Advanced Computer Architecture

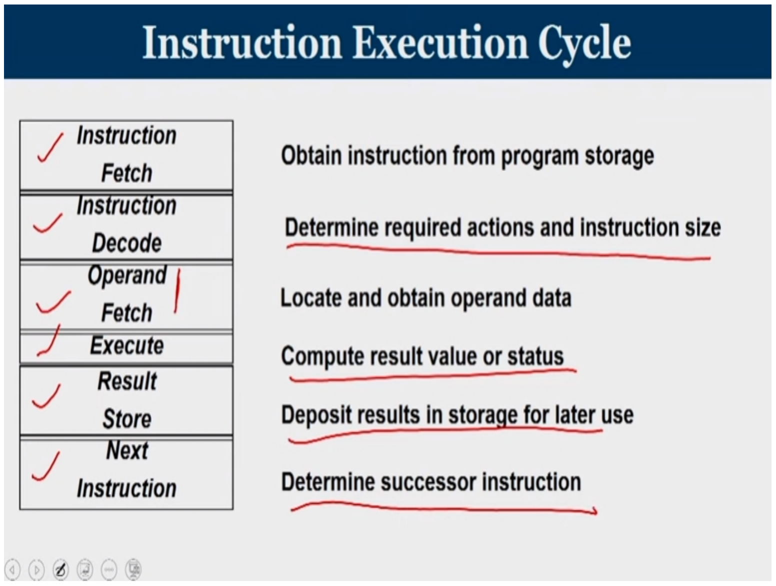
Prof. Dr.John Jose
Department of Computer Science & Engineering
Indian Institute of Technology – Guwahati, Assam

Module No # 01
Lecture No # 02
Review of Basic Computer Organization

Welcome to the first lecture on advanced computer architecture and NPTEL MOOC's course. The entire course is organized as 20 lectures, supplemented by few problem solving and tutorial sessions. In the first lecture, we will be trying to recap the basic concepts that you all might have learned in an undergraduate level computer organization course. This quick recapping will make sure that all of you are at the same pace once we dig into the advanced concepts in computer architecture.

So, for those who are familiar with the concepts of computer organization, this will be a quick revision and for others, it helps us to understand the fundamentals associated with advanced computer architecture design. We will focus our attention first into how instruction execution happens inside a microprocessor.

(Refer Slide Time: 01:30)



Instruction Execution Cycle

✓ Instruction Fetch	Obtain instruction from program storage
✓ Instruction Decode	Determine required actions and instruction size
✓ Operand Fetch	Locate and obtain operand data
✓ Execute	Compute result value or status
✓ Result Store	Deposit results in storage for later use
✓ Next Instruction	Determine successor instruction

We all know that computers are used to execute tasks. For a computer to do a task, the first step involved is the task has to be represented in a machine understandable language. So, when you have a task that is in a high level language like C or C++ or basic or Fortran, we use a compiler to translate this task into a set of machine understandable instructions.

These instructions are then stored inside the memory of a computer and then processor for execution of this task will interface itself with the memory and try to take these tasks or these instructions from memory and do an analysis inside the processor and carry out the execution and then continue with the future task. When you look into the instruction execution cycle, we can see that the execution of a single instruction is subdivided into smaller sub micro operations.

We will try to understand what are they, In the slide given we can see that for each operation that is to be carried out with respect to an instruction, we have multiple sub operations like instruction fetch, decode, operand fetch, result store and next instruction. We will try to understand what the sub operations are. When you talk about instruction fetch, it is a stage in which an instruction is brought from the memory and placed inside the processor.

To facilitate this, the processor will issue the address from which the instruction is to be fetched and that is typically the content of program counter. So program counter issue the address and this will go into memory from there the addressed word the addressed instruction is being brought into the processor and that is called instruction fetch. The next step is called instruction decode.

Once the instruction is brought into the processor, the processor has to understand what does instruction is. So, an instruction basically consists of two components. The first component is called the OP code with respect to a instruction, which means what is the operation to be done. The second aspect is the operand which specifies where the operation is to be done. So trying to understand what the OP code and operand is that is called instruction decode.

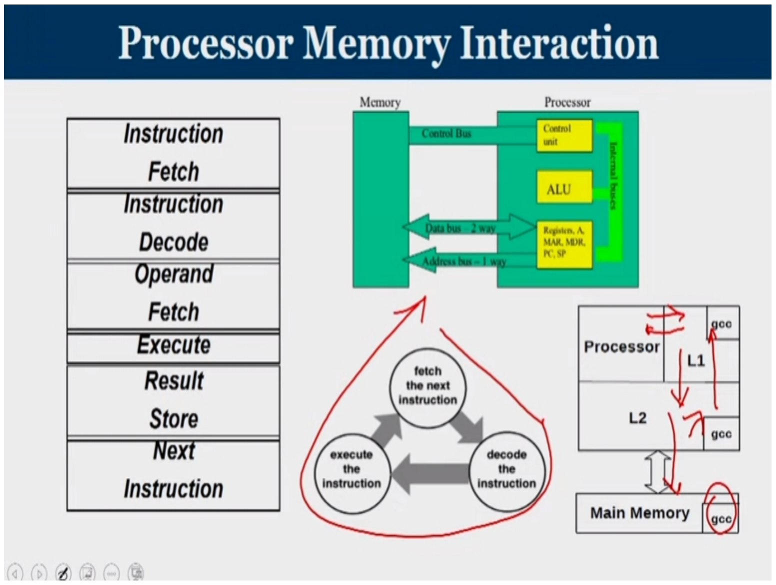
And followed by the decoding as a byproduct of decoding, the process of has to generate appropriate control signals in a sequence to carry out the execution of the operation. Having completed instruction fetch and instruction decode, the very next stage is called operand fetch. Sometimes in order to carry out an operation the operand has to be brought from registers or from memory that is called operand fetch stage.

Operand is a place where data is located. After understanding what the operand is necessary control signals are being issued by the control unit of a processor upon which the operands are brought into the execution unit. Once the OP code is understood, and the operands are ready, now, the next stage is execution of instruction. So, compute the result value typically these been done in the arithmetic logic unit if it is an arithmetic or a logical operation or in other functional unit, if it is done on a floating point unit or any other associated functional units.

The next operation is you have to store the results. Once the result is computed, the very next step is we have to store the result. The storing can be either inside memory or it can be inside the processor. With that the execution of one instruction is over now, the task is to find out what is the next instruction to be executed. And that is the final step, determine the successor instruction. If it is a normal sequencing operation, then the successor instruction is the very next instruction of the current instruction.

If the current instruction is a branch, then depending on the outcome of the branch operation, we have to find out what is the next instruction. It can be either the adjacent instruction or it will be a target instruction which is specified as part of the instruction. So, these are the various stages that are associated with execution of a single instruction. We will now try to understand how a processor is going to interact with the memory, the interfacing of a processor and memory.

(Refer Slide Time: 06:45)



Processor Memory Interaction

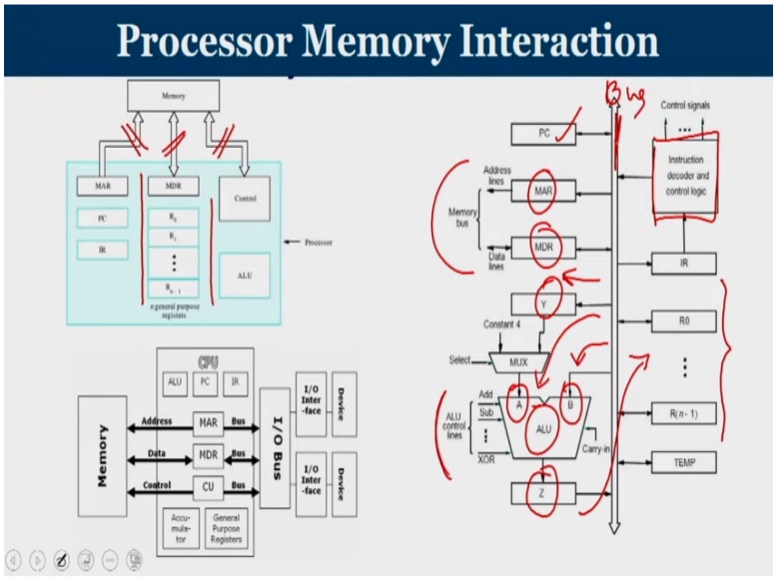
So, when you look at the slide you can see there is a processor which has an arithmetic logic unit which will take care of the basic operations and the whole operations are controlled by a control unit. And we have a couple of registers, which takes care of all these operations processor is interacting with the memory with the address bus through which the address of the desired location in the memory is communicated.

And then we have a data bus through which data is going to flow from the processor to memory and from memory back to the processor. And the signals that are generated from the processor and the response signal from the memory will pass through the control bus. So, it is the unidirectional address bus and the bi-directional data bus as well as the control bus which takes care of smooth handshaking of operation between the processor and the memory.

If you look into further deeper into what a processor does, you can see that the processor is going to fetch the next instruction always, decode the corresponding instruction and execute the instruction and this whole process is going inside a cycle. Generally, processors have multiple levels of cache memories, which are high speed memories that are inside the processor chip and outside the processor is the physical memory or it is also known as main memory.

So here we can see that there is a program let us say it is GCC grew compiler it is running inside the main memory since the processor is currently running a compiler a copy of this GCC is been brought into the L2 cache and from their copies made into L1 cache. So processor fetching operation happens from L1 cache and if at all it is not there, you go to look into L2 and bring it to L1. If in L2 also you are not able to find out that desired location which is demanded by the processor then control is transferred to main memory to carry out the remaining operation.

(Refer Slide Time: 08:56)



Processor Memory Interaction

This is yet another representation of processor and memory interaction. We have a couple of special registers inside process of known as MAR and MDR. Then program counter instruction register, then we have general purpose registers, you have ALU and the control unit. Then the data bus, the address bus and the control bus that is going to operate. MAR, MDR and control unit are the three important components of the which MAR and MDR registers, they are responsible for the direct interfacing with memory.

If the processor wants to fetch an instruction, the first thing that you have to do is the address of the instruction which is available in the program counter has to be transferred to a MAR. Similarly, if processor wanted to read or write any data into memory, then also the address has to be kept inside MAR. So MAR is a register which is known as memory address register, which contains the address of the next word that has to be accessed in the memory.

It can be either for a read or a fetch operation or it can be for a right operation. Now whatever is the address that is placed in MAR, the data contents are being exchanged through MDR. So, if it is a write operation, then the contents in MDR are transferred to memory on a location specified by MAR. If it is for an instruction fetch or a read operation, then the contents of the designated location specified by MAR are being transferred to the processor and it reaches MDR first.

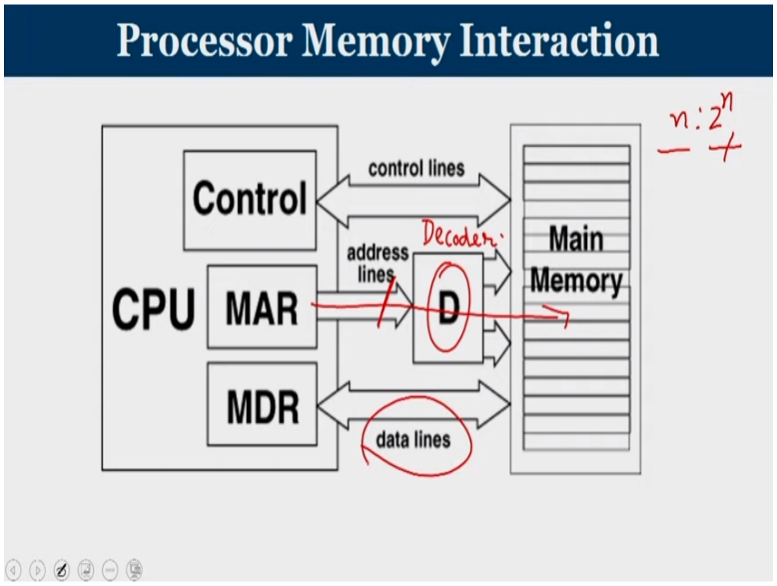
This is the role that has been played by memory address register and memory data register. What you see in the right side of the slide is yet another representation of the internal organization of a processor- a general processor, you can see there are many general purpose registers, you have an instruction decoding and control unit. the program counter, MAR and MDR like what had been mentioned already, these are the two registers which will directly interface with the memory, MAR carries the address and MDR carries the data.

Then you have the arithmetic and logic unit, which has two inputs, one is designated as A other one is designated as B. Through A one of the operand will reach ALU, through B the second option will reach ALU and the control unit will generate necessary control signals, whether it is an add operation or it is a subtract or a logical operation like or, and and all that is been specified by the corresponding control lines.

Since it is not possible to bring two operand through A and B together one of the operand is brought to a temporary register called the Y at one clock cycle and in the adjacent clock cycle you are going to bring the contents into B and then the content that is already stored in temporary register Y is being transferred into A.

The result of ALU operation is stored in set and depending on where is the result to be stored, it will be moved from Z the temporary output register into appropriate general purpose register through the internal process of bus. So this is the internal processor bus.

(Refer Slide Time: 12:11)



Now processor and memory working closely together in order to execute a task that is what we have seen. We can also see that the MAR which generates address is going to link with main memory and their exist a unit which will uniquely choose one main memory word based upon the incoming address and this is the decoder. It is a memory decoder the duty of the decoder is whatever is the address that is been given by MAR based upon that decoder will uniquely choose one word and that is a designated word.

So generally a decoder has

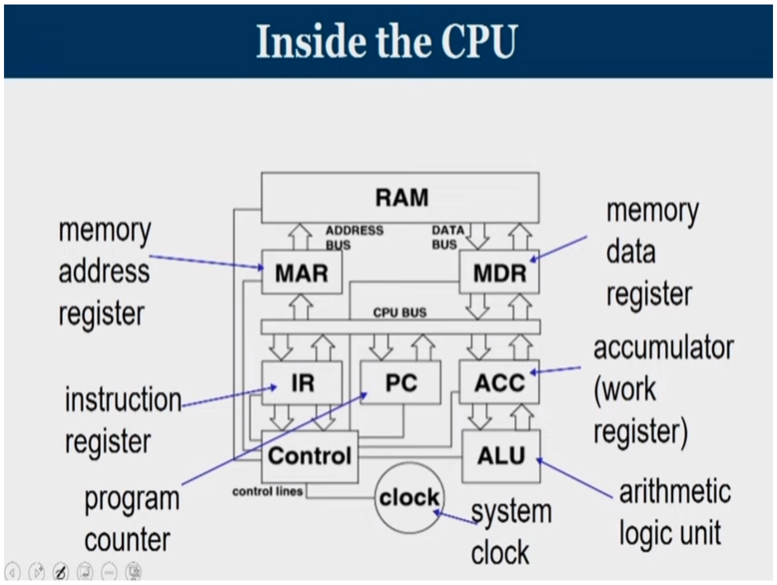
n inputs and 2^n outputs.

Based on the

n inputs, one of the 2^n outputs

are being uniquely chosen and memory data register is interfacing with main memory through the data bus that is also known as data lines.

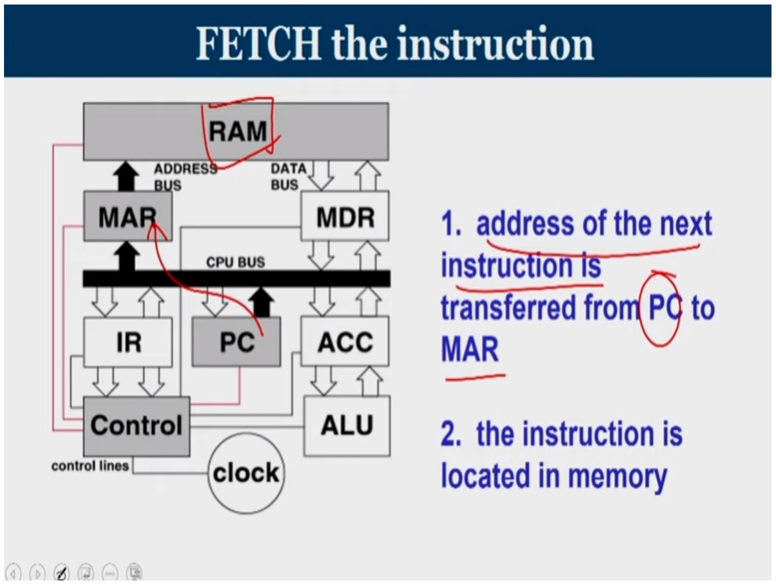
(Refer Slide Time: 13:07)



When you look deep into the CPU, you can see that all the important registers that we discussed are already shown here and the whole thing is being controlled by a clock signal. Let us try to understand what these operations are? We have a memory address register, then we have a memory data register. Then there is a program counter which contains address of the next instruction to be fetched. Instruction register, which carries the instruction once it has been fetched from the memory.

And then you have arithmetic logic unit which takes care of all the operation. And there is an accumulator or it is also known as a work register, which is directly associated with ALU and it carries one of the operating majority of the arithmetic operations. And then there is a system clock, which is directly connected to all these units to make sure that they are working in a coordinative way.

(Refer Slide Time: 14:03)



FETCH the instruction

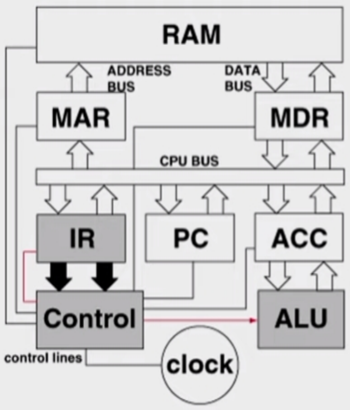
1. address of the next instruction is transferred from PC to MAR
2. the instruction is located in memory

Having non various registers inside the processor.. now, let us try to see what are the implications associated with the fetch operation. So the fetch operation generally starts from a program counter. If you wanted to fetch an instruction, the address of the next instruction, which is located in the program counter is transferred to MAR. So the content of PC is transferred to a MAR, that is the very first step, then the instruction is actually located inside memory.

So whatever the MAR is giving, that is an address based upon that the instruction is located inside the memory.

(Refer Slide Time: 14:38)

EXECUTE the instruction

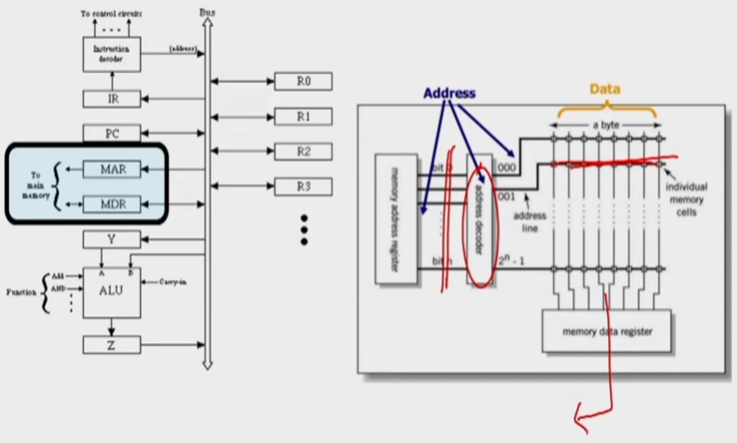


5. control unit sends signals to appropriate devices to cause execution of the instruction

Now, the very next step is the located instruction is now transferred to MDR which is the memory data register. Now, from MDR through the CPU bus, the instruction is transferred to IR where decoding happens with the help of the control unit. Control unit send the signals to appropriate devices or units inside processor to facilitate the execution instruction.

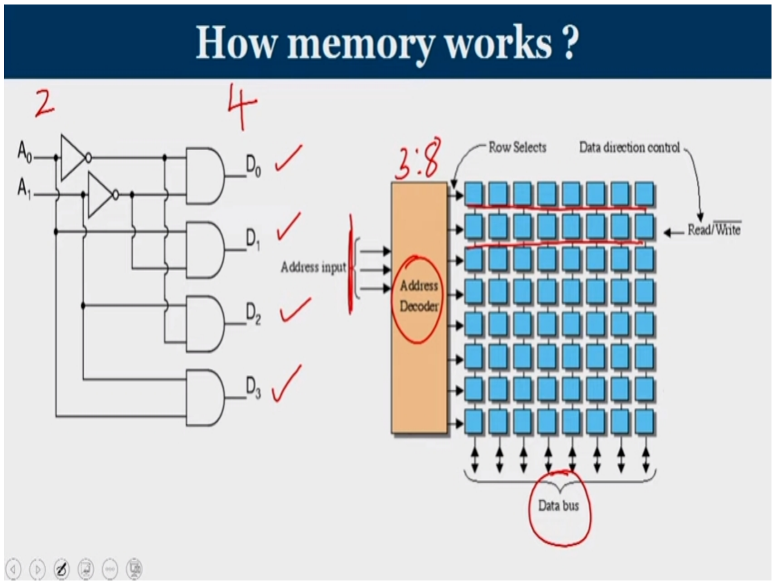
(Refer Slide Time: 15:06)

Processor Memory Interaction



So, we have seen the general structure of the internals of a processor and MAR and MDR are the two registers that are responsible for interfacing with memory. We have already discussed that there is an address decoder which decodes the input lines that has been connected that is nothing other than the memory address registers output the address associated with a word. Based upon address one of the word is uniquely selected and the contents are transferred to memory data register.

(Refer Slide Time: 15:38)



How memory works ?

Like what I mentioned, the process of uniquely selecting a memory word from a given address is being facilitated by an address decoder given is the circuit of a

2x4 decoder has: 2 inputs and 4 outputs.

The function of the decoder is based upon these 2 inputs, one of output is uniquely been chosen,

if both the inputs are 0: then D0 is chosen

If both the input A0 and A1 value is 01: then D 1 is chosen

if it is 10: D2 is chosen

if it is 11, D3 is chosen.

Similarly, the internal circuit of a

3x8 or a 5x32 decoders

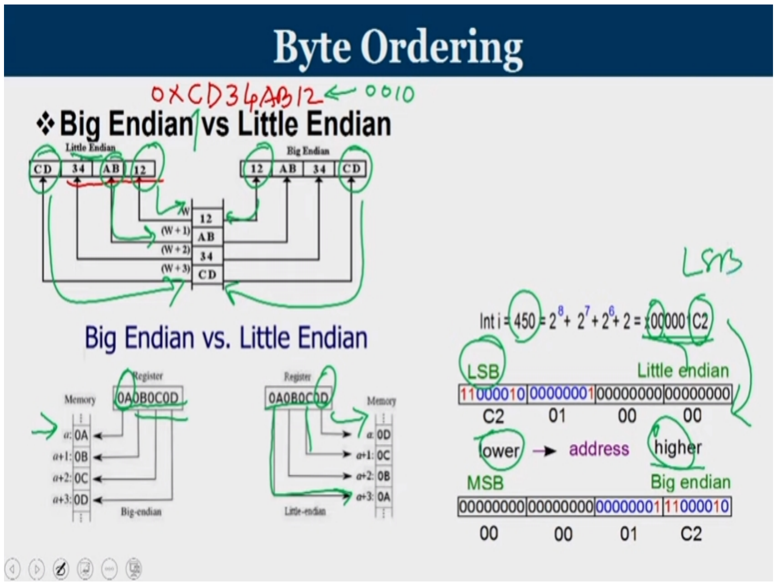
also can be drawn. So, the given on the right side slide, we can see that it is a 3 input address that is coming and the decoder is a 3x8 decoder based upon that one of the 8 words is being uniquely chosen and the contents are transferred through the database. So, when you have larger memory, then proportionately a bigger decoder is being used and this recorder is going to take more time if the number of inputs are more.

So, when you use large memory, naturally it is supported with the help of a bigger decoder with more number of inputs and outputs. So decoding will take more amount of time. So, when you give an address, it will take more time if it is a bigger decoder or addressing smaller memories are much more faster than addressing larger memory because of the size of the decoder that is being involved.

Let us now look into another aspect of interfacing with memory. We know that in memory, you have words, but the basic unit of storage inside memory is a byte. When you operate on data, all data may not be equal to 1 byte, sometimes certain data may be bigger than 1 byte, meaning the word length will be multiple bytes, when processor is going to read and write larger words, when it comes to storage inside memory, we have to understand in what order that they get stored.

So there exist a format in which multi byte words are being stored inside memory they are little Endian format as well as big Endian format. We will now look into and then try to find out what is the difference between these two storages.

(Refer Slide Time: 18:22)



Byte Ordering

- Big Endian vs Little Endian

As mentioned in the slide, that has consider the case that we are going to store a

32 bit number represented us 0XCD34AB12

where each one will represent a hexadecimal number.
Let us say

2 stands for 0010

similarly,

1 stands for 0001

like that, you can define all of them. This 32 bit number when I am going to store inside the memory is the least a significant byte is stored in a memory W that I can store one byte of information in memory whose address is W.

So if LSB saved in W the next byte from LSB stored in W + 1 similarly, if the MSB is saved in W + 3 this kind of a storage representation is known as little Indian format. But some processors follow big Endian format where the MSB more significant byte is stored in the lower address and the least significant byte is stored in the higher address that is known as Big Endian format. So, consider the case wherein in a register you have a value 0A, 0B, 0C and 0D.

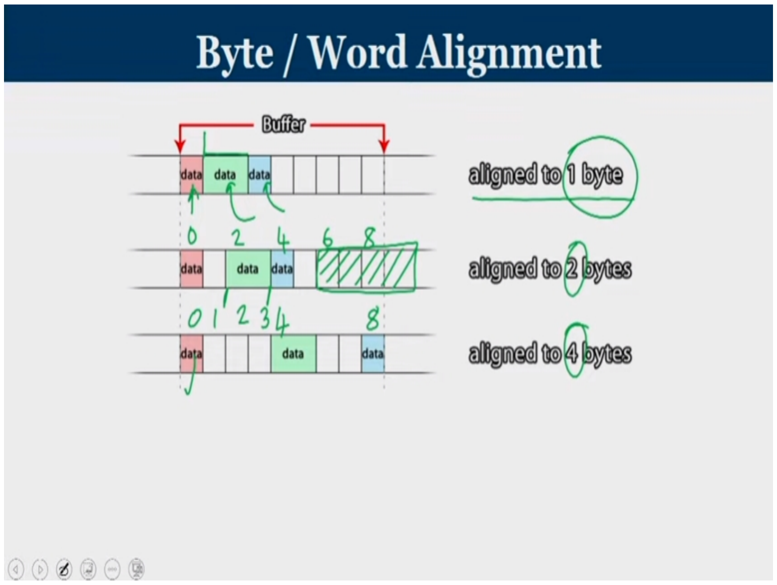
This 32 bit number or a 4 byte data upon storing in a big Endian format than the most significant by 0A will be stored in the lower address the address small letter ‘a’ and the next higher address will carry the remaining bytes in sequence. If the same number if I am going to represent in Little Endian format, then the least significant byte 0D will get stored inside memory location A and the next higher address will carry 0C.

Similarly the highest address will carry the most significant byte that is the basic difference in which. Similarly, if a number 450 if you are going to represent number 450 in a 32 bit value, then this is the numerical value equivalent to 450. If it is storing in Little Endian this is the way by which it is saved where the least significant byte will carry C2 and the most significant byte will carry will be stored in the higher address.

So towards the left side we have the lower address and towards the right side if you have the higher address the LSB -least significant byte is stored in the higher address and it is just vice versa in the case of big Endian. Having seen how multi byte words are being stored, the next aspect that we would like to review inside memory is about alignment of data. You know that memory consists of sequence of bytes.

Now, when you are going to store words, some word or 1 byte, some data may be more than 1 byte say 2 bite, some data may be 4 byte can I save the data immediately after the previous data. No, generally all memory are aligned and based upon what kind of alignment we use new data can start only at specific locations.

(Refer Slide Time: 22:04)



Byte / Word Alignment

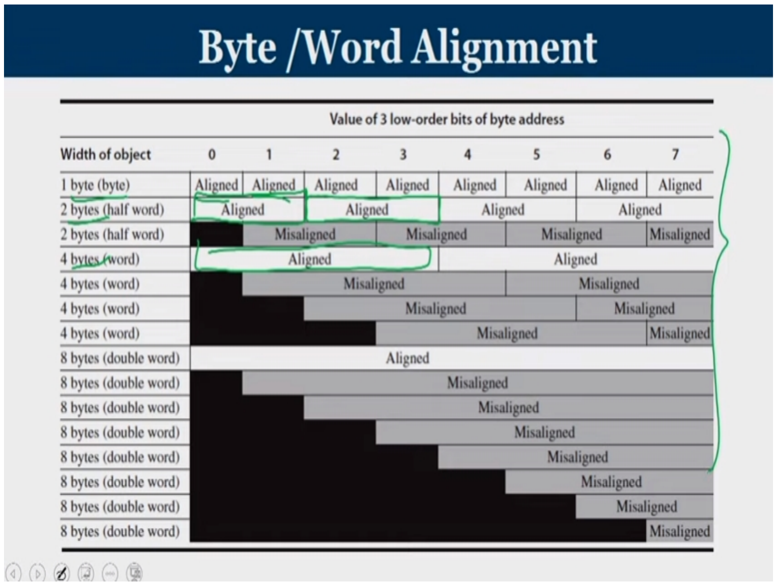
Consider this slide where in the first row shows align to 1 byte. Alignment to 1 byte means that that a data can start at any byte. There is no restriction at all, every byte, you can begin a new data. So the pink color is a 1 byte data. Whereas the green is a 2 byte data and the blue is another 1 byte data. So I can store data at the beginning of every byte. So the first 1 byte is 1 data and then you have 2 byte data that can be immediately stored after the first byte.

You will understand the difference once you go for an alignment of 2 bytes and alignment of 2 bytes means data can start only at words whose byte addresses are multiples of 2. So I can start the data at zeroth byte, second byte, forth byte, sixth byte, eighth byte like that, that is called alignment of 2. If my first that is 1 byte let us say I am going to store at byte number 0, but byte number 1 I cannot store the data because every new data starts at multiples of 2 bytes.

So, the data which is green, which is of size 2 bytes will start only at 2 and it takes 2 bytes. Now the blue one is another data which is of 1 byte storage and it can easily start at 4 because 4 is permitted. If at all you have any new data that is coming the new data can be kept only at 6 such as 4 byte new data that is coming it will get stored in byte number 678 and 9 that is called alignment to 2.

Similarly, if you go for an alignment of 4 then data can start only at memory addresses which are multiples of 4. So after the 1 byte read data, which is at byte number 0, byte number 1, 2 and 3 we cannot store a data. The next new data will start at byte number 4. Another new data will start byte number 8. This is called memory alignment and alignment can be in the order of 1 byte, 2 byte, 4 byte etc.

(Refer Slide Time: 24:09)

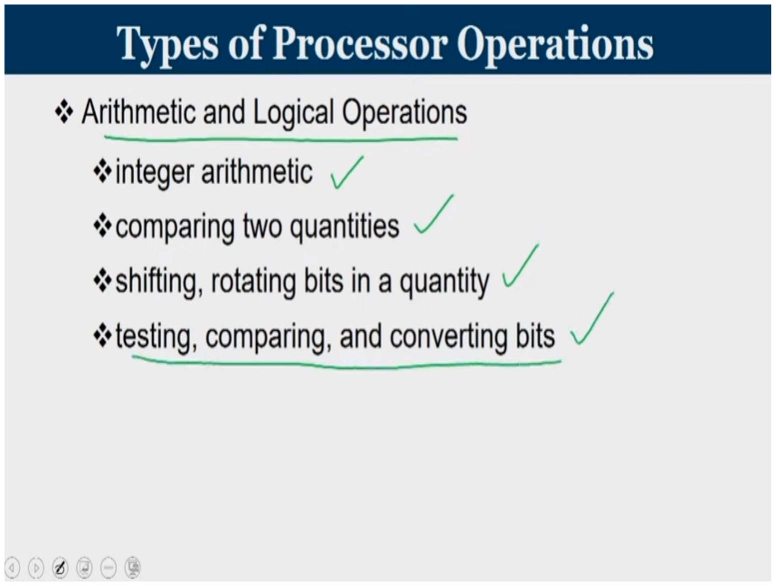


Byte /Word Alignment

	Value of 3 low-order bits of byte address							
Width of object	0	1	2	3	4	5	6	7
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)	■	Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)	■	Misaligned				Misaligned		
4 bytes (word)	■	■	Misaligned				Misaligned	
4 bytes (word)	■	■	■	Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)	■	Misaligned						
8 bytes (double word)	■	■	Misaligned					
8 bytes (double word)	■	■	■	Misaligned				
8 bytes (double word)	■	■	■	■	Misaligned			
8 bytes (double word)	■	■	■	■	■	Misaligned		
8 bytes (double word)	■	■	■	■	■	■	Misaligned	
8 bytes (double word)	■	■	■	■	■	■	■	Misaligned

This slide also shows how alignment is being done. So, if you have a 1 byte alignment, every byte you can start a new data if it is 2 byte then if the first data should be there in byte 0 and byte 1 next to data can be in byte 2 and by 3. So, if you look at every new data starts at multiples of 2 and if you are going for a 4 byte data, then data can start only at multiple of 4. Any other thing is misaligned data that is what you see out here.

(Refer Slide Time: 24:40)

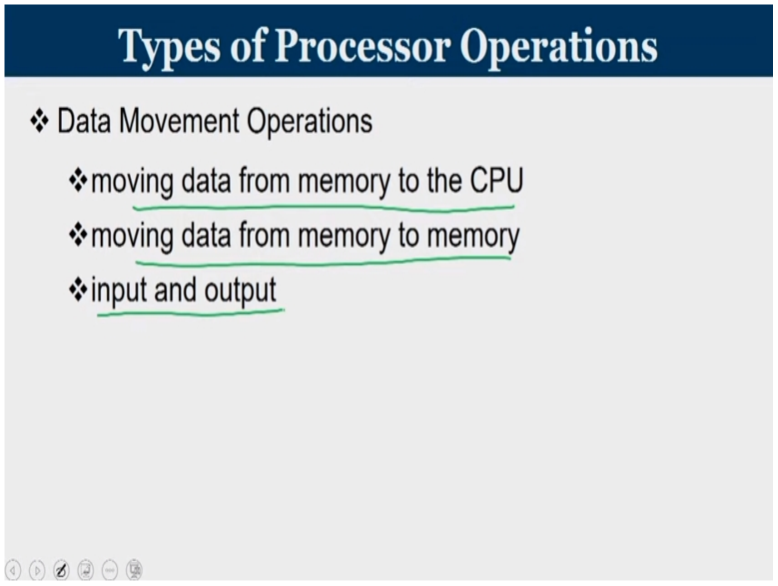


Types of Processor Operations

- ❖ Arithmetic and Logical Operations
  - ❖ integer arithmetic ✓
  - ❖ comparing two quantities ✓
  - ❖ shifting, rotating bits in a quantity ✓
  - ❖ testing, comparing, and converting bits ✓

Let us know try to understand what are the basic operations that a processor can do. It is called basically classification of instructions. The first and foremost height of operations that basically microprocessors do is on arithmetic and logical operations wherein it performs integer arithmetic, logical arithmetic, comparison of quantities, shifting and rotation of numbers testing, comparing and converting bits. These are the basic operations that come under arithmetic and logic operations.

(Refer Slide Time: 25:13)

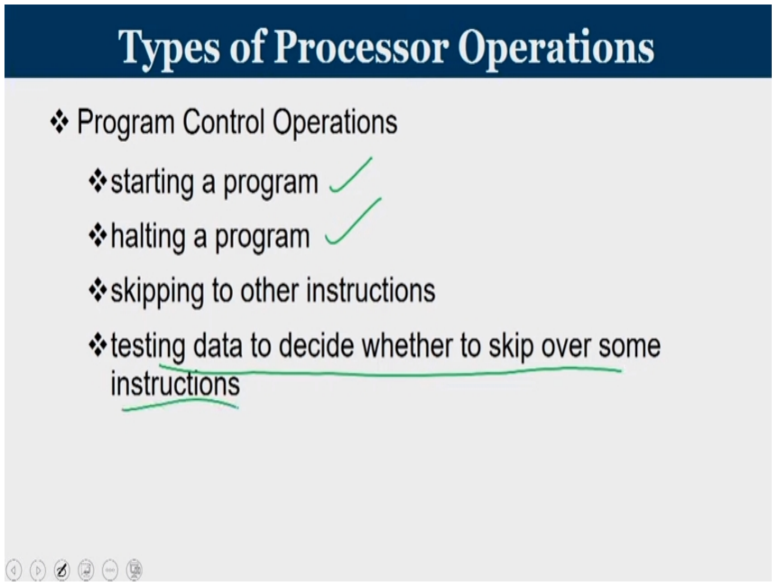


Types of Processor Operations

- ❖ Data Movement Operations
  - ❖ moving data from memory to the CPU
  - ❖ moving data from memory to memory
  - ❖ input and output

The next category of operation that microprocessor do is all about moving of data, moving data from memory to the CPU and moving data between memory and moving data from CPU back to memory and from input and output. So, when data is located, either in memory and if the data is needed in CPU, this move instruction or data transfer instruction will help the microprocessor to bring the data which is in a different unit. Similarly, it is also used for getting and putting data from peripheral devices also.

(Refer Slide Time: 26:04)



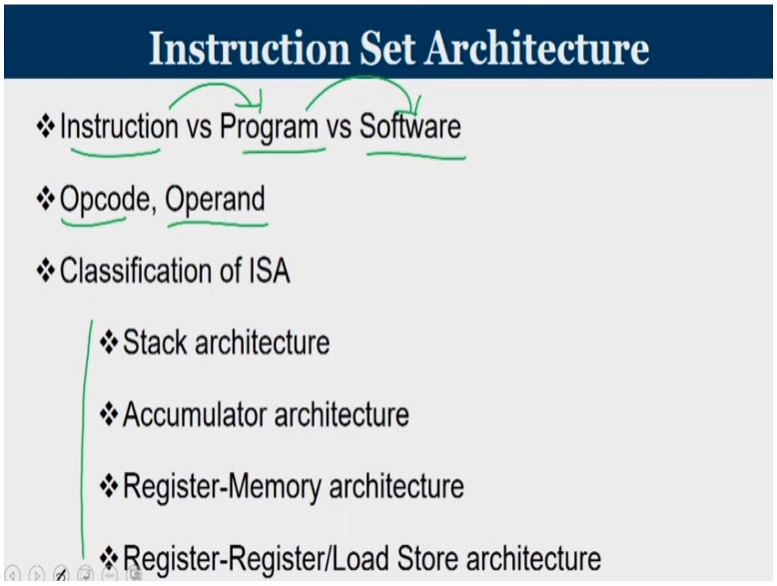
Types of Processor Operations

- Program Control Operations
  - starting a program ✓
  - halting a program ✓
  - skipping to other instructions
  - testing data to decide whether to skip over some instructions

The next category of operations comes as program control operations, which is basically used for starting of a program halting of a program. And let us say, if I wanted to skip you instructions, we use conditional statements and these are also known as program control operations and to test to decide whether you how to skip few instructions or not. So, any kind of thing that is going to transfer the control they come under the program control operations.

We will now look into what is called instruction set architecture, we know that instruction is the basic unit of operation. So generally when you take a task to be done, the task has to be represented as a sequence of instructions.

(Refer Slide Time: 26:47)



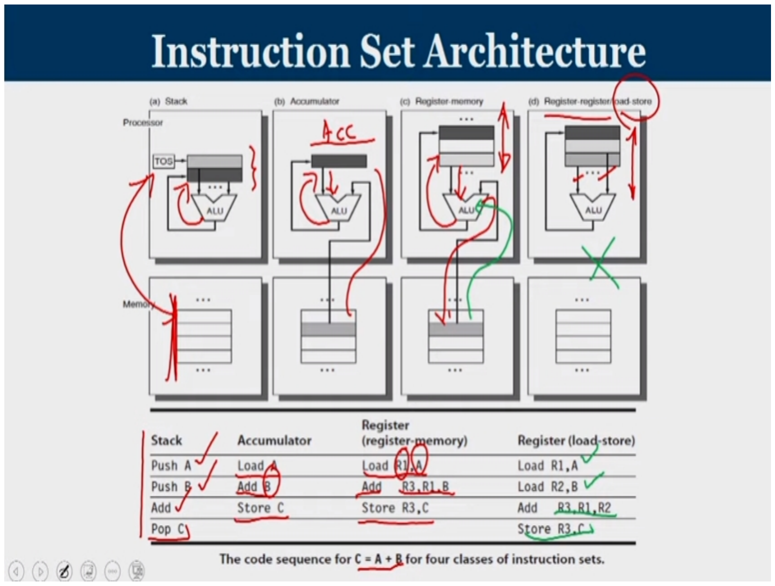
Instruction Set Architecture

- Instruction vs Program vs Software
- Opcode, Operand
- Classification of ISA
  - Stack architecture
  - Accumulator architecture
  - Register-Memory architecture
  - Register-Register/Load Store architecture

So, you know what an instruction program and the software is multiple instructions combined together to form program and multiple programs combined together to form software which will basically is a big task. There is OP code and Operand like what we have mentioned OP code specifies what needs to be done operand specifies where it needs to be done. And the set of all possible instruction that a processor can do it is known as instruction set architecture.

Based on where the operands are, instructions set architecture are classified into four stack architecture, accumulator architecture, register memory architecture, and register-register architecture. Registered-register architecture is also known as load store architecture.

(Refer Slide Time: 27:38)



Instruction Set Architecture

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

The code sequence for C = A + B for four classes of instruction sets.

We will now see what is the difference between these instructions set architectures. The first architecture is known as stack architecture. In the stack architecture, we have the arithmetic logic unit, which are connected only to a system stack. That means the operands of any arithmetic or a logical operation can be from processor stack only. Generally we have the memory where in your data is located.

From the memory you have to use push and pop operations to transfer data between the system stack and the memory. And then when an operation is to be done. In the case of a stack architecture, we will not specify where the operands are. Operands are by default located in the top two elements of the stack. Consider the case that we are going to have

$$C = A + B$$

a basic operation to be done, where C, A and B are memory locations.

In the case of a stack architecture, if you wanted to perform

$$C = A + B$$

our first job is to make sure that A and B are transferred into the stack. And the operation to transfer a memory location to the stack is called push. First, the value A is pushed onto the stack and then value B is pushed on to stack. Now the stack contains both A and B and then we perform ADD operation. In the case of a stack architecture, like what I mentioned, we are not going to mention what is the operand only the Opcode is mentioned and operands are always there in stock.

So add means take two elements from the stack, perform add operation and store the result back in the stack, you can see that the result is stored back in the stack. So after the add operation, the result that is

$$A + B$$

that is available in the stack. Now whatever is there in the top of the stack, I am going to push into the C where i will get the result there in C. So basically in stack architecture, whatever it is the operand that has to be transferred to the stack.

And then the appropriate operation has to be called. Moving on to accumulator architecture. The peculiarity accumulator architecture is you do not have a stack inside you the processor instead, you have one register, which is known as accumulator and accumulator is connected to one of the inputs of ALU and to the output, you can see that it is connected to one of the input of ALU and one the output of ALU as well.

So, when you wanted to perform an operation, if it is a binary operation, the first input should be present in accumulator, the second input I can mention and it can come from memory. So, accumulator is connected to memory, but the result always goes to accumulator and we use load instruction to transfer the content of a memory location to accumulator. Similarly, we use a store instruction to transfer the content of accumulator back to memory.

In order to do the same instructions

$$C = A + B$$

first we load A. Load A means content of memory location A is transferred to accumulator and then we perform add B value perform add B the content of accumulator that is A is added with the content of memory location B. So always the operand that you specify in accumulator architecture is the memory location add B and then B will perform a store operation.

So, in the case of an accumulator architecture, the result of ALU operation will also be reflected inside accumulator and store C means, it will transfer the content from the accumulator back into the location C. The third category is called register memory architecture, in register memory architecture, similar to what we have seen in accumulator architecture, we have registers inside your CPU, we have many such registers inside CPU.

It is not restricted to one register like accumulator and these registers are named as r1, r2, r3 and so on. Now, when you use load instruction, we have to clearly specify to which register you have to load and from which memory location you have to load. So, the loads are now associated with two operands, one is a destination register and the other one is the source memory location. And when you perform an add operation, the resultant should be there inside a register one of the operand will be register, but the second option can be from anywhere in memory.

So

add r3,r1, B

means content of r1 is added with content of memory location B and the resultant is stored in the registered r3 and to transfer the result that is already available in register r3, we use

store r3, C

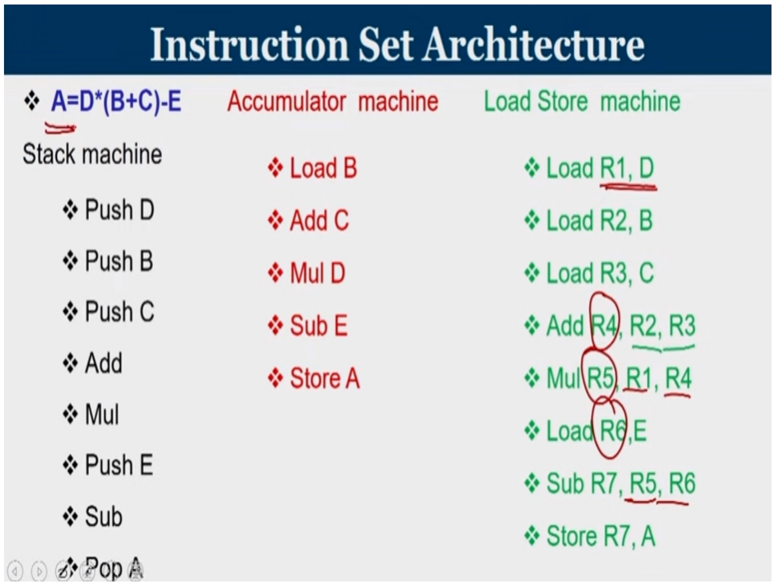
wherein at the end you get the final result in the variable C.

Moving on to the last architecture which is known as register-register architecture or load store architecture. Here also we have a couple of registers inside your processor, but ALU is connected only with these registers, ALU is not connected with any of the memory locations like what we have seen in register memory architecture. So that is the basic difference between them in the case of a register memory architecture, you have a path to the ALU from memory, whereas in this case, such a kind of a path is not existing there.

Now, when go to the load store architects to understand the operands of an arithmetic logic operation can only be registers. So, you have an load operation, wherein the first operand is moved to register r1 another load operation wherein the second option is moved to register r2 perform add operation, so that the result is now in register r3 and transfer the result. So, that is all about register or load store architecture.

Moving further now, we will try to have a simple exercise example, wherein we can see the difference in terms of number of instructions needed, if you are going to carry out simple operation in stack machine and the other architectures.

(Refer Slide Time: 33:45)



Instruction Set Architecture

- A=D*(B+C)-E

Stack machine	Accumulator machine	Load Store machine
Push D	Load B	Load R1, D
Push B	Add C	Load R2, B
Push C	Mul D	Load R3, C
Add	Sub E	Add R4, R2, R3
Mul	Store A	Mul R5, R1, R4
Push E		Load R6,E
Sub		Sub R7, R5, R6
Pop A		Store R7, A

So, let us consider

$$A = D * B + E$$

as the operation to be done let us first see how we will do in stack machine. So, we know that we have to perform a push operation on D, wherein D is first transferred and then you push B and C. So now B and C are in the top performing an add operation, so

$$B + C$$

will be done and the result will be stored in the stack. And then we will do a multiplication operation such that D and the sum of

$$B + C$$

will be multiplied that result will be there and then push E and then you perform the subtraction.

And finally, you pop the value back into A this is how

$$A= D * B + C – E$$

is being carried out. Let us look into an accumulator machine how are you going to perform the operation. In the case of an accumulator machine you have to load the value B. So B will be going into the accumulator and then you perform add C. So in that content of accumulator, which is already B is added with the memory location C and the result is available in the accumulator.

And then we can perform multiplication of D. So already you have the result we will be there is accumulator. So at this point accumulator carries B + C, now you multiply with the D.

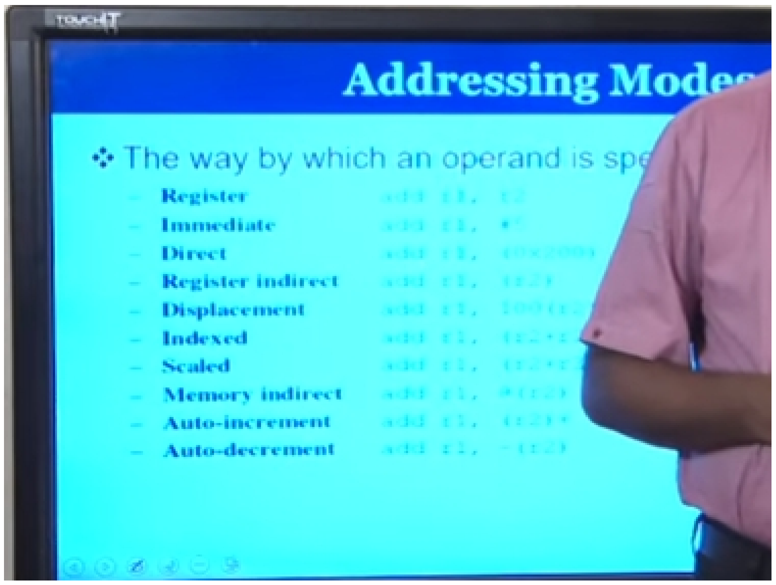
Now

D * B + C

available in the accumulator. And then you can perform subtraction of E followed by storing the result in A.

So when you move to load store machine, first we will be loading the value of D into register R1. And then the value of B loaded to R2 and value of C is loaded to R3. So now the values of B and C is already there in register, we can perform an add operation on R2 and R3 which is already this mentioned, R2 and R3 values are there. So R2 and R3 are now added and the result is stored in R4 now once you have the result in R4 then whatever is your D which is that an R1 can be multiplied to the result of R4 to get the result in r5 and then you load the value of E into R6 and perform the subtraction operation such that the content of R5 and R6 are there and you subtract R6 from R5 to store the result in R7 and then you store the final result which is available in R7 back into our register A which stores a final result.

(Refer Slide Time: 36:13)



Addressing Mode

- The way by which an operand is spe
  - Register — add r1, r2
  - Immediate — add r1, #5
  - Direct — add r1, (0x200)
  - Register indirect — add r1, (r2)
  - Displacement — add r1, 100(r2
  - Indexed — add r1, (r2+r
  - Scaled — add r1, (r2*r
  - Memory indirect — add r1, @(r2)
  - Auto-increment — add r1, (r2)+
  - Auto-decrement — add r1, -(r2)

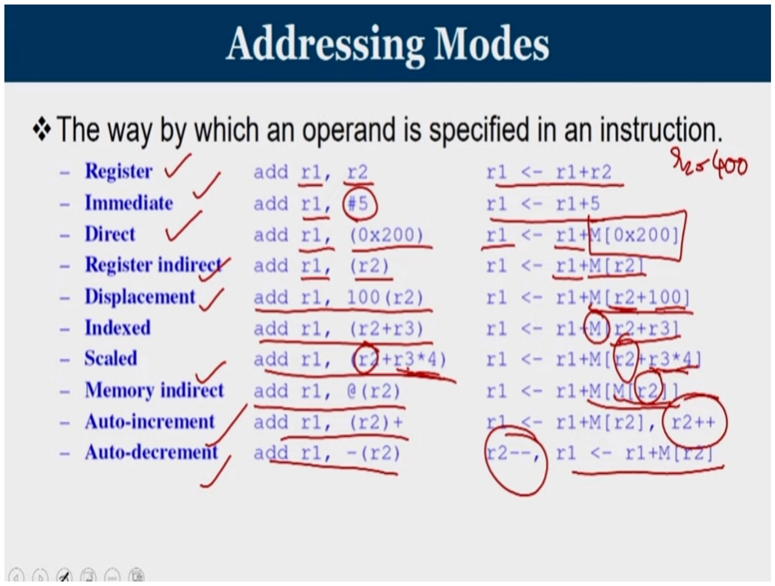
The next section what we are going to address today is about addressing mode is the way by which the operand is specified as part of the instruction, we will try to understand addressing mode with the help of an example. Consider the case that I wanted to call a student and solve an exercise on the board. I can use many ways to call the student. Let us say I wanted to mention the name of the student directly then that is known as immediate addressing mode.

Consider the case that the students name is Ram, I can tell Ram please come to the stage and solve the exercise on the board. So Ram please come, the name Ram already mentioned as the instruction I am giving it is called immediate way of mentioning the operand. The second thing is I am going to specify the role number of Ram. So the address of Ram is been mentioned. If I know Rams number is 20 I will tell whose roll number 20 please come and then solve it in the board.

So rather than mentioning it does Rams name I mentioned his number. Similarly, I can put it in a different way I can actually keep this role number 20 inside a box, so I will tell I want the student to come and solve it for me on the board. His roll number is kept inside this box. So, somebody can open that box and the box contains a chit with a number 20 written and 20 is Ram and RAM will come and do.

When I directly mentioned the roll number it is called a direct mode of addressing when I specify it is an indirect way then it is called indirect mode of addressing. Similar to that, when you specify instructions, the operand can also be mentioned in different ways that is what is known as addressing mode.

(Refer Slide Time: 38:22)



Addressing Modes

❖ The way by which an operand is specified in an instruction.

r2=400

– Register	add r1, r2	r1 <- r1+r2
– Immediate	add r1, #5	r1 <- r1+5
– Direct	add r1, (0x200)	r1 <- r1+M[0x200]
– Register indirect	add r1, (r2)	r1 <- r1+M[r2]
– Displacement	add r1, 100(r2)	r1 <- r1+M[r2+100]
– Indexed	add r1, (r2+r3)	r1 <- r1+M[r2+r3]
– Scaled	add r1, (r2+r3*4)	r1 <- r1+M[r2+r3*4]
– Memory indirect	add r1, @(r2)	r1 <- r1+M[M[r2]]
– Auto-increment	add r1, (r2)+	r1 <- r1+M[r2], r2++
– Auto-decrement	add r1, -(r2)	r2--, r1 <- r1+M[r2]

This slide shows you different ways in which I can perform the operation, so consider the case that I wanted to add 2 numbers. If the 2 numbers what I am going to discuss are all part of the same register, I can use

add r1, r2

meanings is

r1 = r1 + r2

it is known as register addressing mode. Second one is I want to go directly add r1 with 5 where 5 is my data. So

add r1, #5

is one general representation used to mention immediate values that meaning is

r1 = r1 + 5.

The third one is direct addressing mode where content of location 200 has to be added with r1 so

r1 = r1 + content of memory location 200

that is called direct addressing mode. Second one is called register indirect addressing mode add r1 at r2, meaning is you go to r2. Let us say r2 values is value is 400 go to memory location 400 that has to be added with r1 in order to get the result that is called register indirect.

Then we have something called memory indirect add r1 at r2 where content of r2 is taken that will give you memory location go to that memory location that will give another number go to that is called double in directing it is called memory indirect. Then there is displacement addressing mode where I specifying

add r1, 100(r2)

go to r2 get a location add 100 to it then that is your effective memory address and go to that memory location in order to access the operand.

Index addressing mode is r2 and r3. So, both are been added together and then you are going to consider that as a memory address and then you have scaled addressing mode. So, r2 is multiplied with the 4 and then you added with another register. So effectively in this case, you have r3 that is acting as the index r3 into 4, content of r3 into 4 and then there is a base of r2. Then we have auto increment and auto decrement addressing modes where the value of memory

location mentioned in r2 is taken and automatically r2 incremented. So that next time when in go, it will point to the next location.
Similarly, when you go for auto decrementing mode, first you decrement the value of r1 and then you are going to take it. So, these are all different ways by which we can address the different operands based upon the convenience that has to be used in mentioning the instructions.

So, with this, we come to the end of the first lecture, we will try to have a quick recap on what we have done in this lecture.

The basic objective of the first lecture was to get a quick recap on basic computer organization concepts. We started with what is instruction execution what are the various sub operations associated with execution of an instruction. After completing that, we then went to see what is the internal organization the specific registers MAR, MDR, program counter PC then IR instruction register and then we saw how the fetching happens, how instruction is decoded? How appropriate signals are being generated.

We saw the roll of decoders in the case of addressing in memory location and then we have seen different types of instruction that the processor can do followed by instruction set architecture. Classification of instructions set architecture based upon where operands of an ALU operation can be. Like stack architecture, accumulator architecture, register memory architecture and load store architecture. We have seen an example also.
And we concluded the lecture by having a detailed discussion on what are the various addressing modes? Kindly go through the lecture videos if you are not familiar with compute organization concepts, feel free to post in your queries. And slowly we will learn this text books are also there that has been mentioned. Kindly go through the textbook, the specific chapters in order to get more grip into the subject.

I hope that you will enjoy the subject in the rest of the lectures while we move on to advanced computer architecture. Make sure that you clarify concepts in that week itself by posting it in the queries. We will be happy to address your doubts and clarifications. Wish you all a happy learning. Thank you