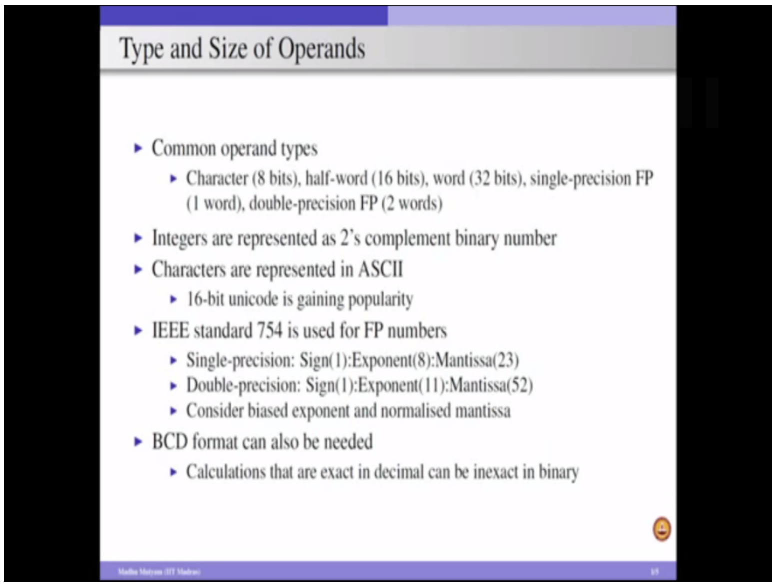
# Computer Architecture

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

## Lecture - 05
## Instruction Set Principles (Part 3)

So, having discussed memory addressing and addressing modes in the previous modules, we now discuss the type of operands and operations considered in instructions and also instruction encoding in this module. So, instruction set supports different types of operands and their sizes also differ depending on the type of operand we consider.

(Refer Slide Time: 00:41)



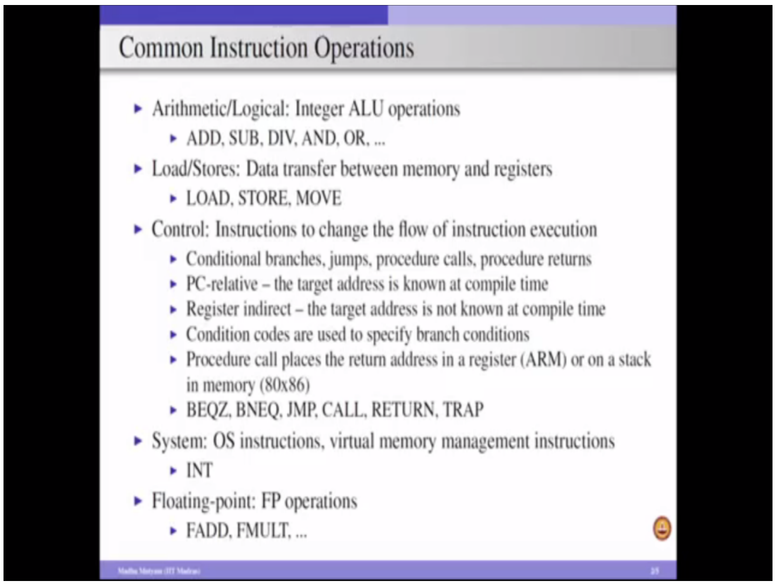
### Type and Size of Operands

- Common operand types
  - Character (8 bits), half-word (16 bits), word (32 bits), single-precision FP (1 word), double-precision FP (2 words)
- Integers are represented as 2's complement binary number
- Characters are represented in ASCII
  - 16-bit unicode is gaining popularity
- IEEE standard 754 is used for FP numbers
  - Single-precision: Sign(1):Exponent(8):Mantissa(23)
  - Double-precision: Sign(1):Exponent(11):Mantissa(52)
  - Consider biased exponent and normalised mantissa
- BCD format can also be needed
  - Calculations that are exact in decimal can be inexact in binary

The common operand types are character - which takes 8 bits of storage, half word - is like a two bytes, a word - four byte, single precision floating point numbers - which take one word and double precision floating point takes two words. And integers are typically represented in the 2's complement form. And for characters we either use seven bit ASCII codes or 16 bit unicode and for floating point numbers we typically consider a IEEE standards 754. For a single precision, a floating point number is represented in 32 bit where one bit is given for sign, 8 bits are given for exponent and the remaining bits are given for mantissa.

Whereas, in the case of double precision, if you want to increase the precision, so we use 64 bit where one bit is for sign, eleven bit exponent and the mantissa is 52. And for representing exponent and the mantissa we consider biased exponent form and for the mantissa we consider normalized mantissa. And sometimes it may not be possible to perform exact calculations using binary numbers. So, in those cases, especially financial calculations and so on which may require the calculations may need to be performed on decimal numbers. So, to support those things we can consider the BCD binary coded decimal form

(Refer Slide Time: 02:29)



## Common Instruction Operations

- Arithmetic/Logical: Integer ALU operations
  - ADD, SUB, DIV, AND, OR, ...
- Load/Stores: Data transfer between memory and registers
  - LOAD, STORE, MOVE
- Control: Instructions to change the flow of instruction execution
  - Conditional branches, jumps, procedure calls, procedure returns
  - PC-relative – the target address is known at compile time
  - Register indirect – the target address is not known at compile time
  - Condition codes are used to specify branch conditions
  - Procedure call places the return address in a register (ARM) or on a stack in memory (80x86)
  - BEQZ, BNEQ, JMP, CALL, RETURN, TRAP
- System: OS instructions, virtual memory management instructions
  - INT
- Floating-point: FP operations
  - FADD, FMULT, ...

Now, what type of operations, the instruction set typically supports or what type of instructions we need to consider in our new ISA that we want to design? So, generally any ISA should support arithmetic and logic operations its nothing but integer ALU operations, those are add, subtract, multiply, division and our shift operations and etcetera. So, after these ALU operations because we want to read the data from memory or write the data to the memory.

So, we need data transfer operations to perform transferring of data between the memory and the registers, where we typically require instructions such as load store and move. Move is typically memory to memory transfer operations and we also require control transfer instructions. These are the instructions which change the flow of execution of instructions in the program. So, these are the conditional branches, jump instructions, procedure calls, procedure returns and so on. So, when we are dealing with these control instructions because in the normal flow of execution after executing on one instruction we execute the next instruction in the sequence, but because of this control transfer instructions the control goes through a different address which we need to be mentioned efficiently.

So, for that we typically use pc relative addressing mode, when we know the target address at the compile time. So, when we are compiling the code if we know the target address, we typically apply pc relative addressing. As we discussed in the addressing mode module that pc relative addressing provides the displacement with respect to the next instruction address which is stored in the program counter register, but what happens if the target address is not known at compiled time.

So, what is the addressing mode we can use? In those cases, we typically use register indirect. We dedicate a register and this register will be updated with the target address at run time. So, as a result we specify the address of the register in our instruction. So, it is called as a register indirect. And once we go to the particular register we actually get the address of the target instruction. So, there is an indirection through a register file. So, that is why it is called as register indirect addressing mode.

And we also use conditional codes especially in the ARM and 80x86 ISAs and these conditional codes are used to specify the branch conditions. So, typically when a particular operation is performed. So, certain condition flags or condition bits will be set in a register and these bits will be checked to see whether a particular branch to be taken place or not. Because a conditional branch is like a branch happens based on some condition is true or false. So, for those things we typically check the bits in this condition code.

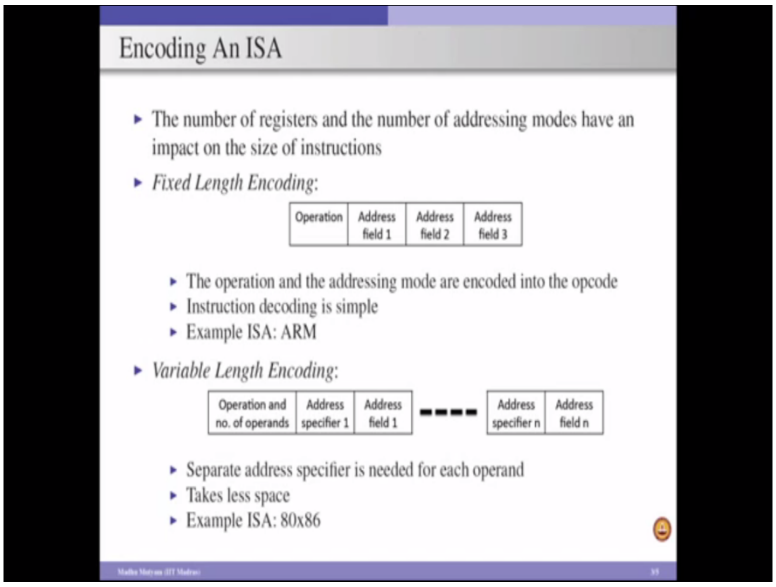
And because control transfer instructions also include procedure calls and procedure returns. When we are making a procedure call we are actually going from the current instruction address to a different instruction address, which may not be in the sequence. And after executing that procedure we have to come back to the next instruction in the sequence from where we actually called this procedure. So, there are two ways in which we store the return address. In the case of ARM we store this return address in a register.

And in the case of 80x86, we use stack to store the return address and typical examples for this control transfer instructions are branch equal to zero, branch not equal to zero, jump, call, return, trap and so on. So, these are the three types of instructions almost all ISAs need to have. In addition to that, we need to have support for system instructions which are like operating system instructions and also virtual memory management instructions. So, these instructions are required to deal with protection, privilege mode executions, handling interrupts and so on.

And finally, if you want to design our computer to perform floating point operations, scientific calculations and so on. We may need to have instructions supporting these floating point operations. So, these floating point operations are like floating point addition, floating point multiplication and so on. In addition to these the basic instruction types we can also consider other type of instructions such as graphic instructions, performing operations and pixel level operations or vector operations and so on.

So, we can consider again depending on the requirements are for ISA we can select the required the instructions in our set. So having discussed the type of operands, the type of operations, now we look at encoding an instruction. An instruction set architecture. We know that a typical instruction consists of operands, opcode. So, the size of this operand field, opcode field actually affects the overall size of the instruction.

(Refer Slide Time: 09:06)



## Encoding An ISA

- The number of registers and the number of addressing modes have an impact on the size of instructions
- *Fixed Length Encoding*:

| Operation | Address field 1 | Address field 2 | Address field 3 |
|---|---|---|---|

  - The operation and the addressing mode are encoded into the opcode
  - Instruction decoding is simple
  - Example ISA: ARM
- *Variable Length Encoding*:

| Operation and no. of operands | Address specifier 1 | Address field 1 | ---- | Address specifier n | Address field n |
|---|---|---|---|---|---|

  - Separate address specifier is needed for each operand
  - Takes less space
  - Example ISA: 80x86

Madhu Mutyam (IIT Madras)

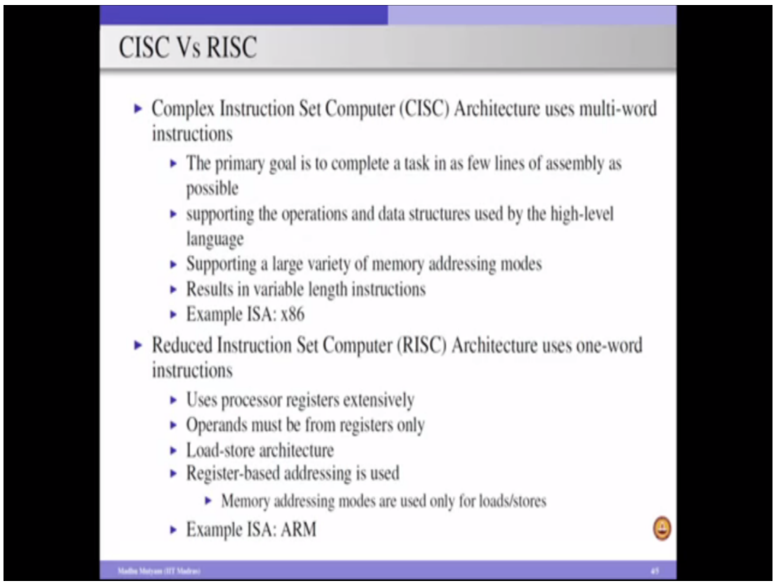
In addition to that the number of register that we use in our ISA also affects our, the overall, the size of the instruction. For example, if you have 64 registers as compared as compared to 32 registers, we have to spend 6 bits of the address to refer to a register in the register file in the case of 64 entry register file whereas, in the case of 32 entry register file we require only 5 bits to specify a register. And at the same time if we increase the number of addressing modes, that also has an impact on our instruction length because we need to specify what is the addressing mode we are using for that particular operand.

So, given all these things, these instructions are encoded traditionally in two ways. One is a fixed length encoding, where total instruction is occupied in one word and each instruction is divided into multiple fields and each field has a specific meaning to it. The first field of the instruction specifies the operation and the addressing mode that we use and the remaining fields specifies the address field or operand field. So, the main advantage with this fixed length encoding is the decoding is simple.

And typically ARM uses this type of encoding. So, the other class of instruction encoding considers variable length, where we can consider any number of operands and each operand can take any addressing mode. We have one field for the opcode which specifies the instruction; the next field specifies the total number of operands associated with that instruction. And then for each operand as it can assume any addressing mode, we need to have two fields one specifies the addressing mode the second one specifies the actual address. So, in this figure we can clearly see.

Address Specifier 1 address field one similarly, address specifier n and address field n. There are n operands associated with this particular instruction and this n should be specified in the number of operands field. So, the main advantage with this encoding is, it takes less space because you can keep as much information as possible to realize an operation specified in your high level programming language. And the ISA which considers this type of encoding is 80x86. So, having discussed this instruction encoding, we look at the two classes of ISAs which are the CISC and RISC.

(Refer Slide Time: 12:27)



## CISC Vs RISC

- Complex Instruction Set Computer (CISC) Architecture uses multi-word instructions
  - The primary goal is to complete a task in as few lines of assembly as possible
  - supporting the operations and data structures used by the high-level language
  - Supporting a large variety of memory addressing modes
  - Results in variable length instructions
  - Example ISA: x86
- Reduced Instruction Set Computer (RISC) Architecture uses one-word instructions
  - Uses processor registers extensively
  - Operands must be from registers only
  - Load-store architecture
  - Register-based addressing is used
    - Memory addressing modes are used only for loads/stores
  - Example ISA: ARM

Madhu Mutyam (IIT Madras) 45

Among these two the CISC is the older ISA. CISC stands for the complex instruction set Computer architecture and it uses multi-word instructions. So, if we just recall our instruction encoding discussion, we know that variable length instruction typically fall into the CISC type of ISA. So, what was the main reason the CISC architectures were proposed because this is the very old the ISA considered. And in those times, the main memory was very premium. So, the main objective was to reduce our code size.

To reduce the code size so, we need to have we need to specify the task that is given by your high level programming language using as few as assembly instructions as possible. So, we need to encode too much information per instruction. So to do that, we need to support our operations and the data structures used by the high level language. So, that is the reason why this CISC architecture uses many addressing modes and also uses the variable length operands because of this the goal of reducing the number of instructions assembly instructions to perform a task and so on and supporting large number of addressing modes and also supporting more number of operands in the instruction.

So, it results in variable length instructions. And the typical example for this class of architecture is x86, but as the technology advances and so on the memory cost was reduced significantly. As a result so we do not have to worry about the code size so that we can always go for the simple instruction. Remember when we are using an instruction belongs to

CISC type of architecture because it has variable length instructions, our decoding of instruction will become very complex.

And our CPI per instruction increases significantly. So, as a result it is difficult to perform pipeline operations on CISC type of instructions. To overcome all these problems and also to exploit the cost reduction of the memory, a new class of architecture is proposed, which is called as a RISC. RISC stands for Reduced Instruction Set Computer architecture and which uses one word instructions rather than multi word instructions.

And because the processor design has evolved significantly supports multiple registers inside it. So, we can exploit these processor registers for efficient computations. So, we can make use of this processor registers extensively inorder to deal with these RISC instructions. And RISC puts a condition that all ALU operations should be performed on register operands only. You can access memory only for getting the data for accessing the memory locations and that too by using special purpose instructions which are called as load and store.

And that is the reason why this RISC architecture also called as load store architectures and the example ISA is ARM. And note that because we are actually using the registers extensively in RISC, we typically use addressing modes which are based on registers. So, effectively register based addressing modes are used in RISC architectures, but there is no constraint on this in the CISC architectures. So, effectively the RISC architectures are simple and we can efficiently implement the pipelining concept and decoding is simple.

And also because we use a fixed length instruction, instructions, in the RISC architectures and processor registers can be used efficiently or extensively. And also accessing register takes much less time compared to accessing the memory and so on. So, the current day the processors are actually using explicitly the RISC architectures or implicitly the RISC architectures. When I say RISC architecture is used implicitly for example, if you consider Intel processors, though their instruction set belongs to CISC because of the backward compatibility.

Internally they covert this each of these complex instructions into set of simple instructions which are almost like a RISC type of instructions and perform their operations on processor registers. So, with this I am going to conclude the instruction set principles unit. And in the next module we are going to discuss the memory hierarchy starting with the cache memory.

Thank you.