Topic: 1.4.3 The processor's instruction set

Instruction set - the range of instructions that a CPU can execute

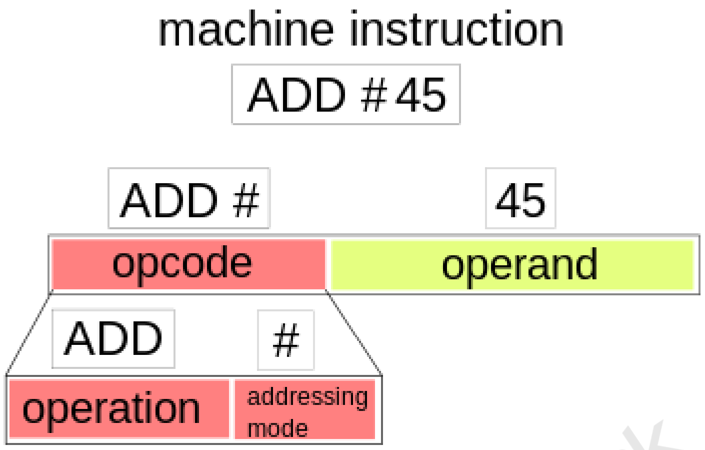
There are many different instructions that we can use in machine code, you have already met three (LDD, ADD, STO), but some processors will be capable of understanding many more. The selection of instructions that a machine can understand is called the instruction set. Below is a list of some other instructions that might be used:

Instruction		Explanation
Op Code	Operand	
LDM	#n	Immediate addressing. Load the number n to ACC
LDD	<address>	Direct addressing. Load the contents of the given address to ACC
LDI	<address>	Indirect addressing. The address to be used is at the given address. Load the contents of this second address to ACC
LDX	<address>	Indexed addressing. Form the address from <address> + the contents of the index register. Copy the contents of this calculated address to ACC
LDR	#n	Immediate addressing. Load the number n to IX
STO	<address>	Store the contents of ACC at the given address
ADD	<address>	Add the contents of the given address to the ACC
INC	<register>	Add 1 to the contents of the register (ACC or IX)
DEC	<register>	Subtract 1 from the contents of the register (ACC or IX)
JMP	<address>	Jump to the given address
CMP	<address>	Compare the contents of ACC with the contents of <address>
CMP	#n	Compare the contents of ACC with number n
JPE	<address>	Following a compare instruction, jump to <address> if the compare was True
JPN	<address>	Following a compare instruction, jump to <address> if the compare was False
IN		Key in a character and store its ASCII value in ACC
OUT		Output to the screen the character whose ASCII value is stored in ACC
END		Return control to the operating system

Topic: 1.4.3 The processor's instruction set

You'll notice that in general, instructions have two main parts:

- **opcode** - instruction name
- **operand** - data or address



Depending on the word size, there will be different numbers of bits available for the opcode and for the operand. There are two different philosophies at play, with some processors choosing to have lots of different instructions and a smaller operand (Intel, AMD) and others choosing to have less instructions and more space for the operand (ARM). Know it now and we will study it in detail in paper 3.

- **CISC** - Complex Instruction Set Computer - more instructions allowing for complex tasks to be executed, but range and precision of the operand is reduced. Some instruction may be of variable length, for example taking extra words (or bytes) to address full memory addresses, load full data values or just expand the available instructions.
- **RISC** - Reduced Instruction Set Computer - less instructions allowing for larger and higher precision operands.

Topic: 1.4.3 The processor's instruction set

Memory Addressing Modes, an Introduction

There are many ways to locate data and instructions in memory and these methods are called 'memory address modes'

Memory address modes determine the method used within the program to access data either from within the CPU or external RAM. Some memory addressing modes can control program flow.

The five memory address modes are:

- Direct
- Indirect
- Immediate
- Indexed
- Relative

Immediate Addressing

Immediate addressing means that the data to be used is hard-coded into the instruction itself.

This is the fastest method of addressing as it does not involve main memory at all.

For example, you want to add 2 to the content of the accumulator

The instruction is:

```
LDM #2
```

Nothing has been fetched from memory; the instruction simply loads 2 to the accumulator immediately.

Immediate Addressing is very useful to carry out instructions involving constants (as opposed to variables). For example you might want to use 'PI' as a constant 3.14 within your code.

Topic: 1.4.3 The processor's instruction set

Direct Addressing

This is a very simple way of addressing memory - direct addressing means the code refers directly to a location in memory

For example

```
LDD 3001
```

In this instance the value held at the direct location 3001 in RAM is loaded to the accumulator.

The good thing about direct addressing is that it is fast (but not as fast as immediate addressing) the bad thing about direct addressing is that the code depends on the correct data always being present at same location.

It is generally a good idea to avoid referring to direct memory addresses in order to have **'re-locatable code'** i.e. code that does not depend on specific locations in memory.

You could use direct addressing on computers that are only running a single program. For example an engine management computer only ever runs the code the car engineers programmed into it, and so direct memory addressing is excellent for fast memory access.

Indirect Addressing

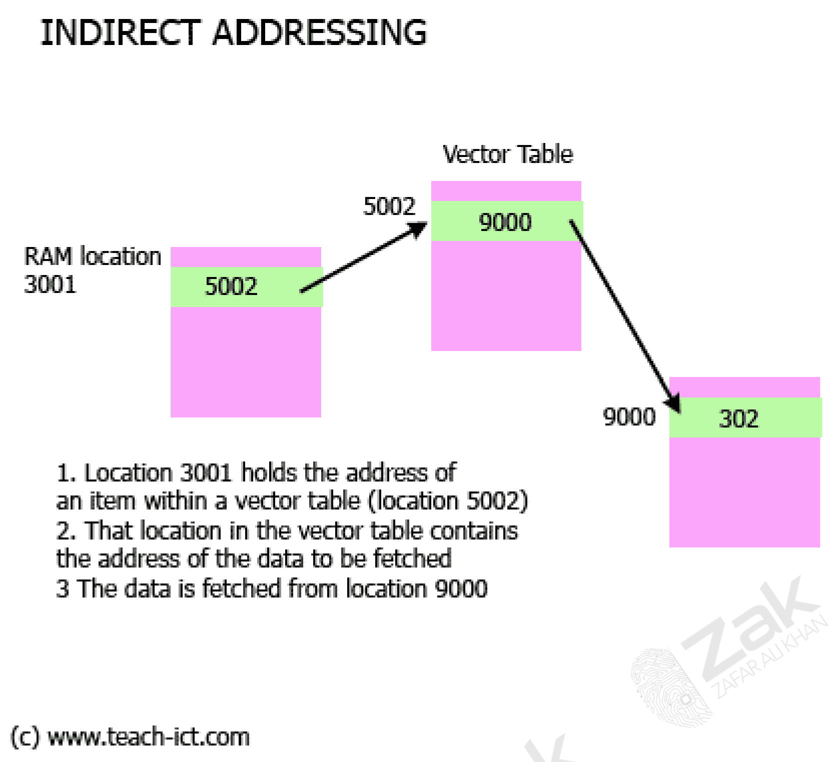
Indirect addressing means that the address of the data is held in an intermediate location so that the address is first 'looked up' and then used to locate the data itself.

Many programs make use of software libraries that get loaded into memory at run time by the loader. The loader will most likely place the library in a different memory location each time.

So how does a programmer access the subroutines within the library if he does not know the starting address of each routine?

Answer: Indirect Addressing

Topic: 1.4.3 The processor's instruction set


INDIRECT ADDRESSING
Vector Table
5002
9000
RAM location 3001
5002
9000
302
1. Location 3001 holds the address of an item within a vector table (location 5002)
2. That location in the vector table contains the address of the data to be fetched
3 The data is fetched from location 9000
(c) www.teach-ict.com

It works like this

1. A specific block of memory will be used by the loader to store the starting address of every subroutine within the library. This block of memory is called a '**vector table**'. A vector table holds addresses rather than data. The application is informed by the loader of the location of the vector table itself.

2. In order for the CPU to get to the data, the code first of all fetches the content at RAM location 5002 which is part of the vector table.

3. The data it contains is then used as the address of the data to be fetched, in this case the data is at location 9000

A typical assembly language instruction would look like

```
LDI 5002
```

This looks to location 5002 for an address. That address is then used to fetch data and load it into the accumulator. In this instance it is 302.

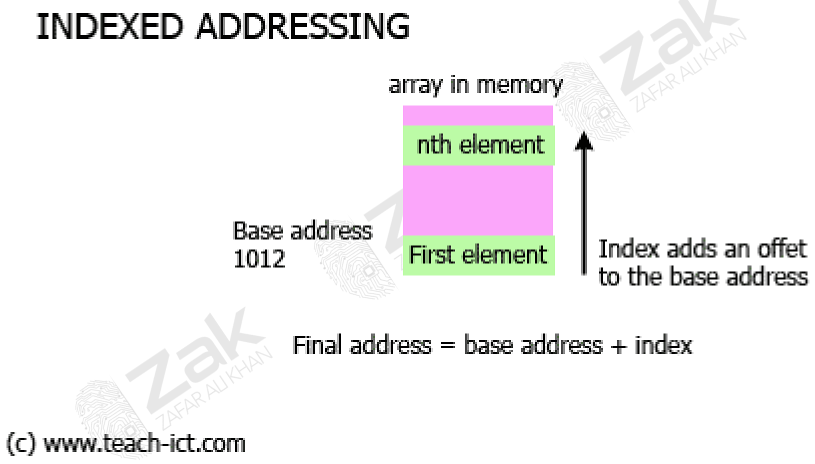
Topic: 1.4.3 The processor's instruction set

Indexed Addressing

Indexed addressing means that the final address for the data is determined by adding an offset to a base address.

Very often, a chunk of data is stored as a complete block in memory.

For example, it makes sense to store arrays as contiguous blocks in memory (contiguous means being next to something without a gap). The array has a **'base address'** which is the location of the first element, then an '**index**' is used that adds an offset to the base address in order to fetch any other element within the array.



Index addressing is fast and is excellent for manipulating data structures such as arrays as all you need to do is set up a base address then use the index in your code to access individual elements.

Another advantage of indexed addressing is that if the array is re-located in memory at any point then only the base address needs to be changed. The code making use of the index can remain exactly the same.

Topic: 1.4.3 The processor's instruction set

Relative Addressing

Quite often a program only needs to jump a little bit in order to jump to the next instruction. Maybe just a few memory locations away from the current instruction.

A very efficient way of doing this is to just add a small offest to the current address in the program counter. (Remember that the program counter always points to the next instruction to be executed). This is called '**relative addressing**'

DEFINITION:

Relative addressing means that the next instruction to be carried out is an offset number of locations away, relative to the address of the current instruction.

Consider this bit of pseudo-code:

```
jump +3 if accumulator == 2
code executed if accumulator is NOT = 2
jmp +5 (unconditional relative jump to avoid the next line of code)

acc:
  code executed if accumulator is = 2)

carryon:
```

In the code snippet above, the first line of code is checking to see if the accumulator has the value of 2 in it. If it is has, then the next instruction is 3 lines away. This is called a **conditional jump** and it is making use of relative addressing.

Another example of relative addressing can be seen in the jmp +5 instruction. This is telling the CPU to effectively avoid the next instruction and go straight to the 'carryon' point; let's say present at this address +5.

Topic: 1.4.3 The processor's instruction set

Summary of memory modes

Type	Comment
Immediate	Apply a constant to the accumulator. No need to access main memory
Direct or Absolute addressing	This is a very simple way of addressing memory - the code refers directly to a location in memory. Disadvantage is that it makes relocatable code more difficult.
Indirect Addressing	Looks to another location in memory for an address and then fetches the data that is located at that address. Very handy of accessing in-memory libraries whose starting address is not known before being loaded into memory
Indexed Addressing	Takes a base address and applies an offset to it and fetches the data at that address. Excellent for handling data arrays
Relative Addressing	Tells the CPU to jump to an instruction that is a relative number of locations away from the current one. Very efficient way of handling program jumps and branching.