Instruction Set Architecture

ISA is an abstract interface between the hardware and the lowest-level software that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O and so on. ISAs differ based on the internal storage in a processor. Accordingly, the ISA can be classified as follows, based on where the operands are stored and whether they are named explicitly or implicitly:

- **Single accumulator organization**, which names one of the general-purpose registers as the accumulator and uses it to necessarily store one of the operands. This indicates that one of the operands is implied to be in the accumulator and it is enough if the other operand is specified along with the instruction.
- **General register organization,** which specifies all the operands explicitly. Depending on whether the operands are available in memory or registers, it can be further classified as

  – **Register – register**, where registers are used for storing operands. Such architectures are in fact also called **load – store** architectures, as only load and store instructions can have memory operands.

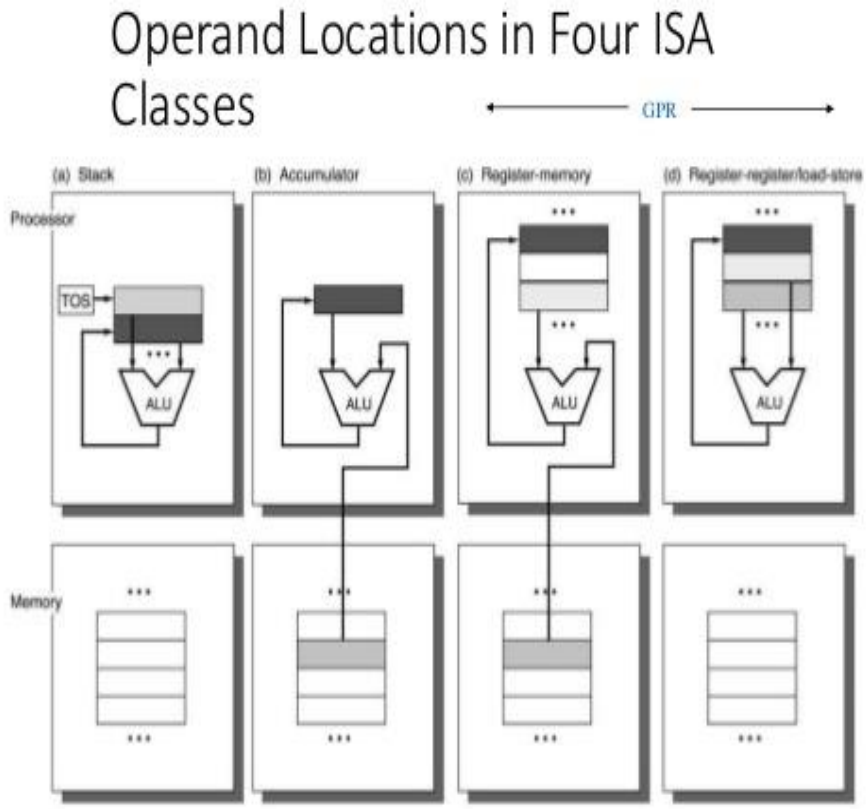
  – **Register – memory**, where one operand is in a register and the other one in memory.

  – **Memory – memory**, where all the operands are specified as memory operands.

- **Stack organization**, where the operands are put into the stack and the operations are carried out on the top of the stack. The operands are implicitly specified here.

Let us assume you have to perform the operation A = B + C, where all three operands are memory operands. In the case of an accumulator-based ISA, where we assume that one of the general-purpose registers is being designated as an accumulator and one of the operands will always be available in the accumulator, you have to initially load one operand into the accumulator and the ADD instruction will only specify the operand's address. In the GPR based ISA, you have three different classifications. In the register memory ISA, one operand has

to be moved into any register and the other one can be a memory operand. In the register – register ISA, both operands will have to move to two registers and the ADD instruction will only work on registers. The memory – memory ISA permits both memory operands. So, you can directly add. In a stack-based ISA, you'll have to first of all push both operands onto the stack and then simply give an add instruction which will add the top two elements of the stack and then store the result in the stack. So, you can see from these examples that you have different ways of executing the same operation, and it obviously depends upon the ISA. Among all these ISAs, it is the register – register ISA that is very popular and used in all RISC architectures.



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 different features that need to be considered when designing the instruction set architecture

- Types of instructions (Operations in the Instruction set)
- Types and sizes of operands
- Addressing Modes
- Addressing Memory
- Encoding and Instruction Formats

Types of Instructions

- Data transfer instructions

Data transfer instructions perform data transfer between the various storage places in the computer system, viz. registers and memory. Since, both the instructions as well as data are stored in memory, the processor needs to read the instructions and data from memory. After processing, the results must be stored in memory. Therefore, two basic operations involving the memory are needed, namely, *Load* (or *Read* or *Fetch*) and *Store* (or *Write*). The Load operation transfers a copy of the data from the memory to the processor and the Store operation moves the data from the processor to memory. Other data transfer instructions are needed to transfer data from one register to another or from/to I/O devices and the processor.

- Data manipulation instructions

Data manipulation instructions perform operations on data and indicate the computational capabilities for the processor. These operations can be arithmetic operations, logical operations or shift operations. Arithmetic operations include addition (with and without carry), subtraction (with and without borrow), multiplication, division, increment, decrement and finding the complement of a number. The logical and bit manipulation instructions include AND, OR, XOR, Clear carry, set carry, etc. Similarly, you can perform different types of shift and rotate operations.

- Program sequencing and control instructions

We generally assume a sequential flow of instructions. That is, instructions that are stored in consequent locations are executed one after the other. However, you have program sequencing and control instructions that help you change the flow of the program. This is best explained with an example. Consider the task of adding a list of *n* numbers. A possible sequence is given below.

```
Move DATA1, R0
Add DATA2, R0
Add DATA3, R0
Add DATAn, R0
Move R0, SUM
```

The addresses of the memory locations containing the *n* numbers are symbolically given as DATA1, DATA2, . . , DATAn, and a separate Add instruction is used to add each Databer to the contents of register R0. After all the numbers have been added, the result is placed in memory location SUM. Instead of using a long list of Add instructions, it is possible to place a single Add instruction in a program loop, as shown below:

```
Move N, R1
Clear R0
LOOP Determine address of “Next” number and add “Next” number
to R0
Decrement R1
Branch > 0, LOOP
Move R0, SUM
```

The loop is a straight-line sequence of instructions executed as many times as needed. It starts at location LOOP and ends at the instruction Branch>0. During each pass through this loop, the address of the next list entry is determined, and that entry is fetched and added to R0. The address of an operand can be specified in various ways, as will be described in the next section. For now, you need to know how to create and control a program loop. Assume that the number of entries in the list, n, is stored in memory location N. Register R1 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R1 at the beginning of the program. Then, within the body of the loop, the instruction, Decrement R1 reduces the contents of R1 by 1 each time through the loop. The execution of the loop is repeated as long as the result of the decrement operation is greater than zero.

You should now be able to understand *branch* instructions. This type of instruction loads a new value into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the *branch target,* instead of the instruction at the location that follows the branch instruction in sequential address order. The branch instruction can be conditional or unconditional. An *unconditional branch* instruction does a branch to the specified address irrespective of any condition. A *conditional branch* instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed. In the example above, the instruction Branch>0 LOOP (branch if greater than 0) is a conditional branch instruction that causes a branch to location LOOP if the result of the immediately preceding instruction, which is the decremented value in register R1, is greater than zero. This means that the loop is repeated as long as there are entries in the list that are yet to be added to R0. At the end of the *n*th pass through the loop, the Decrement instruction produces a value of zero, and, hence, branching does not occur. Instead, the Move instruction is fetched and executed. It moves the final result from R0 into memory location SUM. Some ISAs refer to such instructions as *Jumps*.

The processor keeps track of information about the results of various operations for use by subsequent conditional branch instructions. This is accomplished by recording the required information in individual bits, often called *condition code flags*. These flags are usually grouped together in a special processor register called the *condition code register* or *status register*. Individual condition code flags are set to 1 or cleared to 0, depending on the outcome of the operation performed. Some of the commonly used flags are: Sign, Zero, Overflow and Carry. The call and return instructions are used in conjunction with subroutines. A subroutine is a self-contained sequence of instructions that performs a given computational task. During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program. Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is made back to the main program, through the return instruction.

Interrupts can also change the flow of a program. A program interrupt refers to the transfer of program control from a currently running program to another

service program as a result of an external or internally generated request. Control returns to the original program after the service program is executed. The interrupt procedure is, in principle, quite similar to a subroutine call except for three variations: (1) The interrupt is usually initiated by an internal or external signal apart from the execution of an instruction (2) the address of the interrupt service program is determined by the hardware or from some information from the interrupt signal or the instruction causing the interrupt; and (3) an interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter. Therefore, when the processor is interrupted, it saves the current status of the processor, including the return address, the register contents and the status information called the Processor Status Word (PSW), and then jumps to the interrupt handler or the interrupt service routine. Upon completing this, it returns to the main program. Interrupts are handled in detail in the next unit on Input / Output.

- Input and output instructions

Input and Output instructions are used for transferring information between the registers, memory and the input / output devices. It is possible to use special instructions that exclusively perform I/O transfers, or use memory – related instructions itself to do I/O transfers.

Addressing Modes

The way the operands are chosen during program execution is dependent on the *addressing mode* of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

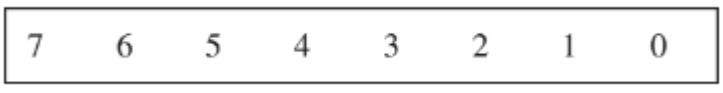
Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4] ← Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4] ← Regs[R4] + 3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4] ← Regs[R4] + Mem[100 + Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1 + R2)`	`Regs[R3] ← Regs[R3] + Mem[Regs[R1] + Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1] ← Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	`Add R1,@(R3)`	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer *p*, then mode yields **p*.
Autoincrement	`Add R1,(R2)+`	`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]` `Regs[R2] ← Regs[R2] + d`	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*.
Autodecrement	`Add R1, –(R2)`	`Regs[R2] ← Regs[R2] – d` `Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1] ← Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers.

Addressing Memory

Basically have two types of interpretation of the memory addresses – Big endian arrangement and the little endian arrangement.

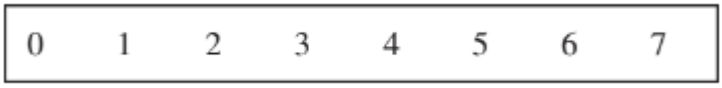
Little Endian byte order puts the byte whose address is “x . . . x000” at the least-significant position in the double word (the little end).

The bytes are numbered:



ie; specify the address of the least significant byte as the address of the word

Big Endian byte order puts the byte whose address is “x . . . x000” at the most-significant position in the double word (the big end). The bytes are numbered:



ie; specify the address of the most significant byte as the address of the word.

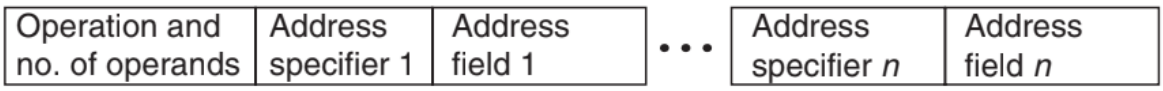
IBM, Motorola, HP follow the big endian arrangement and Intel follows the little endian arrangement. Byte order is a problem when exchanging data among computers with different orderings. Little Endian ordering also fails to match the normal ordering of words when strings are compared. Strings appear “SDRAWKCAB” (backwards) in the registers. A second memory issue is that in many computers, accesses to objects larger than a byte must be aligned. An access to an object of size s bytes at byte address A is aligned if A mod s = 0.

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

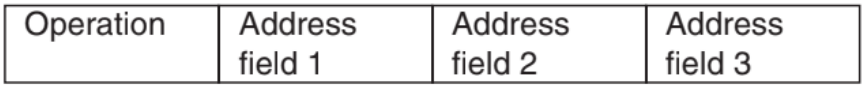
How the instructions are encoded into a binary representation for execution by the processor? This representation affects not only the size of the compiled program but also the implementation of the processor which must decode this representation to quickly find the operation and its operands. The operation is typically specified in one field, called the **opcode**.

There are three popular choices for encoding the instruction set. 1. Variable length: can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. This style is best when there are many addressing modes and operations. 2. Fixed length : always has the same number of operands, with the addressing modes (if

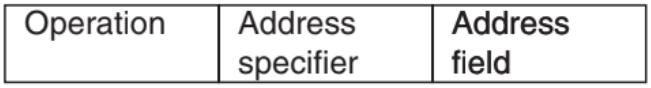
options exist) specified as part of the opcode. It generally results in the largest code size. 3. Hybrid approach: has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address.



(a) Variable (e.g., Intel 80x86, VAX)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)



(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)