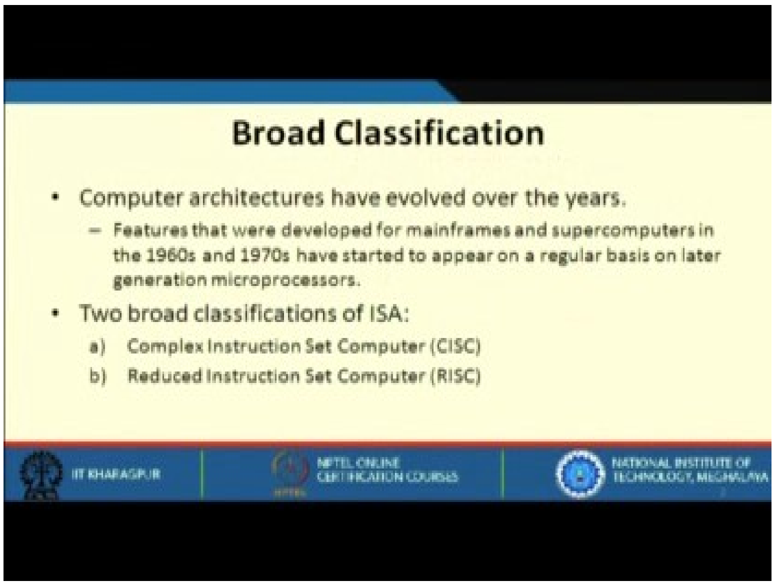
Computer Architecture and Organization

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

Lecture - 08
CISC And RISC Architecture

Welcome to the next lecture on CISC and RISC architecture. Till now what we have seen how an instruction gets executed, what kind of instruction format is available and what kind of addressing modes are used. So, now we will be looking into that architecture where we can divide those sets into some groups, that is, RISC and CISC.

(Refer Slide Time: 01:06)



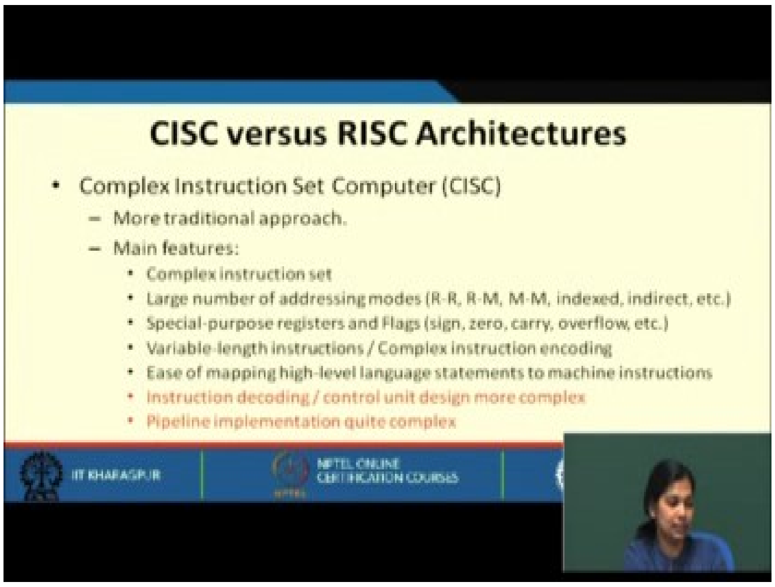
Broad Classification

- Computer architectures have evolved over the years.
  - Features that were developed for mainframes and supercomputers in the 1960s and 1970s have started to appear on a regular basis on later generation microprocessors.
- Two broad classifications of ISA:
  a) Complex Instruction Set Computer (CISC)
  b) Reduced Instruction Set Computer (RISC)

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

This is a broad classification. Computer architectures have evolved over the years. And there are many features that were developed for mainframes and for supercomputers in the 1960s and 70s and that started to appear as regular features on the later generation of microprocessors. Based on that we have so many features; they are broadly classified into complex instruction set computer called CISC, and reduced instruction set computer called RISC.

(Refer Slide Time: 01:49)



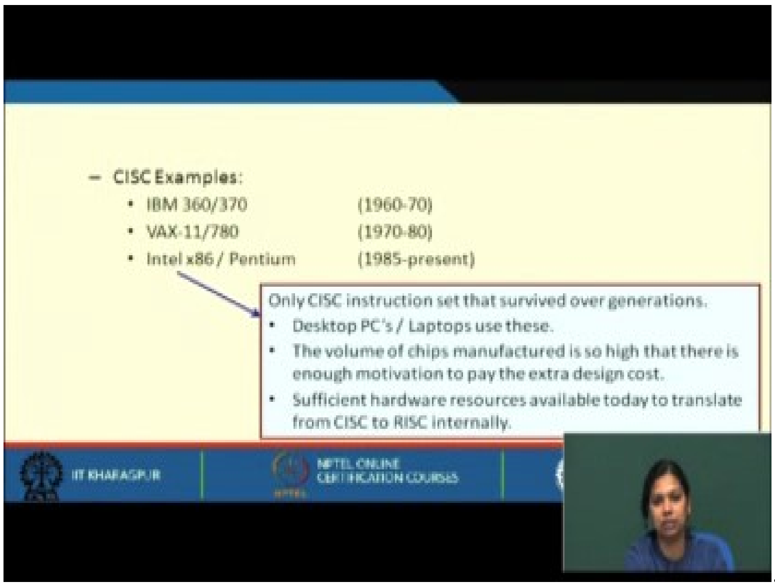
CISC versus RISC Architectures

- Complex Instruction Set Computer (CISC)
  - More traditional approach.
  - Main features:
    - Complex instruction set
    - Large number of addressing modes (R-R, R-M, M-M, indexed, indirect, etc.)
    - Special-purpose registers and Flags (sign, zero, carry, overflow, etc.)
    - Variable-length instructions / Complex instruction encoding
    - Ease of mapping high-level language statements to machine instructions
    - Instruction decoding / control unit design more complex
    - Pipeline implementation quite complex

Now, let us see how CISC and RISC architectures differ. Coming to CISC it is a more traditional approach. So, the main features of this complex instruction set computers are they have complex instruction set. They also have a large number of addressing modes. They also have some special-purpose registers and flags that are used to carry out various operations, and the instructions are not all of same length. So, they have variable length instructions.

So, as I have already told earlier that if you have a fixed-length instruction it becomes easy for encoding and decoding, but if you have variable-length instruction then the encoding needs to be more complex. And it also takes more time for decoding, because you have to know each of the bits and based on that a particular action will be performed. So, ease of mapping high-level language statement to machine code is possible in CISC, but instruction decoding and control unit design are much more complex. And of course, if you have variable length instruction, pipeline implementation will also be quite complex.

(Refer Slide Time: 03:40)


- CISC Examples:
  - IBM 360/370 (1960-70)
  - VAX-11/780 (1970-80)
  - Intel x86 / Pentium (1985-present)

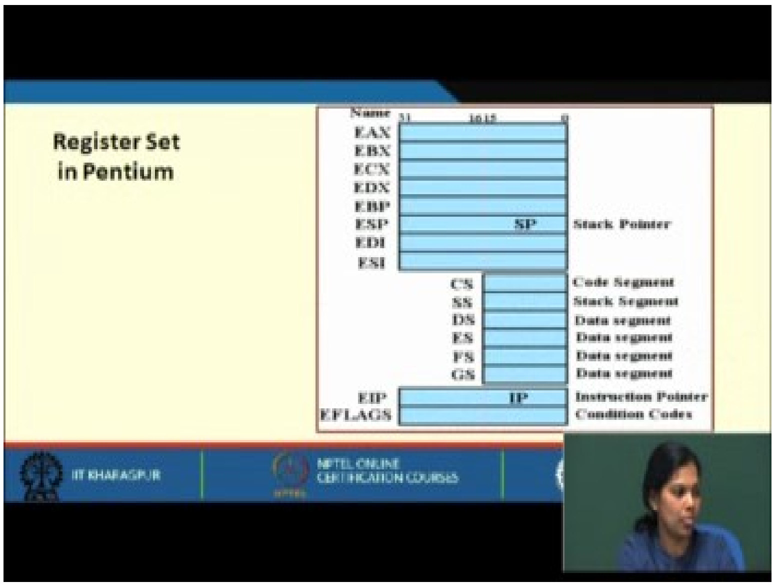
Only CISC instruction set that survived over generations.
- Desktop PC's / Laptops use these.
- The volume of chips manufactured is so high that there is enough motivation to pay the extra design cost.
- Sufficient hardware resources available today to translate from CISC to RISC internally.

.

Now, CISC machines emerged in 60’s and 70’s --- we have seen IBM 360, 370, VAX-11 were popular in seventies and eighties. And from 1985 and till present we have Intel architectures that use CISC architecture; it follows a CISC architecture and has survived over generations. So, over the generation, if you see any CISC machine based on Intel; the desktop PCs, laptops, etc. But in the newer machines you have newer features, but still they are backward compatible.

And this is all possible because the volume of chips that is manufactured is much, much high. So, you can see that there is enough motivation to pay this extra cost of the design although you are paying some extra thing for this CISC architecture. And there are sufficient hardware resources that are available to translate from CISC to RISC internally.

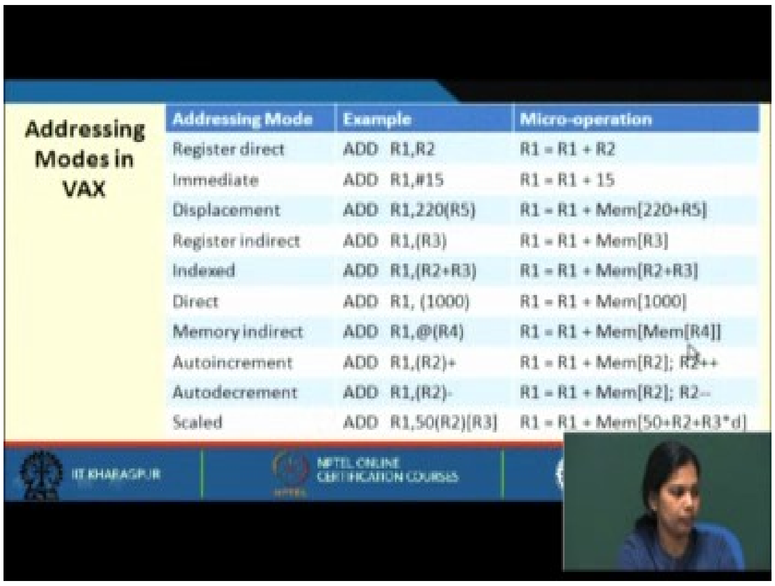
(Refer Slide Time: 05:34)



Register Set in Pentium

So, this is the register set in Pentium. Here you can see that it does not have a large number of registers, although they have got some special-purpose registers like code segment, stack segment, data segment. Also there is program counter which we call instruction pointer, they have some conditional code flags, and there are several other registers to perform different kind of jobs.

(Refer Slide Time: 06:05)

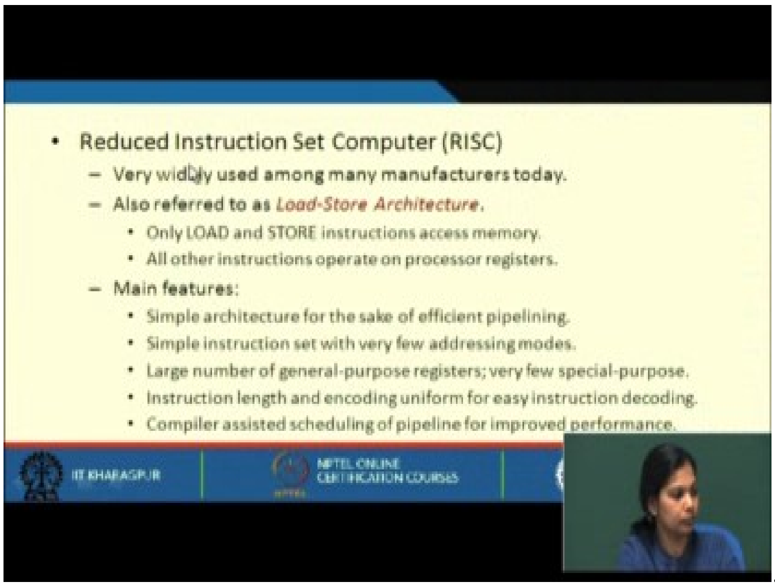


Addressing Modes in VAX

Addressing Mode	Example	Micro-operation
Register direct	ADD R1,R2	R1 = R1 + R2
Immediate	ADD R1,#15	R1 = R1 + 15
Displacement	ADD R1,220(R5)	R1 = R1 + Mem[220+R5]
Register indirect	ADD R1,(R3)	R1 = R1 + Mem[R3]
Indexed	ADD R1,(R2+R3)	R1 = R1 + Mem[R2+R3]
Direct	ADD R1, (1000)	R1 = R1 + Mem[1000]
Memory indirect	ADD R1,@(R4)	R1 = R1 + Mem[Mem[R4]]
Autoincrement	ADD R1,(R2)+	R1 = R1 + Mem[R2]; R2++
Autodecrement	ADD R1,(R2)-	R1 = R1 + Mem[R2]; R2--
Scaled	ADD R1,50(R2)[R3]	R1 = R1 + Mem[50+R2+R3*d]

These are the addressing modes that are present in VAX machine. This was a very popular machine in the 80s. So, just see how many addressing modes are present, starting with register direct, immediate, displacement, register indirect, indexed, direct, memory indirect, auto increment and auto decrement and scaled. So, the VAX machine included a huge set of addressing modes.

(Refer Slide Time: 06:43)



- Reduced Instruction Set Computer (RISC)
  - Very widely used among many manufacturers today.
  - Also referred to as *Load-Store Architecture*.
    - Only LOAD and STORE instructions access memory.
    - All other instructions operate on processor registers.
  - Main features:
    - Simple architecture for the sake of efficient pipelining.
    - Simple instruction set with very few addressing modes.
    - Large number of general-purpose registers; very few special-purpose.
    - Instruction length and encoding uniform for easy instruction decoding.
    - Compiler assisted scheduling of pipeline for improved performance.

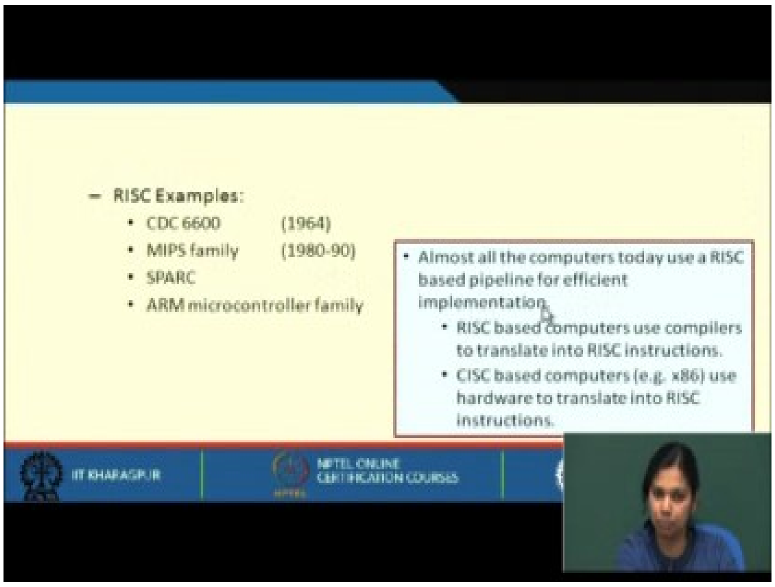
.

Now, coming to reduced instruction set computer which is used in most of the computers today; not only computers, also the processors for microcontrollers follow RISC architecture. This is also referred as load-store architecture. So, by load-store architecture we already discussed that only load and store instruction can be use to access the memory and all other instruction will be operated on register.

So, only when memory data is required, you have to load it from the memory; once the data are available then you can access it. The main features of architecture is very simple for the sake of efficient pipeline. I already discussed very briefly about pipeline that will be discussed in more detail later, but we must know that this is one of the main motivation of RISC; that the RISC architecture is simple for the sake of pipeline.

Simple instruction set is used and very few addressing modes are used. It does not support variety of addressing modes; but it has a large number of general purpose registers. And it does not have many special function registers. Instruction length and encoding is uniform for easy decoding. See if you encode the instruction all the instruction in a specific fashion then the decoding will be much easier. And compiler assisted scheduling of pipeline for improved performance will be there, you will be seeing this particular feature later.

(Refer Slide Time: 09:10)

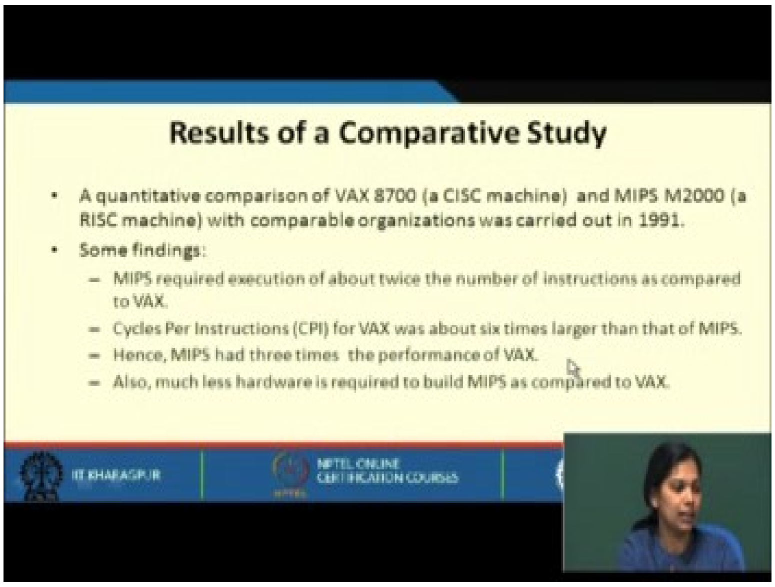


- RISC Examples:
  - CDC 6600 (1964)
  - MIPS family (1980-90)
  - SPARC
  - ARM microcontroller family

- Almost all the computers today use a RISC based pipeline for efficient implementation.
  - RISC based computers use compilers to translate into RISC instructions.
  - CISC based computers (e.g. x86) use hardware to translate into RISC instructions.

So, the popular RISC machine is CDC 6600 that came in 1964. So, RISC is existing since 60s and this is more now that people have moved to RISC architecture. The ARM microcontroller family is all having RISC even some other microcontroller families. So, almost all computers today use RISC based pipeline for efficient implementation. And RISC based computers use compilers to translate into RISC instructions. And CISC based computer use the hardware to translate those instruction into simpler micro-instructions that are RISC instructions.

(Refer Slide Time: 10:12)



Results of a Comparative Study

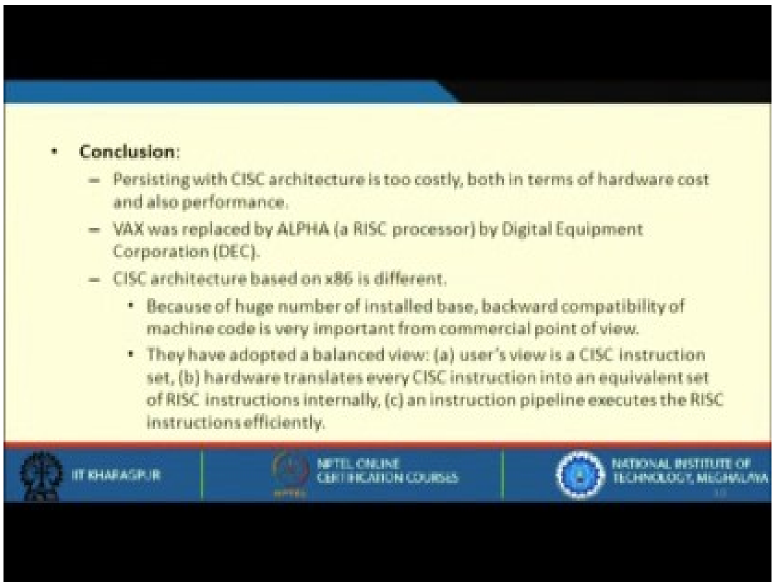
- A quantitative comparison of VAX 8700 (a CISC machine) and MIPS M2000 (a RISC machine) with comparable organizations was carried out in 1991.
- Some findings:
  - MIPS required execution of about twice the number of instructions as compared to VAX.
  - Cycles Per Instructions (CPI) for VAX was about six times larger than that of MIPS.
  - Hence, MIPS had three times the performance of VAX.
  - Also, much less hardware is required to build MIPS as compared to VAX.

A comparative study was performed in the year 1991 where a quantitative comparison between VAX 8700 which was a CISC machine and MIPS M200 a RISC machine was made. And what findings came up are the following. MIPS required execution of about twice the number of instructions as compared to VAX that means the number of instructions that are required by RISC machine is much more compared to CISC. So, as CISC is having more complex instructions, so they require less number of instructions, but on the other hand RISC uses very simple instructions.

There is a parameter that we will be looking little later, cycles per instruction, we call it CPI. So, for each instruction what is the number of cycles that it requires; for VAX machine it was about six times larger than the MIPS machine. So, we can say that in VAX, it is taking only three instruction to execute, but each instruction is taking more number of cycles to execute, whereas MIPS had three times the performance of VAX. So, cycles per instruction were about six time larger in VAX. And, the performance of MIPS was three times the performance of VAX.

Also much less hardware is required to build MIPS as compared to VAX because VAX was having many complex instructions; for which you need to do complex decoding. So, the hardware requirement is much more. Whereas RISC uses simple instructions although they require more number of instructions. So, in that case this MIPS becomes three times faster than VAX.

(Refer Slide Time: 13:11)



- **Conclusion**:
  - Persisting with CISC architecture is too costly, both in terms of hardware cost and also performance.
  - VAX was replaced by ALPHA (a RISC processor) by Digital Equipment Corporation (DEC).
  - CISC architecture based on x86 is different.
    - Because of huge number of installed base, backward compatibility of machine code is very important from commercial point of view.
    - They have adopted a balanced view: (a) user's view is a CISC instruction set, (b) hardware translates every CISC instruction into an equivalent set of RISC instructions internally, (c) an instruction pipeline executes the RISC instructions efficiently.

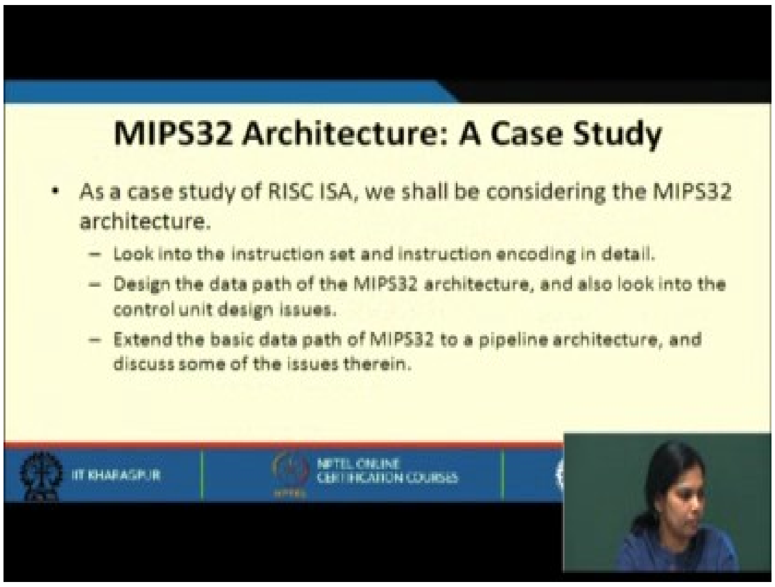
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

So, the conclusion was that persisting with CISC architecture is too costly both in terms of hardware cost and also performance. And with this VAX was replaced by DEC Alpha which was a RISC machine. So, they moved from CISC to RISC after such study. So, this was a machine by DEC. So, CISC architecture based on x86 is different; because of huge number of installation base, backward compatibility of the machine code is very important from commercial point of view.

If backward compatibility is required, you need to have the instructions which you are having earlier. So, because of such thing this x86 which is a CISC machine it is still very important and they have adopted a balanced view. How is that balanced view they have adopted? A user view is a CISC instruction set and there is a hardware that translates every CISC instruction into an equivalent set of RISC instructions internally. So, ultimately internally they are having some kind of RISC instructions.

And also the instruction pipeline executes these RISC instructions efficiently. So, they still have this older CISC architecture at the user point of view level, but at the lower level they have this RISC where those instructions are converted into simpler instructions and those simpler instructions are executed, pipeline can also be performed there which makes it more efficient.

(Refer Slide Time: 15:41)

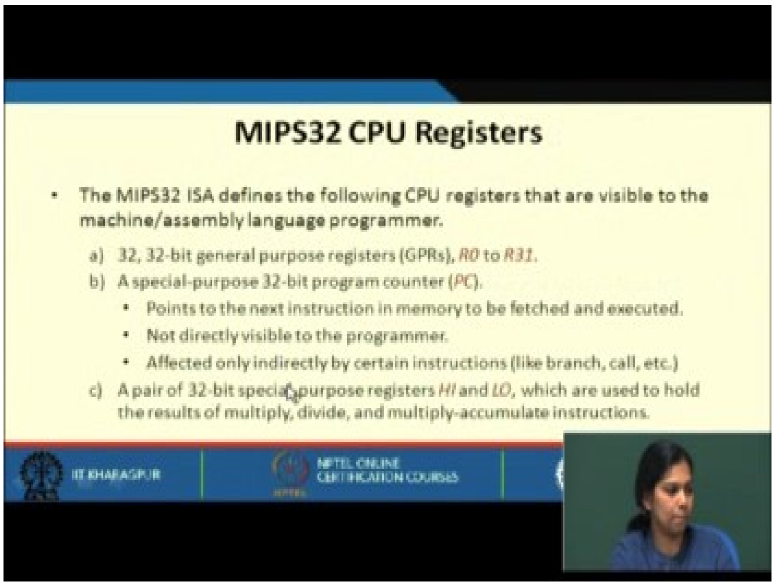


MIPS32 Architecture: A Case Study

- As a case study of RISC ISA, we shall be considering the MIPS32 architecture.
  - Look into the instruction set and instruction encoding in detail.
  - Design the data path of the MIPS32 architecture, and also look into the control unit design issues.
  - Extend the basic data path of MIPS32 to a pipeline architecture, and discuss some of the issues therein.

Now, we will be going into MIPS32 architecture. This is a RISC architecture and we will be performing a case study of this MIPS. We will look into the instruction set instruction encoding in detail also the design of the data path of MIPS architecture and we will also look into the control unit design issues in the later units. And we extend the basic data path of MIPS to a pipeline architecture, and discuss some of the issues therein.

(Refer Slide Time: 16:32)



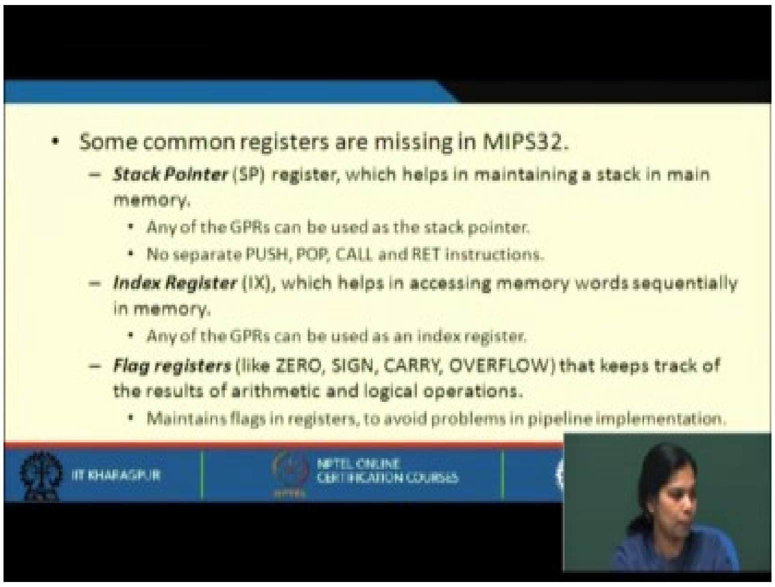
MIPS32 CPU Registers

- The MIPS32 ISA defines the following CPU registers that are visible to the machine/assembly language programmer.
  a) 32, 32-bit general purpose registers (GPRs), *R0* to *R31*.
  b) A special-purpose 32-bit program counter (*PC*).
    - Points to the next instruction in memory to be fetched and executed.
    - Not directly visible to the programmer.
    - Affected only indirectly by certain instructions (like branch, call, etc.)
  c) A pair of 32-bit special-purpose registers *HI* and *LO*, which are used to hold the results of multiply, divide, and multiply-accumulate instructions.

Now, MIPS32 there is a total of 32 general-purpose registers starting from R0 to R31. Also, there is a special 32-bit program counter that is PC, which points to the next instruction in memory to be fetched or executed. Now, this is not directly visible to the programmer and it is only affected by certain instructions. When it is affected, you think of a branch instruction, you think of a call. When you are going to a branch then PC must be loaded with the branch address; you have to calculate the branch address with an offset and then you have to go and execute that.

Now, a pair of 32-bit special purpose registers are also present we call it HI and LO which are used to hold the result of multiply, divide and multiply accumulate instruction. Now, see when you multiply two n-bit numbers the result can be 2n bits. So, you need to store it in a 2n-bit register. So, for that purpose we have two 32-bit registers, HI and LO, which are used for this multiply divide and multiply accumulate instructions.

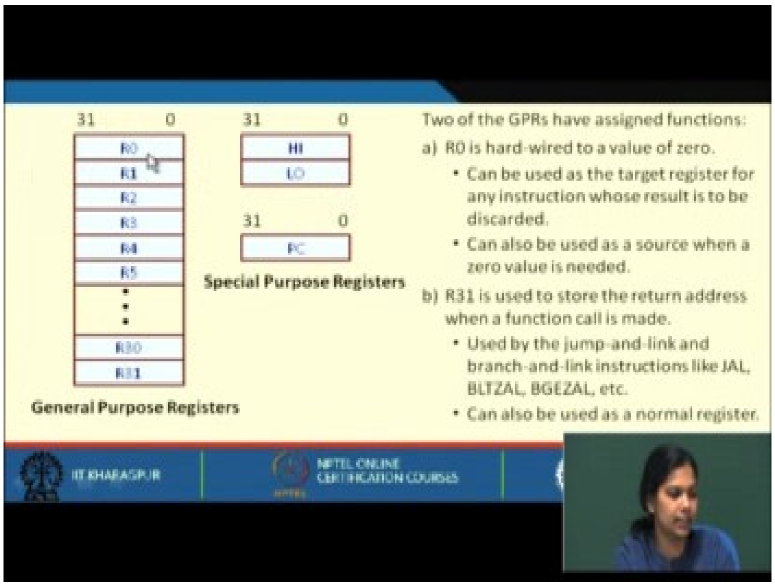
(Refer Slide Time: 18:17)



- Some common registers are missing in MIPS32.
  - ***Stack Pointer*** (SP) register, which helps in maintaining a stack in main memory.
    - Any of the GPRs can be used as the stack pointer.
    - No separate PUSH, POP, CALL and RET instructions.
  - ***Index Register*** (IX), which helps in accessing memory words sequentially in memory.
    - Any of the GPRs can be used as an index register.
  - ***Flag registers*** (like ZERO, SIGN, CARRY, OVERFLOW) that keeps track of the results of arithmetic and logical operations.
    - Maintains flags in registers, to avoid problems in pipeline implementation.

Now, there are some common registers that are not present in MIPS32. What are those registers? We do not have any stack pointer. We already know a stack pointer helps in maintaining the stack in main memory, but we do not have any such kind of stack pointer in MIPS32. We do not have any index register as well that can be used for we have already seen index addressing mode, we require index address index register for some purposes, but it is not present. And what can be done is that as these are not present, you can use any register from the set of 32 registers and use it as an index register. Also there is no flag register; now you can argue that if there is no flag register then how we will be checking the operation? You have to perform the operation, store it in a register check that value for zero or non-zero. So, we do not have any flag register like zero, sign, carry and overflow. Now, why we do not have this you have seen in pipeline there are different instructions that are coming in and they are going in a pipe in an overlapped fashion. Let us say if we have this flag registers it might happen that one instruction has updated some flag then the other instruction is not fully executed but has also updated that flag then there will be a problem. So, instead of doing so this flag can be maintained in some other fashions as done in this MIPS32 machine.

(Refer Slide Time: 20:20)


31
0
R0
R1
R2
R3
R4
R5
R30
R31
General Purpose Registers
HI
LO
PC
Special Purpose Registers
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Two of the GPRs have assigned functions:

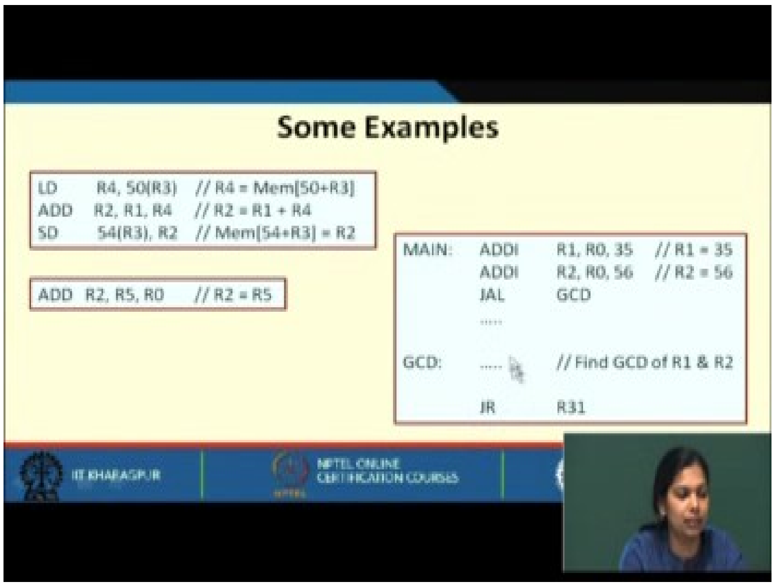
a) R0 is hard-wired to a value of zero.
  - Can be used as the target register for any instruction whose result is to be discarded.
  - Can also be used as a source when a zero value is needed.

b) R31 is used to store the return address when a function call is made.
  - Used by the jump-and-link and branch-and-link instructions like JAL, BLTZAL, BGEZAL, etc.
  - Can also be used as a normal register.

Now, these are the general-purpose registers, and these are the special-purpose registers I have already discussed. R0 register is hardwired to the value 0. So, R0 contains always zero value can be used as target register for any instruction whose result is to be discarded. Or if you add you want to add a zero value to some register you can use R0. R31 is used to store the return address when a function call is made; like as I already discussed that there is no stack pointer.

So, if there is no stack pointer when there will be a call, return, branch these kind of statement it may affect you have to store the address of the PC and then later when again we execute that branch we will come down to that particular address and we execute it. So, in such case we can use this R31 as return address and used by jump and link that is JAL, and other instruction BLTZAL and BGEZAL etc. And it can also be used as a normal register.

(Refer Slide Time: 21:58)



Some Examples

```
LD      R4, 50(R3)    // R4 = Mem[50+R3]
ADD     R2, R1, R4    // R2 = R1 + R4
SD      54(R3), R2    // Mem[54+R3] = R2
```

```
ADD  R2, R5, R0      // R2 = R5
```

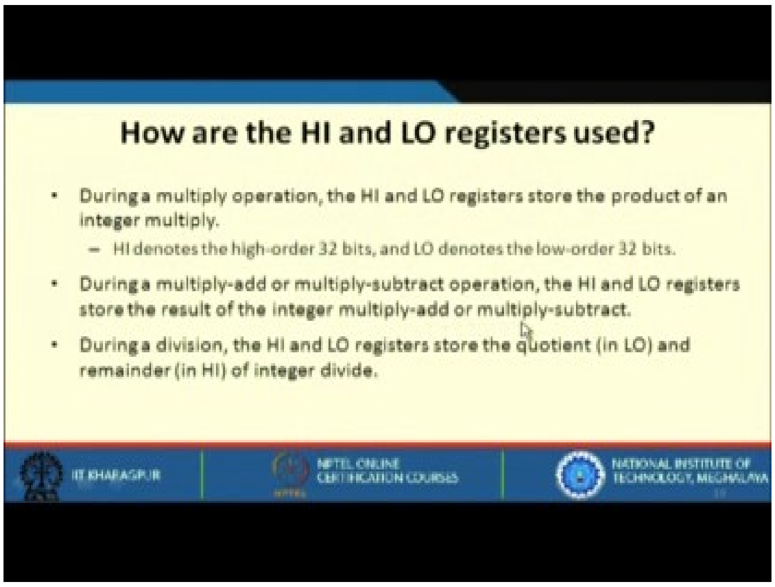
```
MAIN:    ADDI     R1, R0, 35    // R1 = 35
         ADDI     R2, R0, 56    // R2 = 56
         JAL      GCD
         .....

GCD:     .....                  // Find GCD of R1 & R2

         JR       R31
```

Let us take some examples. So, LD R4,50(R3). So, this instruction will load from this particular address. It will add 50 + R3 then it will go to that particular memory location and load the value to R4. Here R1 + R4 is added and it is stored in R2. Similarly, storing means we need to store R2 into this particular location that is 54 + R3. And this is like what we are doing there is no move instruction. If you want to move a data from R5 to R3, all you can do is that you can add R5 with R0 and you can store it in R2. So, what happens R5 is moved to R2.

Now, these are some examples like we have a MAIN code where from where our execution starts and then we have another subroutine. So, what happens here, this is an ADDI, we are adding this immediate value to R0 which is 0. So, R1 will have 35 and R2 will have 56. And this jump instruction will go to a function that is labeled by GCD; and in GCD you will be seeing this some code for your GCD, and jump return (JR).

Now what you need to do now after fetching and decoding you have understood that you have to move there. So, this PC value must be stored in that return address it can be stored in R31, and then we move to this GCD we execute that. At the end of GCD you must have an instruction which is JR to return address where you have to go to R31, get the value loaded in PC, and then from this point the execution will start after returning back here.

(Refer Slide Time: 24:42)



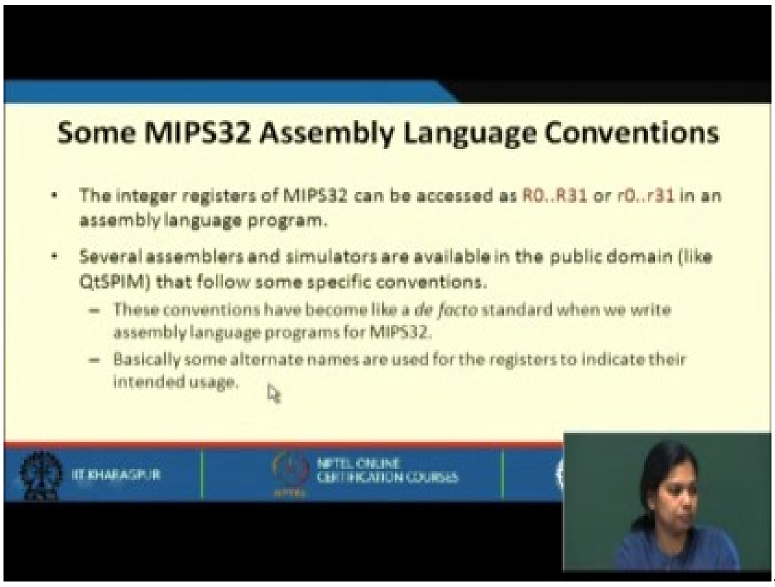
How are the HI and LO registers used?

- During a multiply operation, the HI and LO registers store the product of an integer multiply.
  - HI denotes the high-order 32 bits, and LO denotes the low-order 32 bits.
- During a multiply-add or multiply-subtract operation, the HI and LO registers store the result of the integer multiply-add or multiply-subtract.
- During a division, the HI and LO registers store the quotient (in LO) and remainder (in HI) of integer divide.

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

Now, how are HI and LO registers used? As I said during multiply operation HI and LO registers store the product of an integer multiply. And during multiply-add or multiply-subtract operation the HI-LO registers store the result of an integer multiply-add or multiply-subtract, and during a division HI-LO registers store the quotient in LO and remainder in HI of an integer divide. So, HI-LO registers are used for these few purposes.

(Refer Slide Time: 25:28)



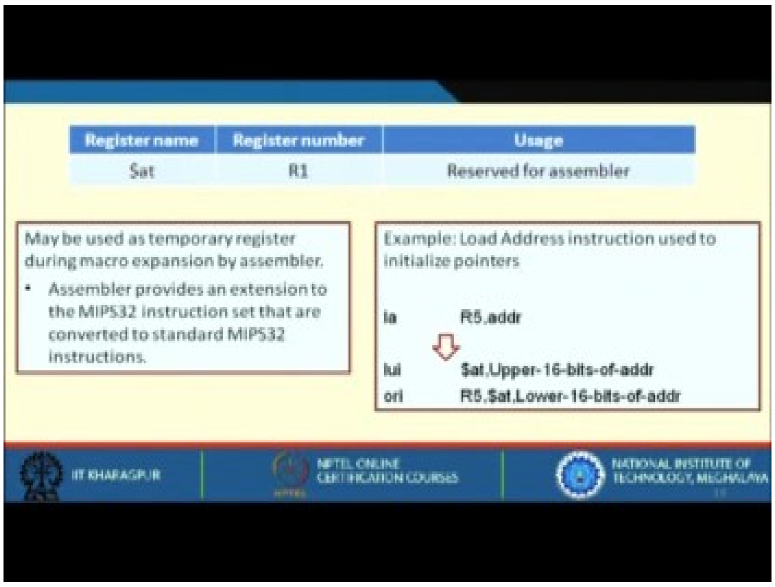
Some MIPS32 Assembly Language Conventions

- The integer registers of MIPS32 can be accessed as R0..R31 or r0..r31 in an assembly language program.
- Several assemblers and simulators are available in the public domain (like QtSPIM) that follow some specific conventions.
  - These conventions have become like a *de facto* standard when we write assembly language programs for MIPS32.
  - Basically some alternate names are used for the registers to indicate their intended usage.

.

Now, let us see little more MIPS32 assembly language conventions. So, the integer registers are numbered from R0 to R31. So, for doing so some simulators are available in public domain like QtSPIM. You can write assembly language code there and execute it. And these conventions have become a de facto standard when we write an assembly language program for MIPS. So, a QtSPIM is exactly a MIPS32 simulator that can be used to write assembly language programs. Basically some alternative names are used for register to indicate their intended use in this.

(Refer Slide Time: 26:31)



Register name	Register number	Usage
$at	R1	Reserved for assembler

May be used as temporary register during macro expansion by assembler.

- Assembler provides an extension to the MIPS32 instruction set that are converted to standard MIPS32 instructions.

Example: Load Address instruction used to initialize pointers

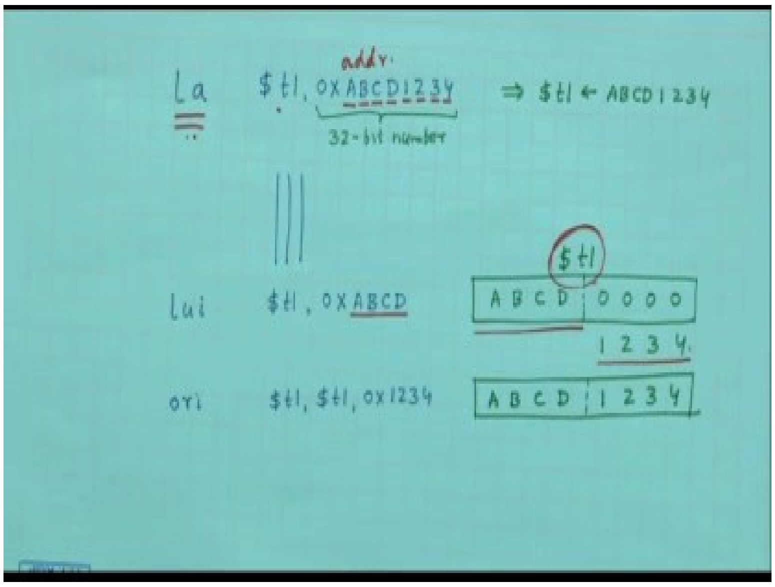
```
la      R5,addr
   ⇩
lui     $at,Upper-16-bits-of-addr
ori     R5,$at,Lower-16-bits-of-addr
```

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

So, register zero - R0 which is a constant zero. So, register name $zero is used to represent constant zero value whenever required in a program. As this particular register is reserved for an assembler, the name is $at. And this may be used as temporary register during macro operation by an assembler. An assembler provides an extension to the MIPS instruction set that are converted to standard MIPS32 instructions.

Let us see an example load address instruction used to initialize the pointer that means, in R5 what we are doing we are loading an address, addr is a label from where we have stored some a data. And I am loading that particular address into R5, but this instruction is not there in your MIPS. So, how MIPS will see this, MIPS will convert this into set of two instructions LUI (Load Upper Immediate). So, at this upper 16-bit of address is loaded and then we do an or immediate (ori) with the lower 16 bit address. Firstly, I have loaded the 16-bit number in the upper immediate.

(Refer Slide Time: 28:33)

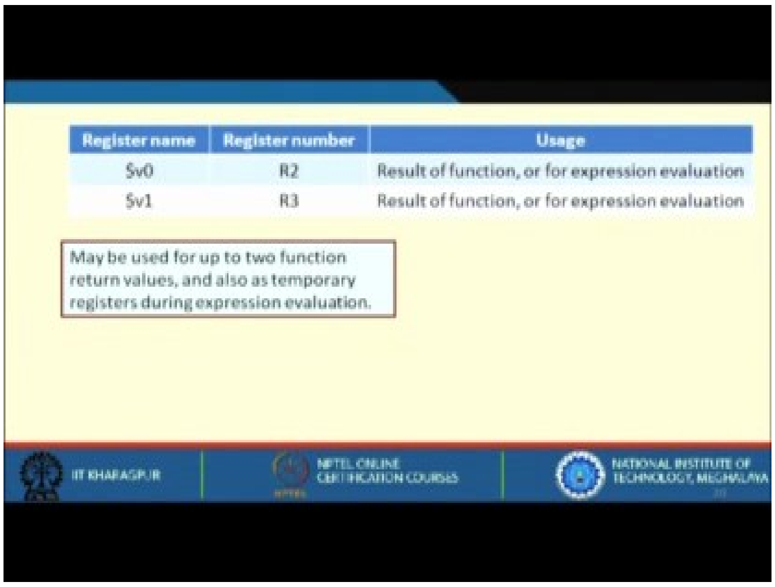


```
la    $t1, 0xABCD1234    ⇒ $t1 ← ABCD1234
lui   $t1, 0xABCD
ori   $t1, $t1, 0x1234
```

Let us take this example that will make it clear. So, this la is loading this particular address into t1 and this is nothing but addr, either you write addr or actually this address. This is a 32-bit number. We do not have such kind of instruction la in MIPS. So, this instruction will break down into couple of instructions when you actually execute. So, this is the upper four nibbles A, B, C and D. So, in $t1 the upper 4 nibbles is added.

So, now you see in t1 this 16-bit is loaded with A, B, C, D. Now, what will we do we do an ori that will add this and this with t1; in t1 you already have this particular value. You are doing an ori with 0x1234 that is the next four nibbles you are or-ing with 1234, and storing back the result in t1 itself. So, in t1 you will have ABCD you have or with 1234, 1234. So, this instruction is doing exactly what I explained.

(Refer Slide Time: 30:31)



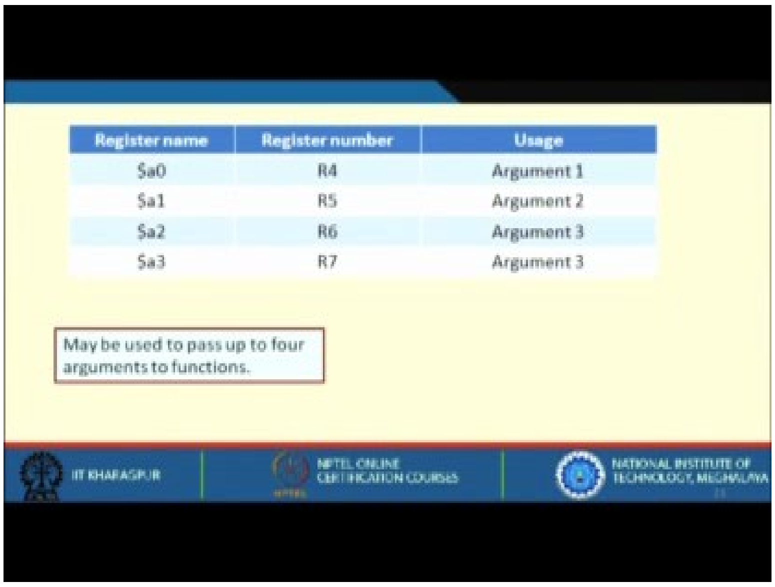
Register name	Register number	Usage
$v0	R2	Result of function, or for expression evaluation
$v1	R3	Result of function, or for expression evaluation

May be used for up to two function return values, and also as temporary registers during expression evaluation.

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

Similarly, we have another register that is $v0 and $v1; the register that are used is R2 and R3. So, result of a function or an expression evaluation is stored in $v0 and in $v1. Let us see, may be it can be used for up to two functions, return values and also as temporary register during expression evaluation.

(Refer Slide Time: 31:21)



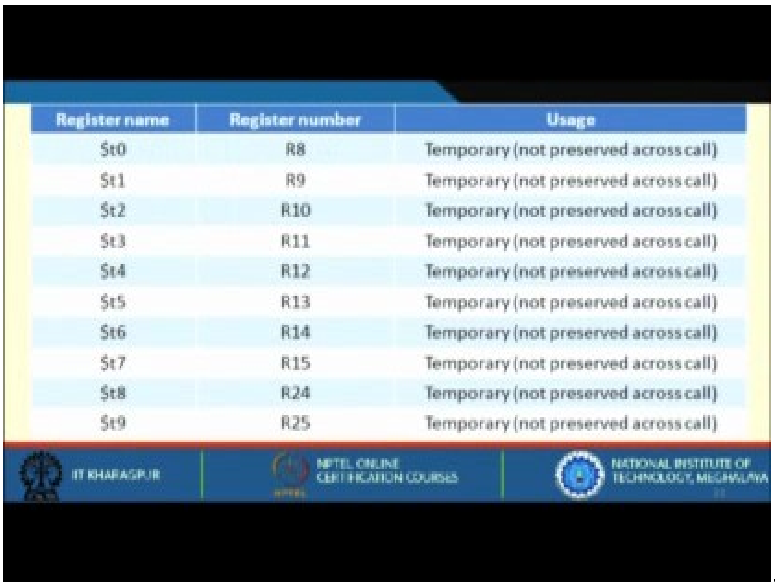
Register name	Register number	Usage
$a0	R4	Argument 1
$a1	R5	Argument 2
$a2	R6	Argument 3
$a3	R7	Argument 3

May be used to pass up to four arguments to functions.

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

Next set of register is $a0, $a1, $a2, $a3 we use as arguments. So, these are used to pass up to four arguments to a function.

(Refer Slide Time: 31:43)



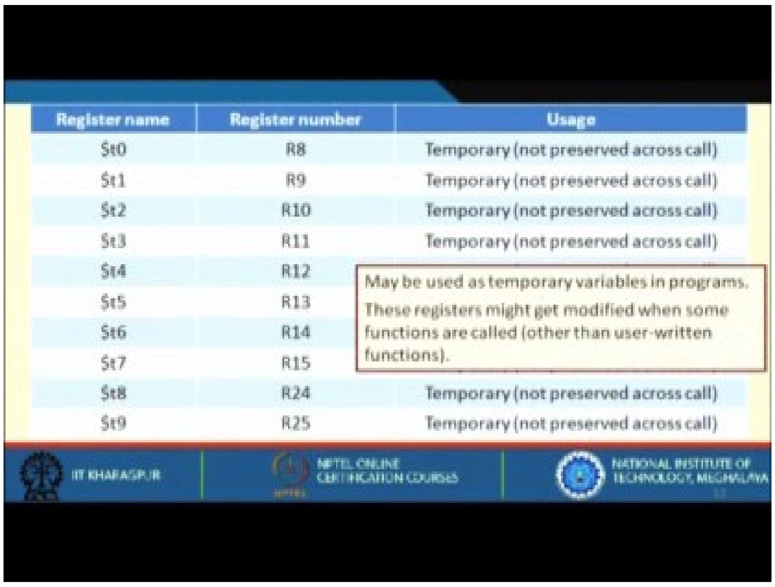
Register name	Register number	Usage
$t0	R8	Temporary (not preserved across call)
$t1	R9	Temporary (not preserved across call)
$t2	R10	Temporary (not preserved across call)
$t3	R11	Temporary (not preserved across call)
$t4	R12	Temporary (not preserved across call)
$t5	R13	Temporary (not preserved across call)
$t6	R14	Temporary (not preserved across call)
$t7	R15	Temporary (not preserved across call)
$t8	R24	Temporary (not preserved across call)
$t9	R25	Temporary (not preserved across call)

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

.

These are temporary register $t0 to $t9. And these are not preserved across calls. So, it might happen that when you are using this register for some purpose, it can be also used for other. So, it is not preserved across the call.

(Refer Slide Time: 32:09)



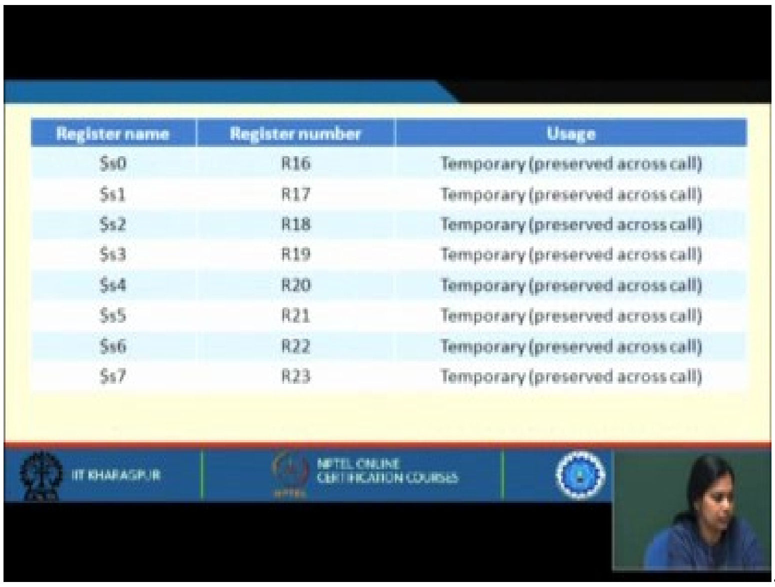
Register name	Register number	Usage
$t0	R8	Temporary (not preserved across call)
$t1	R9	Temporary (not preserved across call)
$t2	R10	Temporary (not preserved across call)
$t3	R11	Temporary (not preserved across call)
$t4	R12	
$t5	R13	
$t6	R14	
$t7	R15	
$t8	R24	Temporary (not preserved across call)
$t9	R25	Temporary (not preserved across call)

May be used as temporary variables in programs.
These registers might get modified when some functions are called (other than user-written functions).

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

May be used as temporary variables in programs, but these registers might get modified. So, it is up to you that if you are using this then you have to be careful enough, and these register might get modified when some functions are called other than user written functions that means, inside also we are calling some function, some instruction are breaking down into some further instructions.

(Refer Slide Time: 32:50)



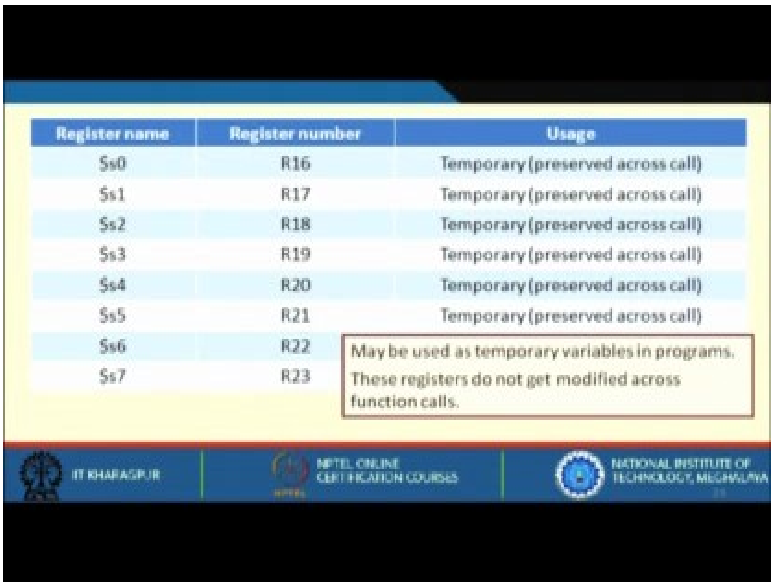
Register name	Register number	Usage
$s0	R16	Temporary (preserved across call)
$s1	R17	Temporary (preserved across call)
$s2	R18	Temporary (preserved across call)
$s3	R19	Temporary (preserved across call)
$s4	R20	Temporary (preserved across call)
$s5	R21	Temporary (preserved across call)
$s6	R22	Temporary (preserved across call)
$s7	R23	Temporary (preserved across call)

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

.

So, in this particular case you can see that it can be user other than user return function it can be overwritten, but these are preserved across call. So, even if there are some other function calls you can use this set of temporary register starting from $s0 to $s7.

(Refer Slide Time: 33:04)



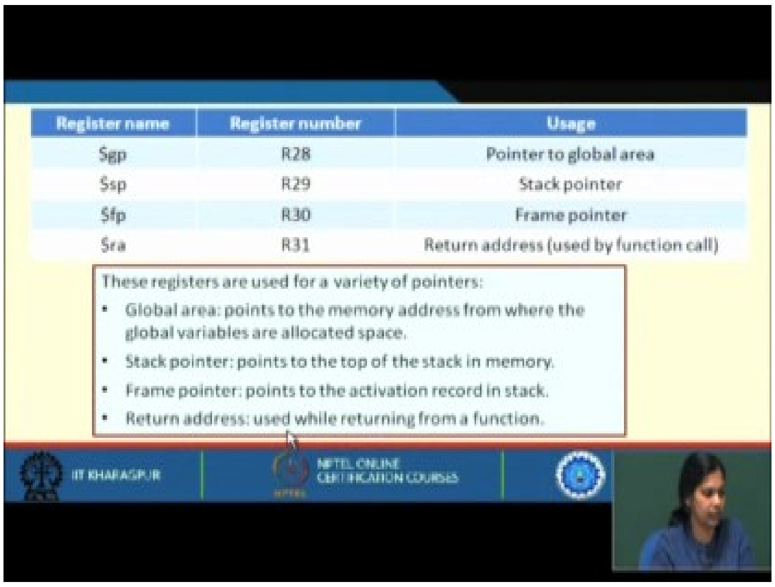
Register name	Register number	Usage
$s0	R16	Temporary (preserved across call)
$s1	R17	Temporary (preserved across call)
$s2	R18	Temporary (preserved across call)
$s3	R19	Temporary (preserved across call)
$s4	R20	Temporary (preserved across call)
$s5	R21	Temporary (preserved across call)
$s6	R22	
$s7	R23	

May be used as temporary variables in programs.
These registers do not get modified across function calls.

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

May be used as temporary variables in the program. So, I will suggest you if you are coding this using you are writing some assembly language program then you can use this set of these set of registers for your programming purpose.

(Refer Slide Time: 33:26)



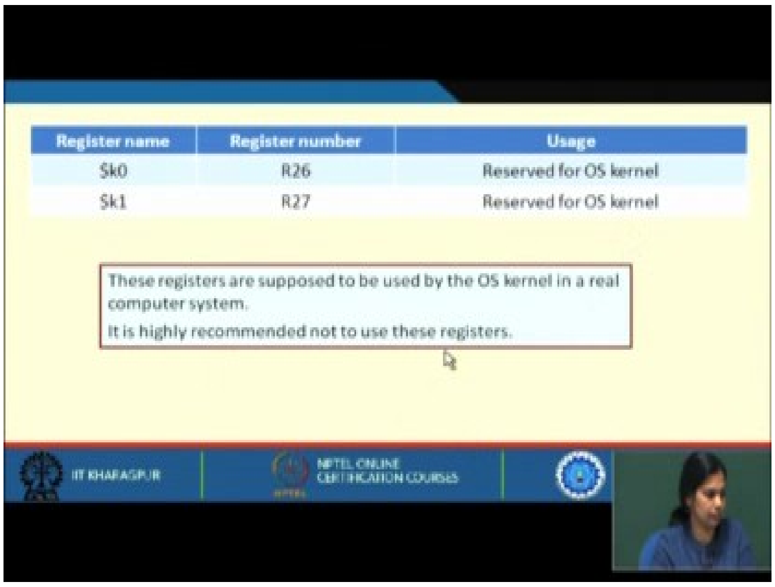
Register name	Register number	Usage
$gp	R28	Pointer to global area
$sp	R29	Stack pointer
$fp	R30	Frame pointer
$ra	R31	Return address (used by function call)

These registers are used for a variety of pointers:

- Global area: points to the memory address from where the global variables are allocated space.
- Stack pointer: points to the top of the stack in memory.
- Frame pointer: points to the activation record in stack.
- Return address: used while returning from a function.

We have another set which is $gp, $sp, $fp, and $ra. So, $gp is pointed to global area, $sp is your stack pointer, $fp is your frame pointer, and $ra is your return address. So, we are having all these, but we are not having with a special function register rather the general purpose registers are only used for this, but we know that R29 is a general purpose register which is used for stack pointer. So, these registers are used for variety of pointers. So, global pointers as I said point to the memory location from where the global variables are allocated space. Stack pointer points to the top of the stack. A frame pointer points to the activation record in stack. And return address is used while returning from a function.

(Refer Slide Time: 34:34)



Register name	Register number	Usage
$k0	R26	Reserved for OS kernel
$k1	R27	Reserved for OS kernel

These registers are supposed to be used by the OS kernel in a real computer system.
It is highly recommended not to use these registers.

We have register $k0 and $k1 reserved for OS kernel. So, it is highly recommended not to use these registers.

So, we come to the end of lecture 8. So, in this lecture, now we have given you an idea what we will be discussing next. We have discussed about some properties of instruction set architecture and now we are moving on with a particular architecture that is MIPS32, and I have discussed some of its features and we will be discussing many more in course of time.

Thank you.