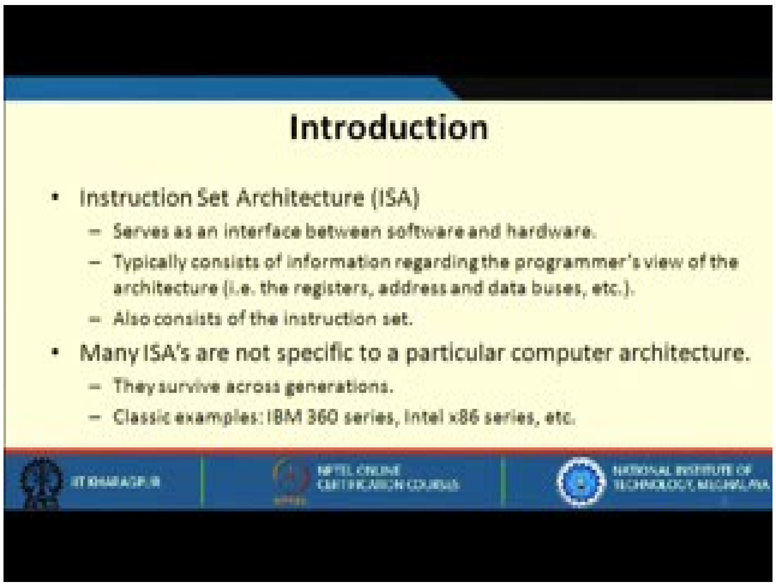
Computer Architecture and Organization
Prof. Kamalika Datta
Department of Computer Science and Engineering
National Institute of Technology, Meghalaya

Lecture - 05
Instruction Set Architecture

Welcome to the fifth lecture, that is, instruction set architecture. Here we are looking into a computer system from programmer's point of view; like the assembly programmer -- what as an assembly programmer will have the view of the computer system.

(Refer Slide Time: 00:47)



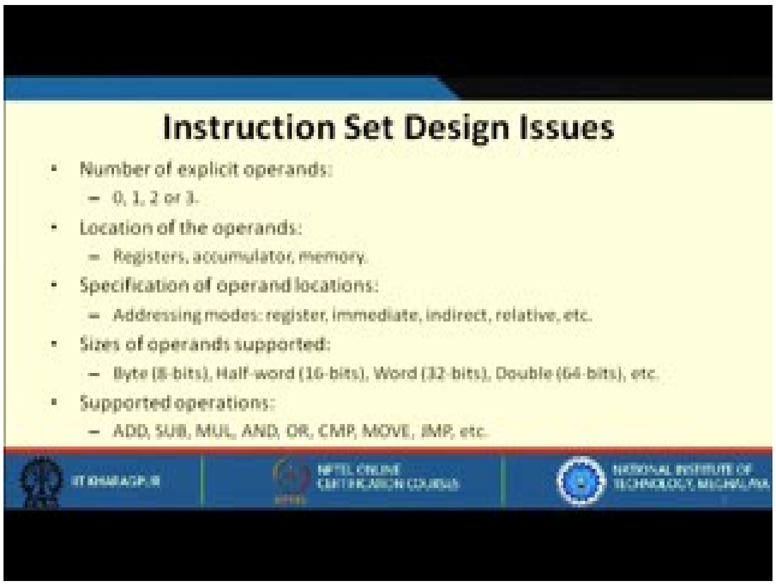
Introduction

- Instruction Set Architecture (ISA)
  - Serves as an interface between software and hardware.
  - Typically consists of information regarding the programmer's view of the architecture (i.e. the registers, address and data buses, etc.).
  - Also consists of the instruction set.
- Many ISA's are not specific to a particular computer architecture.
  - They survive across generations.
  - Classic examples: IBM 360 series, Intel x86 series, etc.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Instruction set architecture serves as an interface between the software and the hardware. Here by hardware we mean the processor system, like we need to know what all registers we have and what kind of features are supported by those registers. Typically consists of information regarding programmer's view of the architectures, that is, the registers, address and data buses, etc. and it also consists of the instruction set. Now, many instruction set architectures are not specific to a particular computer architecture and they survive across generations; like if you see Intel X86 series, across generations it has got no change.

(Refer Slide Time: 02:16)



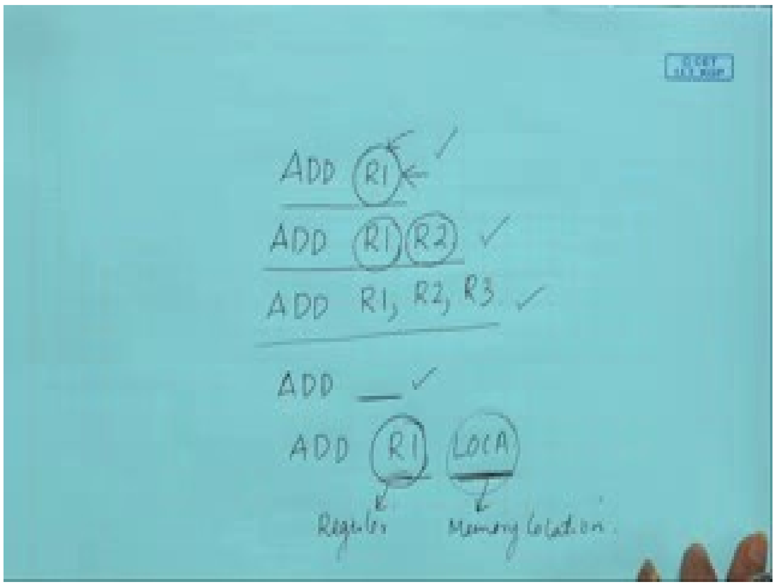
Instruction Set Design Issues

- Number of explicit operands:
  - 0, 1, 2 or 3.
- Location of the operands:
  - Registers, accumulator, memory.
- Specification of operand locations:
  - Addressing modes: register, immediate, indirect, relative, etc.
- Sizes of operands supported:
  - Byte (8-bits), Half-word (16-bits), Word (32-bits), Double (64-bits), etc.
- Supported operations:
  - ADD, SUB, MUL, AND, OR, CMP, MOVE, JMP, etc.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us see what are the important instruction set design issues that should be taken into consideration. First is number of explicit operands. By operands, what do we mean?

(Refer Slide Time: 02:34)



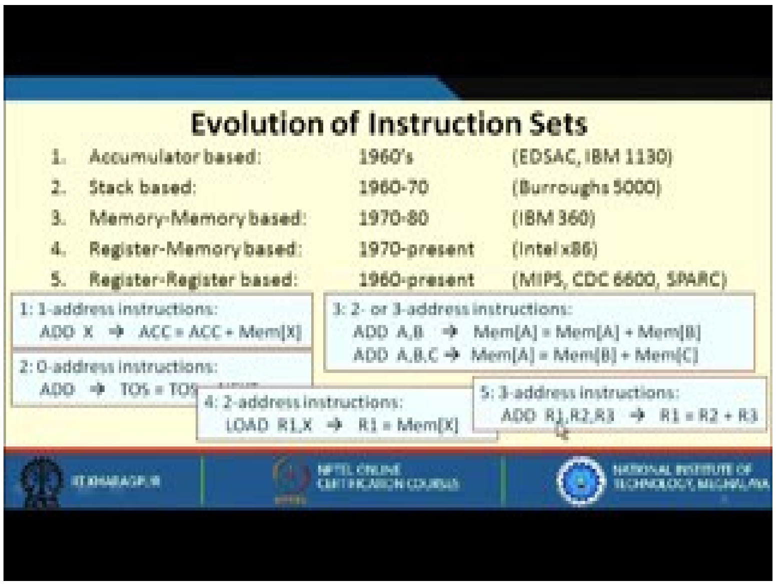
Let us say we have an instruction ADD. We can have several instruction variations, like ADD R1; ADD R1, R2; ADD R1, R2, R3. By number of explicit operands here we have a single operand, two operands, and three operands specified in the instructions. And if we do not specify any operand, then we implicitly take some operands for this operation. We will also see that. So, there can be zero address instruction, there can be one address instruction, there can be two address instruction, or there can be three address instruction.

Let us now see the location of the operands. By location of operand what do we mean is: ADD R1,LOCA -- R1 is a processor register, so this is within processor; LOCA is a location in memory. So, by location of the operands, we mean either it is in a register or it is in accumulator or it is in memory. We will see what is an accumulator in course of time. We have to specify to the computer that the first operand is a register, and so you have to look into a register to get the value. This operand will be a register address where you have to go and see the value.

Now, in LOCA you have to specify that this is a memory location and you have to go to that memory location to access this value. So, addressing mode is the way to specify the operands in your instruction. There can be various addressing modes like register, immediate, indirect, relative, etc, we will see in detail later.

Now, we talk about the size of the operands supported. It can be a byte, it can be half-word, it can be a word, it can be a double, etc. By supported operation we mean that how many operations we can specify specify and what are the various types. When I say ADD, MUL, SUB, these are all arithmetic operations. I can also tell you some other operations like MOVE -- these are data transfer operation, LOAD/STORE are also data transfer operations. And there can be many other kinds of operation that we will see, but by supported operation we mean that how many kinds of operation you are supporting in your architecture.

(Refer Slide Time: 07:25)



Evolution of Instruction Sets

1.	Accumulator based:	1960's	(EDSAC, IBM 1130)
2.	Stack based:	1960-70	(Burroughs 5000)
3.	Memory-Memory based:	1970-80	(IBM 360)
4.	Register-Memory based:	1970-present	(Intel x86)
5.	Register-Register based:	1960-present	(MIPS, CDC 6600, SPARC)

1: 1-address instructions:
ADD X → ACC = ACC + Mem[X]

2: 0-address instructions:
ADD → TOS = TOS

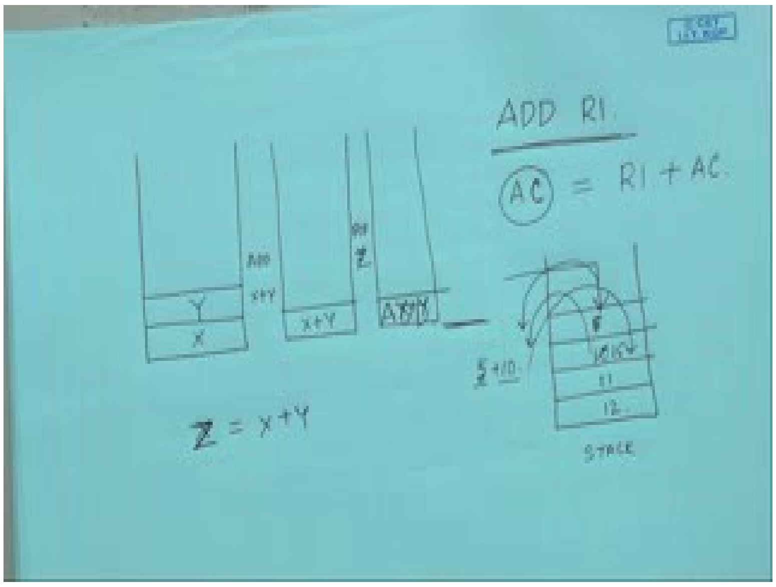
3: 2- or 3-address instructions:
ADD A,B → Mem[A] = Mem[A] + Mem[B]
ADD A,B,C → Mem[A] = Mem[B] + Mem[C]

4: 2-address instructions:
LOAD R1,X → R1 = Mem[X]

5: 3-address instructions:
ADD R1,R2,R3 → R1 = R2 + R3

Now, this actually shows the evolution of instruction set. Initially in 1960s we were having accumulator based systems. By accumulator-based system let me tell what it is.

(Refer Slide Time: 07:50)

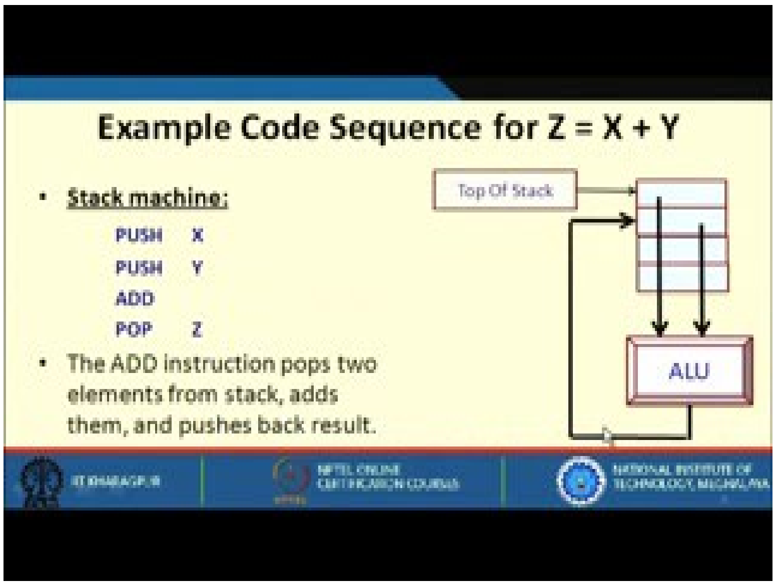


When I write ADD R1; by this R1 will be added with what? R1 will be added with a register present in your processor called **accumulator**. So, accumulator is a register which if we have an instruction like ADD R1; by default, the value of R1 is added with accumulator and the result is stored back in accumulator itself.

During 1960s to 70s another kind of instruction set emerged that is stack based. What do you mean by stack based? In stack-based architecture, a portion of memory called stack is made available. Data on the top of the stack can be accessed. If you just say ADD, and do not specify any operand here, then by default the first two elements of the stack will be taken out, that is 5 and 10, will be added and stored back here; so this becomes 15. By this what we mean is that we are doing some operation where we are loading the data, we are storing the data in the stack, and then we are performing the operation where we need not have to specify any operand --- by default the operands are taken from the stack. So, here it is a 0-address instruction.

Next comes memory-memory based instructions in 70s and 80s, and the representative system is IBM 360 which has both two-address and three-address instructions. ADD A,B --- where the data from memory location A is added with data from memory location B and the result is stored back in A. Now, we also have register-memory based systems where one operand will be a register and one operand will be a memory location. So, LOAD R1,X. What do you mean by that? Load from memory location pointed by X into R1. Similarly, we can also have STORE, store the value of R3 into some other location. And finally, we have three-address instructions where we are specifying three addresses and what does it do? Here R1 stores the value of R2 and R3, the result of the addition is stored in R1. So, in a single step, we can do this.

(Refer Slide Time: 12:45)



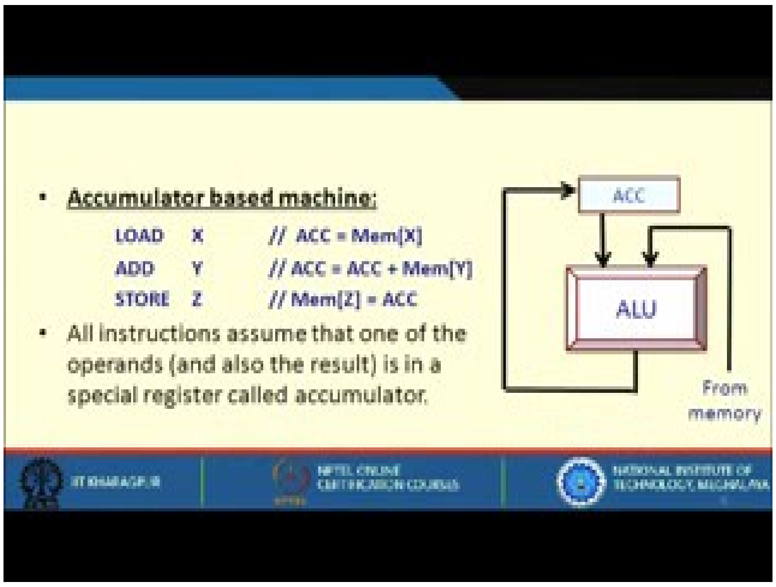
Example Code Sequence for Z = X + Y

- **Stack machine:**
  ```
  PUSH   X
  PUSH   Y
  ADD
  POP    Z
  ```
- The ADD instruction pops two elements from stack, adds them, and pushes back result.

Top Of Stack
ALU
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, let us see some example code sequence for executing some sample instruction that is Z = X + Y, using the various instruction set architectures that I have told in the previous slide. So, let us consider this stack-based machine. First we have to push X, next we have to push Y, and then both of these are now in top two position of the stack. When we perform this ADD, these two values are taken out, added and stored back in the top of the stack. And finally, when we do pop, then the value from top of the stack is taken out and stored back in Z. So, let me explain here in this way. So, by PUSH X, X is added here; by PUSH Y, Y is added here. And then once we perform ADD, X + Y is added and stored back here. And then when we perform POP Z, then Z is a memory location where we will get the result.

(Refer Slide Time: 14:52)



- **Accumulator based machine:**

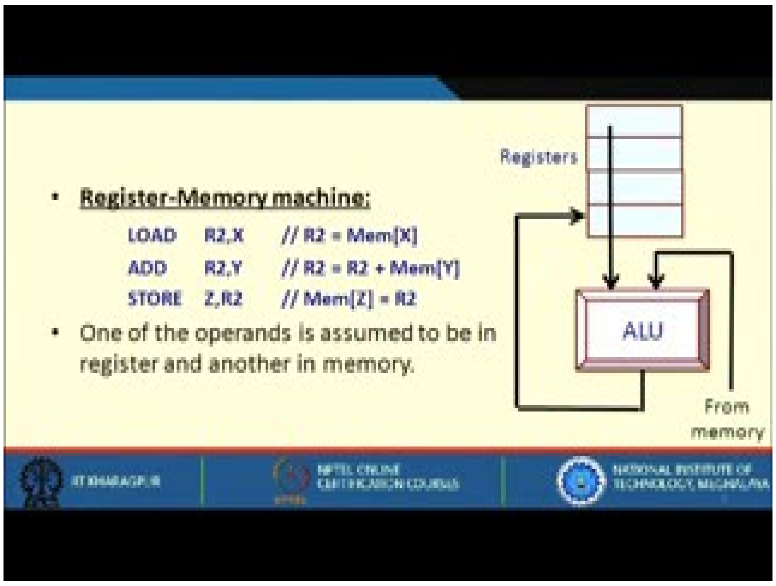
```
LOAD   X    // ACC = Mem[X]
ADD    Y    // ACC = ACC + Mem[Y]
STORE  Z    // Mem[Z] = ACC
```

- All instructions assume that one of the operands (and also the result) is in a special register called accumulator.

ACC
ALU
From memory
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Next, see an accumulator-based system. In an accumulator-based system, if you have to perform the same operation Z = X + Y, then what you have to do? Both X and Y are some values in stored in memory, so you have to load X in accumulator. Then ADD Y will actually add the content of location Y with accumulator and store back the result in accumulator. So, we can see in the ALU one value is coming from the accumulator and another is coming from memory; we are adding these two values and we are storing back the result in accumulator. So, all instructions assume that one of the operands and also the result is in a special register called accumulator.

(Refer Slide Time: 16:06)



- **Register-Memory machine:**

```
LOAD   R2,X   // R2 = Mem[X]
ADD    R2,Y   // R2 = R2 + Mem[Y]
STORE  Z,R2   // Mem[Z] = R2
```

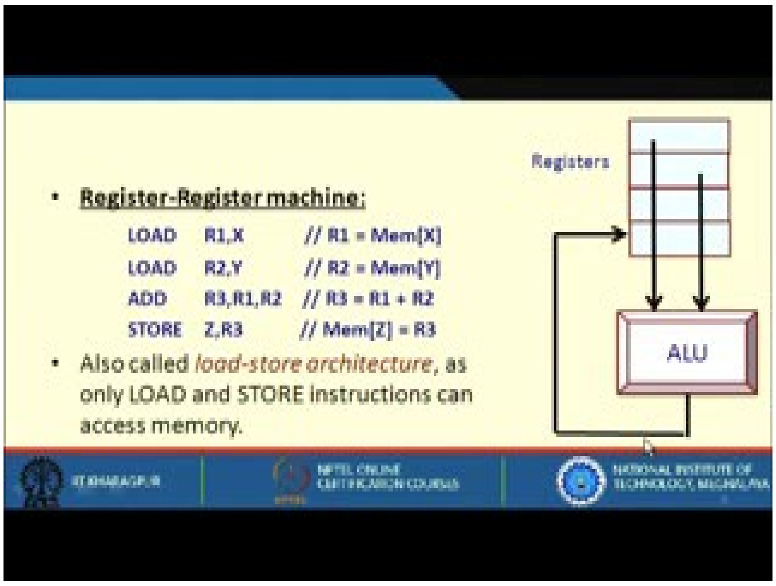
- One of the operands is assumed to be in register and another in memory.

Registers
ALU
From memory
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Next we see register-memory machine. In register-memory machine how this can be performed? LOAD R2,X --- from location X the data will be loaded in R2. When we do ADD R2,Y --- in R2 the content of R2 which was nothing but X will be added with Y and stored back in R2. Finally, we have to store this result R2 in Z.

So, here one of the operands is assumed to be in register and another in memory. So, that is why in the ALU one of the operand is your register. So, one value is coming from register, another is coming from memory and then finally, the result is getting stored here in some register, and finally, the STORE will store back the result in Z memory location.

(Refer Slide Time: 17:34)



- **Register-Register machine:**

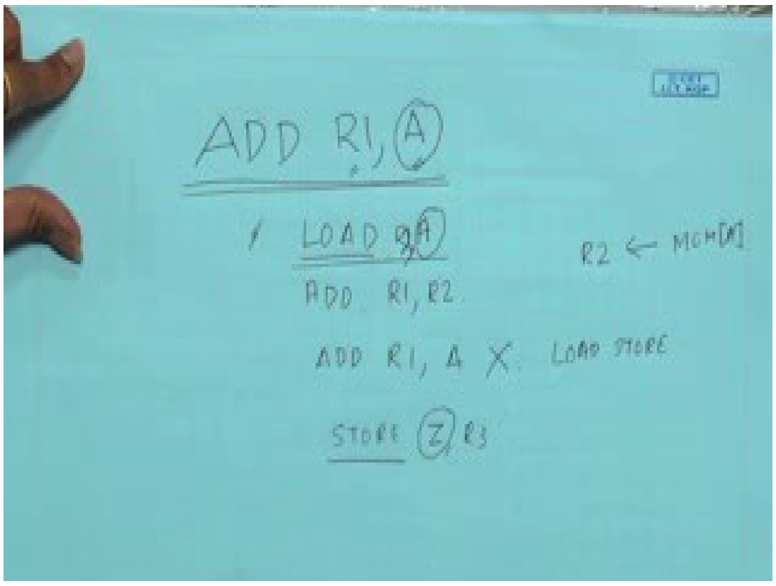
```
LOAD   R1,X        // R1 = Mem[X]
LOAD   R2,Y        // R2 = Mem[Y]
ADD    R3,R1,R2    // R3 = R1 + R2
STORE  Z,R3        // Mem[Z] = R3
```

- Also called *load-store architecture*, as only LOAD and STORE instructions can access memory.

Registers
ALU
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

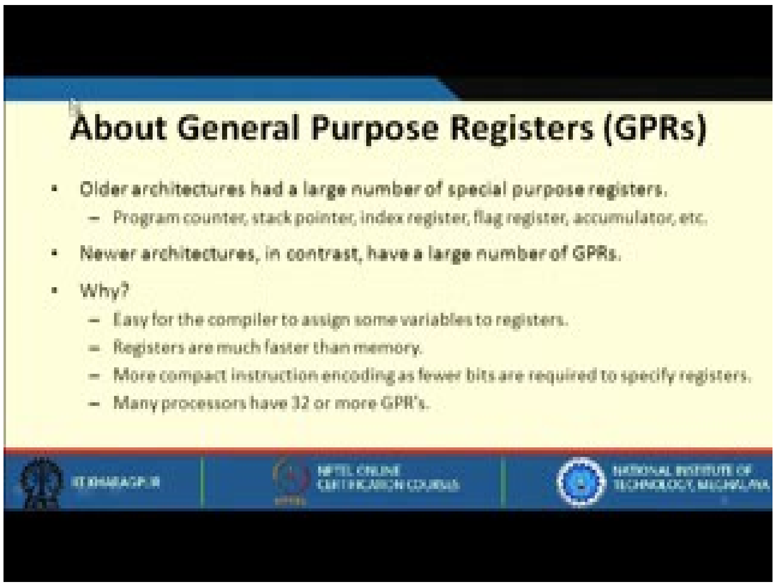
Now, we see register-register machine. So, in register-register machine what you need to do is that you have to load everything into some register first. So, here instead of doing ADD operation or any kind of operation if you want to perform, you have to perform only on registers and not on any memory location; that is why in register-register machine you have to first load all the values of the locations memory location into some register. That is why we are using two back-to-back LOAD. In the first LOAD, value of X will be loaded in R1, and in next the value of Y will be loaded in R2. Now we can add these two registers and store the value in R3. Now, finally, we have to store the result in Z. This kind of architecture is also called load-store architecture. By load-store architecture we mean that only these two instructions LOAD and STORE will be used to access the memory, and no other instructions will be used to access the memory.

(Refer Slide Time: 19:11)



Earlier we have seen that using instruction like ADD R1,A, --- where A is a memory location. In this instruction, we are allowing one register operand and one memory operand; but in LOAD/STORE architecture what will happen, you cannot access this. So, what you have to do, you have to load A using LOAD R1,A. But you cannot do ADD R1,A in LOAD/STORE architecture. You can only use load and store to access memory.

(Refer Slide Time: 20:41)



About General Purpose Registers (GPRs)

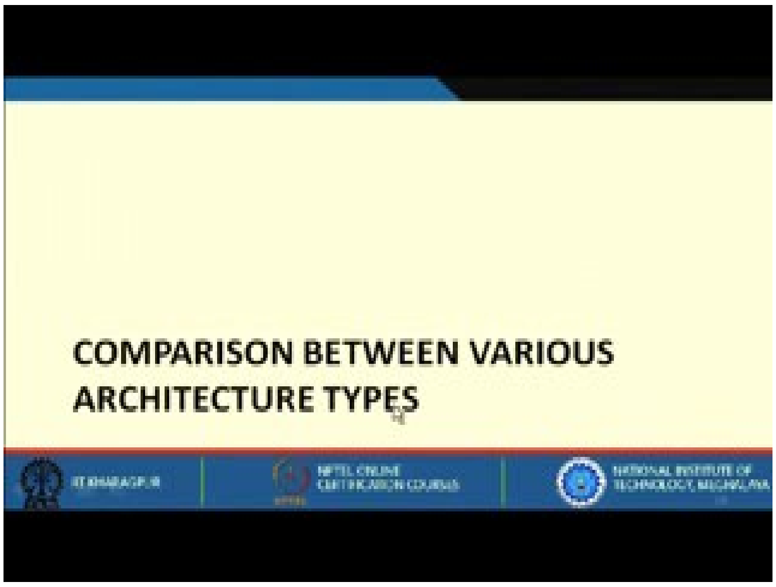
- Older architectures had a large number of special purpose registers.
  - Program counter, stack pointer, index register, flag register, accumulator, etc.
- Newer architectures, in contrast, have a large number of GPRs.
- Why?
  - Easy for the compiler to assign some variables to registers.
  - Registers are much faster than memory.
  - More compact instruction encoding as fewer bits are required to specify registers.
  - Many processors have 32 or more GPR's.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Let us see about the general-purpose registers. If you recall, older architectures have large number of special-purpose register like we talked about program counter, stack pointer, some index register, flag registers, accumulator, etc. But in newer architectures we have more number of general-purpose registers. And instead of using special purpose registers, most of the operations are performed using general purpose registers. And why that is so? The compiler can assign some variables to registers. So, there are so many variables that can be used and they can be assigned it to registers, and registers are much faster than memory. So, once you load the data into the registers and you are performing operation within the register it will be much faster. But first you have to load the data from your memory to register and then only you can perform the operation within register.

More compact instruction encoding is possible as fewer bits are required to specify register. So, the instruction encoding can be much less why they are saying like see you are bringing everything into registers and the registers cannot be unlimited as compared to the memory addresses. Memory addresses a 32-bit memory addresses will have 32-bit, but if you have 40 register or 100 registers how many bits do you require? For 100 register you will require a maximum of 7-bits to encode. So, many processors have 32 or more general purpose registers.

(Refer Slide Time: 22:42)

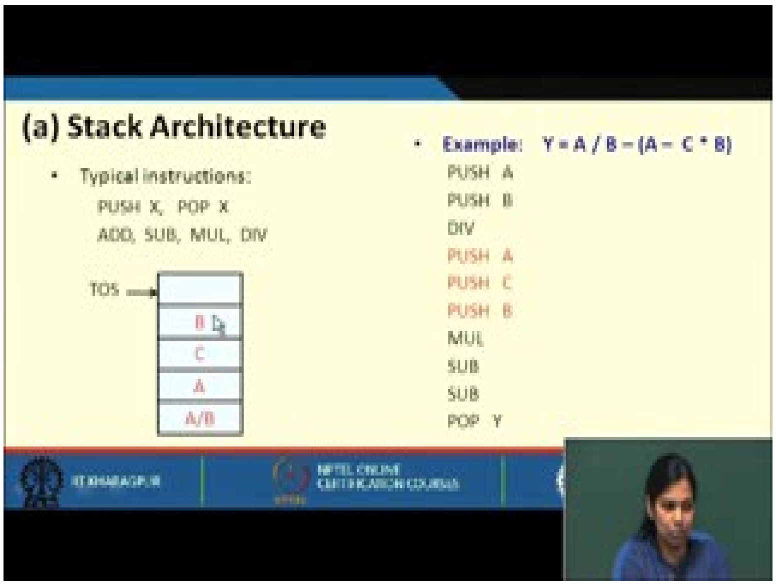


COMPARISON BETWEEN VARIOUS ARCHITECTURE TYPES

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Now, coming to comparison between various architectural types. I have discussed about many machines: stack based, accumulator based, register-register memory. Now, let us do a comparison and figure out that which one is better than the other.

(Refer Slide Time: 23:07)



(a) Stack Architecture

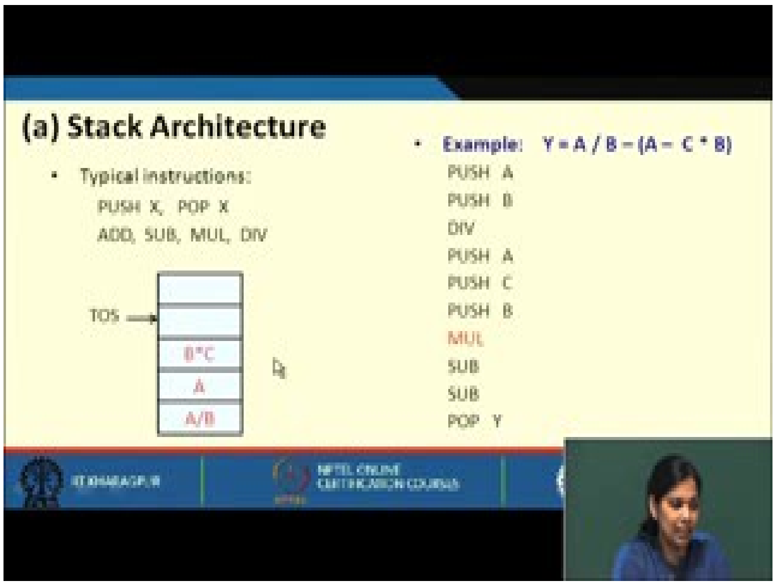
- Typical instructions:
  PUSH X, POP X
  ADD, SUB, MUL, DIV

- Example: Y = A / B − (A − C * B)

```
PUSH A
PUSH B
DIV
PUSH A
PUSH C
PUSH B
MUL
SUB
SUB
POP Y
```

Firstly, we will do that with the example. So, we will be executing this particular instruction. This particular instruction will be executed and we will see that with various architectures how many steps it takes to execute. This particular instruction for stack based what you have to do. So, these are all memory location data. So, what we need to do. So, we have to first PUSH A, because we need to put this value into stack, and then only we can perform the operation. Then we have to PUSH B, then we can specify the operation DIV that will perform that particular operation and store back in the stack. Next we will do PUSH A, PUSH C and PUSH B. We have pushed first A, then B, then C then B why we have done so? Just see we have performed this operation now we have to perform this operation. For doing this operation we will first perform C * B and then A that is why we have first put A and then we have put C and B.

(Refer Slide Time: 24:39)



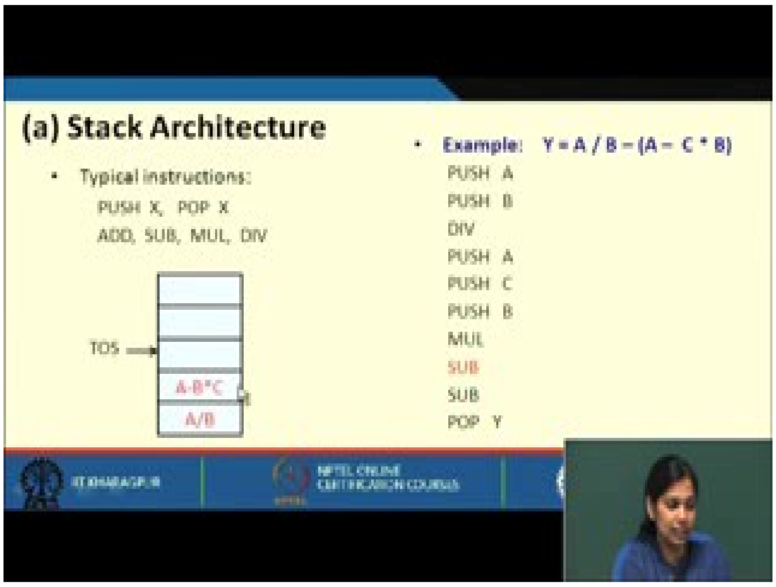
(a) Stack Architecture

- Typical instructions:
  PUSH X, POP X
  ADD, SUB, MUL, DIV

- Example: Y = A / B – (A – C * B)
  PUSH A
  PUSH B
  DIV
  PUSH A
  PUSH C
  PUSH B
  MUL
  SUB
  SUB
  POP Y

Now, when we do the next operation MUL then what will happen B and C will be multiplied and it will be stored back.

(Refer Slide Time: 24:54)



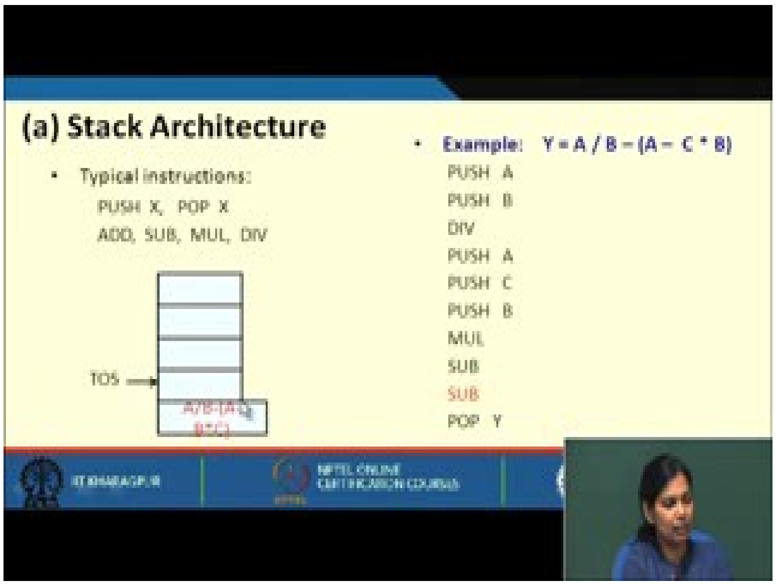
(a) Stack Architecture

- Typical instructions:
  PUSH X, POP X
  ADD, SUB, MUL, DIV

- Example: Y = A / B – (A – C * B)
  PUSH A
  PUSH B
  DIV
  PUSH A
  PUSH C
  PUSH B
  MUL
  SUB
  SUB
  POP Y

And now if we do a SUB then A - B into C will happen. So, we have performed this part, we have already performed this part earlier which is stored in the stack. Now, if you just perform a SUB again then what will happen A / B – (A - B * C) will take place.

(Refer Slide Time: 25:15)



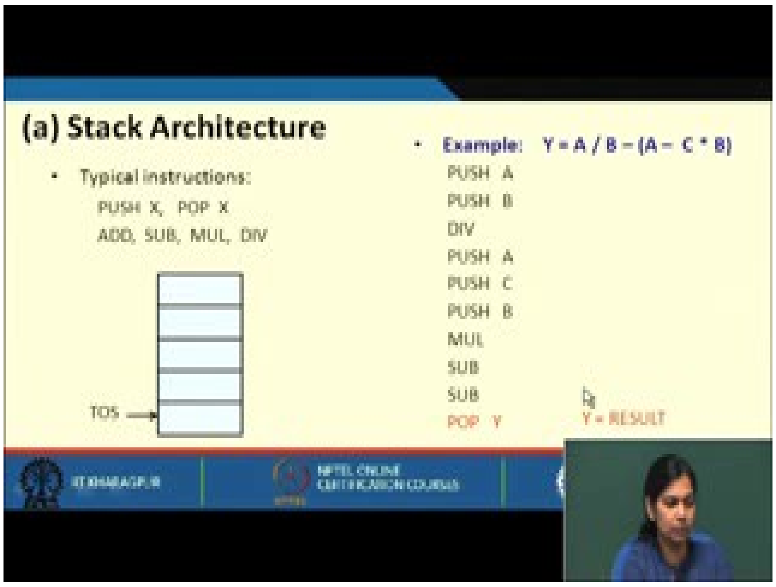
(a) Stack Architecture

- Typical instructions:
  PUSH X, POP X
  ADD, SUB, MUL, DIV

- Example: Y = A / B – (A – C * B)
  PUSH A
  PUSH B
  DIV
  PUSH A
  PUSH C
  PUSH B
  MUL
  SUB
  SUB
  POP Y

So, now in my stack we have A / B – (A - B * C), but finally we have to store this into a memory location Y. So, we are storing this into Y by doing a POP. So, POP will take out the result from top of the stack and put the result into Y.

(Refer Slide Time: 25:33)



(a) Stack Architecture

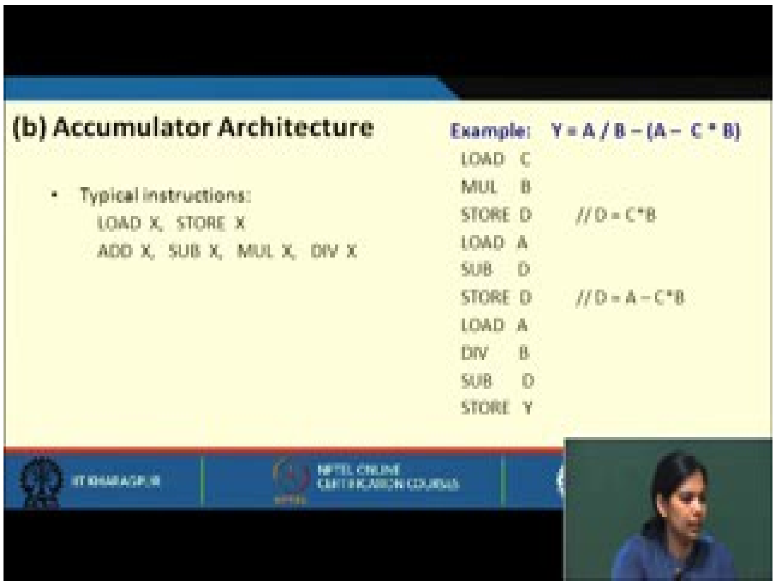
- Typical instructions:
  PUSH X, POP X
  ADD, SUB, MUL, DIV

- Example: Y = A / B – (A – C * B)
  PUSH A
  PUSH B
  DIV
  PUSH A
  PUSH C
  PUSH B
  MUL
  SUB
  SUB
  POP Y

Y = RESULT

So, how many steps basically we took to execute this? We took 10 steps.

(Refer Slide Time: 25:55)



(b) Accumulator Architecture

- Typical instructions:
  LOAD X, STORE X
  ADD X, SUB X, MUL X, DIV X

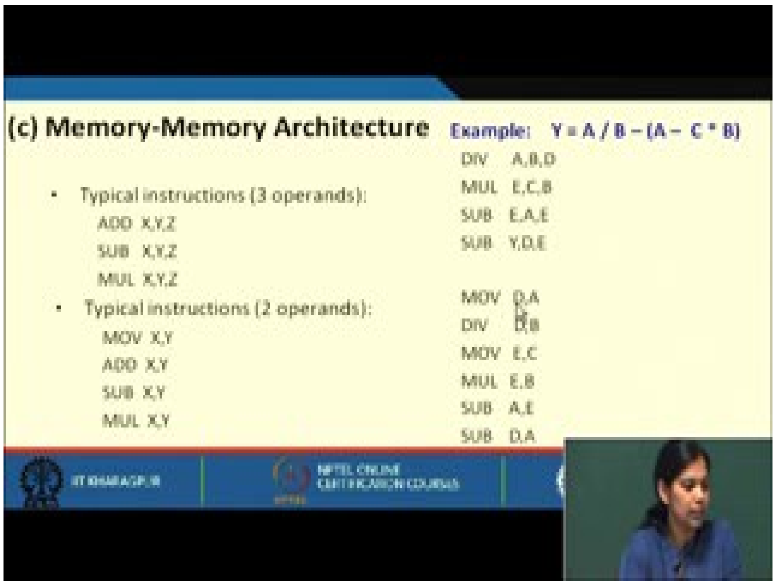
Example: Y = A / B – (A – C * B)

```
LOAD  C
MUL   B
STORE D      // D = C*B
LOAD  A
SUB   D
STORE D      // D = A – C*B
LOAD  A
DIV   B
SUB   D
STORE Y
```

Next we see accumulator architecture. In an accumulator architecture the typical instructions that we will be using is LOAD, STORE and of course, along with that we can use other memory operation along with ADD. So, first we LOAD C then we MUL B, so B is multiplied with C and stored in the accumulator, but we have to store back the result somewhere because later we will be again using it. So, we store the result in D. Then we LOAD A again then we SUB D and we STORE D. So, the operation A - C * B is performed and it is stored in D. Now, finally, what we have to do we have to perform this and we have to subtract this from this.

So, then again we LOAD A and DIV B; this will make A / B and it is stored in accumulator and then we do SUB D. So, whatever is in D will be subtracted from whatever was in accumulated. So, what was in accumulator was A / B, and then it gets subtracted and finally, we store the result in Y. So, whatever was in accumulator is the result of this we store back in Y. And now let us see how many steps we require to execute this. We require 10 steps.

(Refer Slide Time: 27:47)



(c) Memory-Memory Architecture

- Typical instructions (3 operands):
  ADD X,Y,Z
  SUB X,Y,Z
  MUL X,Y,Z
- Typical instructions (2 operands):
  MOV X,Y
  ADD X,Y
  SUB X,Y
  MUL X,Y

Example: Y = A / B – (A – C * B)

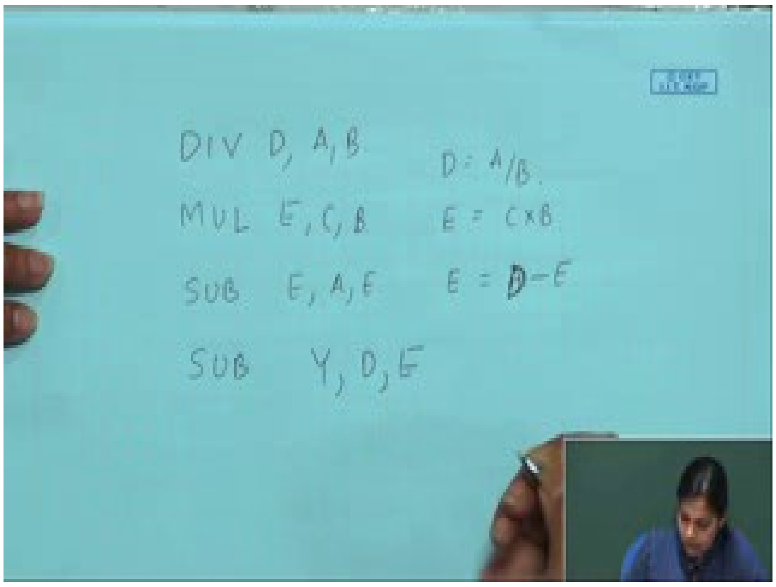
```
DIV  A,B,D
MUL  E,C,B
SUB  E,A,E
SUB  Y,D,E

MOV  D,A
DIV  D,B
MOV  E,C
MUL  E,B
SUB  A,E
SUB  D,A
```

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

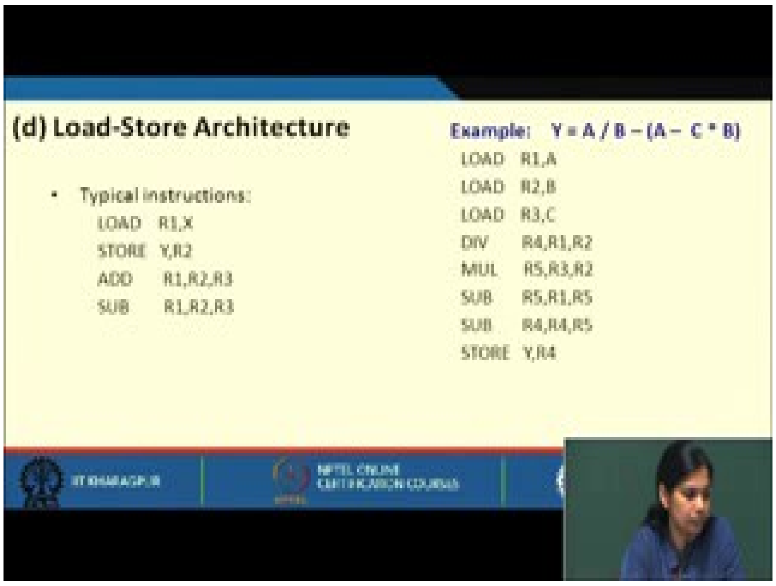
Let us see memory-memory architecture. The programs using both three-address and two-address instructions are shown..

(Refer Slide Time: 28:47)



```
DIV D, A, B     D = A/B
MUL E, C, B     E = CxB
SUB E, A, E     E = D-E
SUB Y, D, E
```

(Refer Slide Time: 32:37)



(d) Load-Store Architecture

- Typical instructions:

```
LOAD   R1,X
STORE  Y,R2
ADD    R1,R2,R3
SUB    R1,R2,R3
```

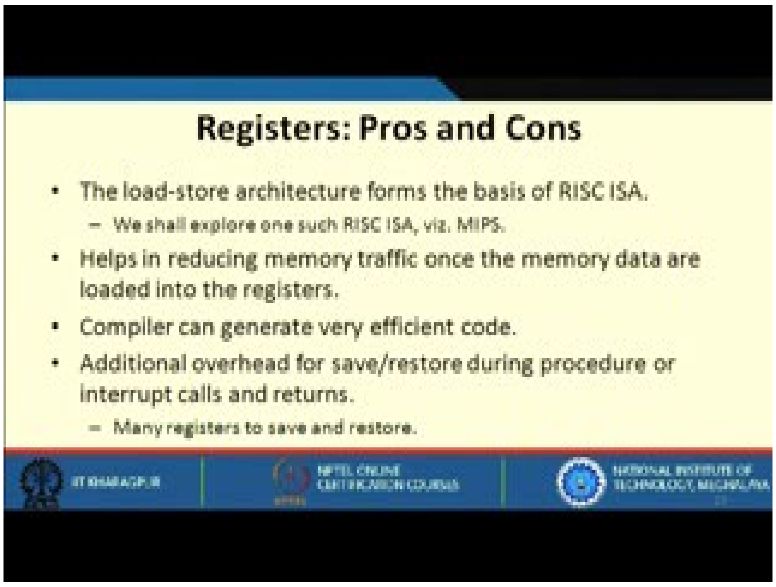
Example: Y = A / B – (A – C * B)

```
LOAD   R1,A
LOAD   R2,B
LOAD   R3,C
DIV    R4,R1,R2
MUL    R5,R3,R2
SUB    R5,R1,R5
SUB    R4,R4,R5
STORE  Y,R4
```

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Next is load-store. In load-store instruction as I said the instruction that will is required to access the memory is LOAD and STORE. So, first of all we have three operands A, B and C --- we will load that into three different registers R1, R2 and R3. Finally, we do DIV where we divide A by B and we store it in R4; then we do MUL B and C, that is R2 and R3 store it in R5. Then we sub R1 minus R5 and store the result back in R5. So, we got this one. We continue this way where the number of instructions required is much less.

(Refer Slide Time: 34:36)



Registers: Pros and Cons

- The load-store architecture forms the basis of RISC ISA.
  - We shall explore one such RISC ISA, viz. MIPS.
- Helps in reducing memory traffic once the memory data are loaded into the registers.
- Compiler can generate very efficient code.
- Additional overhead for save/restore during procedure or interrupt calls and returns.
  - Many registers to save and restore.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, what are the pros and cons that we see. The load store architecture forms the basis of RISC (reduced instruction set computer) instruction set architecture. And in this course we shall explore one such RISC ISA, that is MIPS. What it does it helps in reducing the memory traffic once the memory data are loaded into registers. As you have seen that if only load and store instructions are used to access memory then we can load the data in a go using number of load operations and then we perform all the operations within the processor. So, the processor will be performing all the operations with some register values because we have already loaded those data from your memory into the register. And then finally, if we have to store again back from register to some memory location we can do that.

So, compiler once knows this can of course can generate very efficient code and additional overhead for save and restore during procedure or interrupt calls and return will be clear because now we have many registers to save and restore. But again save and restore will be much more it would not be so much like; what I wanted to say is that it is of course, an additional overhead for saving and restoring during procedure call because we have to save the data and store. But this can be done in an efficient way if we can store it in some other registers as well but although this is an overhead that we see. So, these are the pros and cons of registers.

So, by this we came to the end of lecture 5 and we also came to the end of Week 1 lecture. So, to summarize what we have studied in week 1 --- we started with how computers have evolved, then we have seen that how an instruction can get executed. So, to execute an instruction, we perform set of steps; instructions and data are stored in memory; to execute it we have to bring it from memory to processors, execute it and store it back. We have also seen that what kind of software are existing like application software and system software. We have also seen that both von-Neumann architecture and Harvard architecture are required. And in the last lecture, we have seen that how this instruction set architecture has evolved, what kind of machines were used in early stages, accumulator based machine then stack based machine, then finally, how we are now in a stage where we perform some kind of operation faster. So, our thrust is how we can execute programs faster.

So, in this course, in the next lecture, we will be seeing that how these concepts can be further used to enhance the speed of a computer.

Thank you.