianness.

For example, if we are transferring the data from a computer which uses a little endian to a computer which uses big endian, data you interpret will be completely wrong because the data will be completely reversed to that of the other order and so on. So, as a result when you are transferring the data and so on, you need to specify what is the Endianness used. So, that you reconvert it back and get the correct data. And in the case of registers, Endianness does not matter because the registers typically store the least significant bit at the rightmost position and the most significant bit is stored at the leftmost position irrespective of what Endianness used and soon.

So, when I say 32 bit register. So, always the 0th bit is stored at the rightmost and the 31st bit is stored at the leftmost bit position in the register. So, as a result we do not have to worry about the data stored in a register, but that is not true in the case of memory because memory

follows one of these two Endianness. And coming to alignment again some ISAs specify that alignment should be followed and some do not.

(Refer Slide Time: 16:08)




An access to an object of size s bytes at byte address a, is aligned if a mod s is equal to 0. So, this specifies that if I have an object of 1 byte. So, it is aligned it can be placed anywhere and it is aligned, whereas in the case of 2 byte object, always I have to store this object in the even locations of the address. If I store that in the odd address location it is not said to be aligned. So, here we consider an example which shows the alignment for storing 1 byte data, 2 byte data all the way up to 8 bytes of data.

So, 1 byte data is aligned perfectly, in the case of 2 byte only those which are placed at even address location that is 0, 2, 4, 6 are aligned, but whereas, if we store the data at the addresses 1, 3, 5 those are not aligned. Similarly, in the case of 4 byte and 8 byte, but what is the problem with this alignment? Why do we have to worry too much about this alignment? So, consider a scenario where you want to access 4 bytes of data, but the data item is stored at the 7th byte address or odd address, which is at the boundary of this memory.

Typically, memory is placed in the aligned order of 4 bytes, 8 bytes and so on. If your data item is starting from the boundary of one particular location in the memory to the next memory location, then in that case we require 2 accesses to the memory to get that element. An example is, consider a 4 byte address, 4 byte data which is starting at the $7^{th}$ byte of the memory. So, it will go to $7^{th}$ byte, $8^{th}$ byte, $9^{th}$ and $10^{th}$ byte.

So, you have to access $7^{th}$, $8^{th}$, $9^{th}$, and $10^{th}$ bytes from the memory to get that 4 bytes of data because we know that, in this particular example memory is aligned at 8 byte granularity. So, $7^{th}$ byte will be there in 1 particular location and $8^{th}$, $9^{th}$ and $10^{th}$ bytes will be there in the other location. So, you have to access memory twice. First time you access the complete 8 byte word and take only the last byte of that. And in the second memory access you again access again another 8 byte data and take the first 3 and put these two together to supply the 4 byte of data.

So, because in any access is taking more time so it incurs significant penalty in terms of performance. So, as a result it is always good to have aligned data. So, that is what the misaligned operations take multiple aligned memory references which incurs penalty. Even if the system which supports aligned accesses and so on, but supporting multi-byte accesses such as 1 byte access, 2 byte access, 4 byte accesses, to store the data in the corresponding 1 byte, 2 byte and 4 byte locations in 64 bit register, we need to again do alignment because when I get 1 byte of data.

So, I have to store that in a appropriate location in 64 bit register which requires alignment. So, 80x86 does not require any alignment, but whereas, ARM requires that alignment. So, with this I am going to conclude this module and in the next module I am going to discuss addressing modes.

Thank you.