Unit 6 Assembly Language

Machine code - Simple instructions that are executed directly by the CPU.

As we should hopefully already know, computers can only understand binary, 1s and 0s. We are now going to look at the simplest instructions that we can give a computer. This is called machine code.

Machine code	Assembly code
000000110101 =	Store 53

Let's take a look at a quick coding example using assembly code.

```
LDA #23; loads the number 23 into the accumulator
ADD #42; adds the number 42 to the contents of the accumulator = 65
STO 34; save the accumulator result to the memory address 34
```

The code above is the equivalent of saying x = 23 + 42 in VB.NET.

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

Below is a list of some other instructions that might be used:

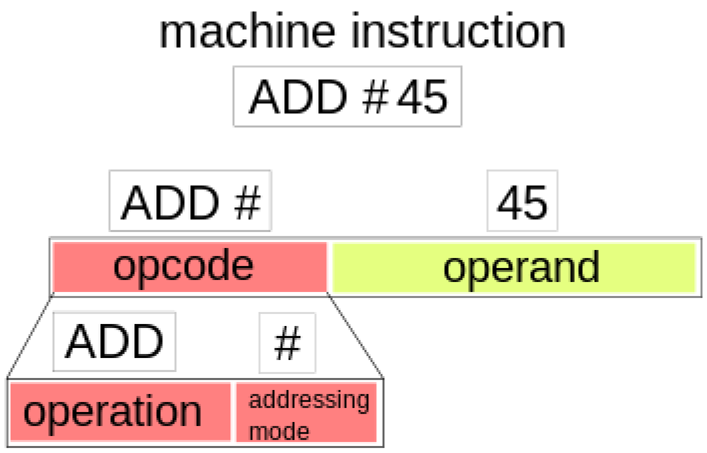
ADD; add one number to another number
SUB; subtract one number to another number
INC; increment a number by 1
DEC; decrement a number by 1
MUL; multiply numbers together
OR; Boolean algebra function
AND; Boolean algebra function
NOT; Boolean algebra function
XOR; Boolean algebra function
JNZ; jump to another section of code if a number is not zero (used for loops and ifs)
JZ; jump to another section of code if a number is zero (used for loops and ifs)
JMP; jump to another section of code (used for loops and ifs)

Let us look at a more complex example of assembly code instructions:

1 LDD #12; loads the number 12 into the accumulator
2 MUL #2; multiplies the accumulator by 2 = 24
3 SUB #6; take 6 away from the accumulator = 18
4 JNZ 6; if the accumulator <> 0 then goto line 6
5 SUB #5; take 5 away from the accumulator (this line isn't executed!)
6 STO 34; saves the accumulator result (18) to the memory address 34

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

- **Opcode** - instruction name.
- **Operand** - data or address.
- **Machine Code Instruction:** a binary code with a defined number of bits that comprises an opcode and, most often, one operand.



Assembly Language: a low-level language related to machine code where opcodes are written as mnemonics and there is a character representation for an operand.
Assembler: a program used to translate an assembly language program into machine code.

Exercise: Instruction sets

Q: What is the instruction set:

A: The range of instructions that a CPU can execute

Q: Name and explain the two parts that make up a machine code instruction:

A: opcode - the command to be executed, operand - the data or address being worked upon

Q: For a word with 4 bits for an opcode and 6 bits for an operand

Q: How many different instructions could I fit into the instruction set? What is the largest number that I could use as data?

A :

- Number of instructions: $2^4 = 16$
- largest operand: $2^6 - 1 = 63$

Q: For a 16 bit word with 6 bits for an opcode, How many different instructions could I fit into the instruction set? What is the largest number that I could use as data?

A:

- Number of instructions: $2^6 = 64$
- largest operand: $2^{10} - 1 = 1023$

Q: Why might a manufacturer choose to increase the instruction set size?

A: So that they can increase the number of discrete instructions that can be executed.

Q: What might be the problem with increasing the space taken up by the opcode?

A: Less space for the operand, meaning reduced range and precision in data be processed in a single instruction.

Q: Give two benefits for increasing the word size of a processor?

A: More space available to increase the instruction set size and greater range and precision available in the operand.

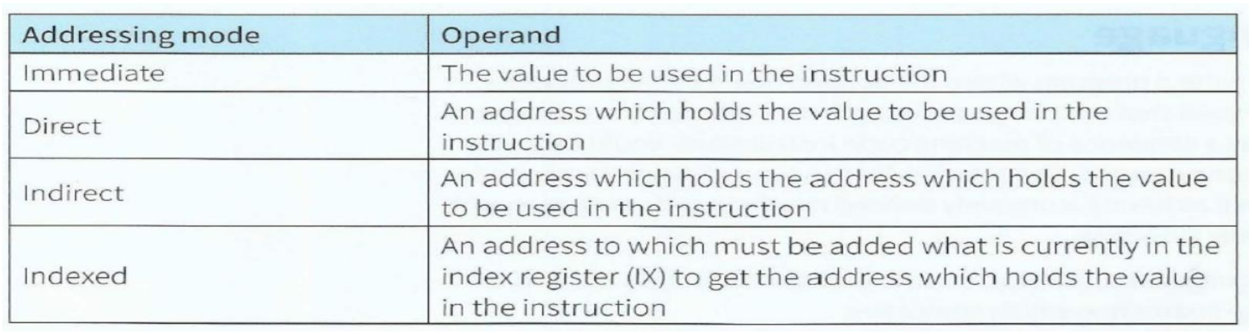
Addressing modes

You might notice that some instructions use a **#** and others don't, you might even have an inkling as to what the difference is. Well here is the truth:

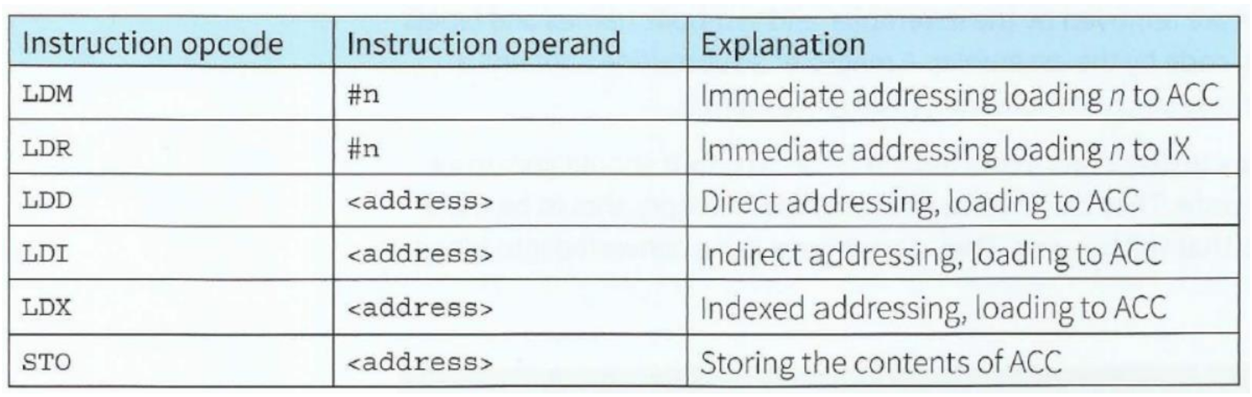
# = number

[no hash] = address

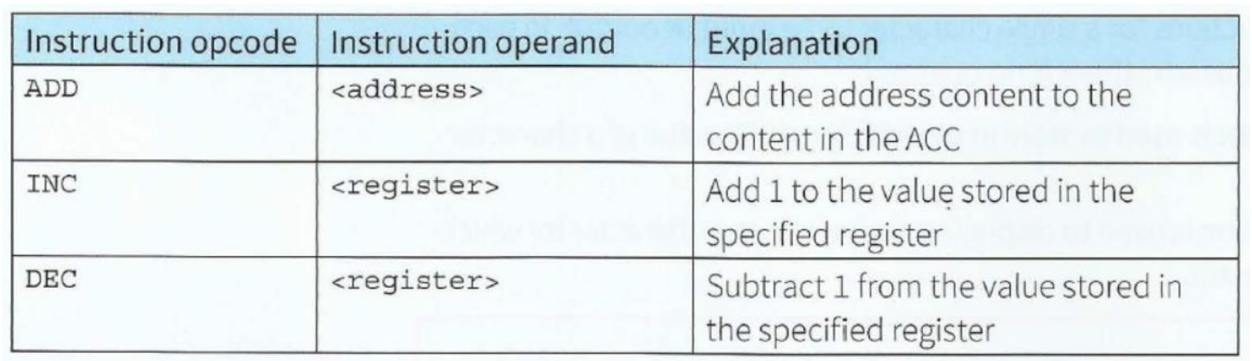
Absolute Addressing: STO 24,
Symbolic Addressing: STO MAX,
Relative Addressing: STO [BR] + 15



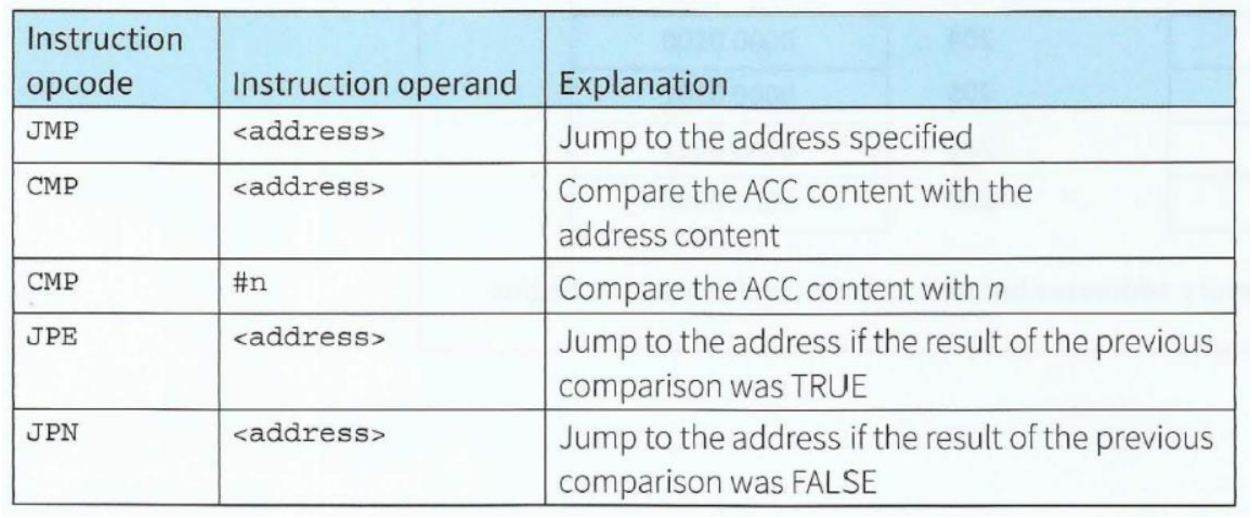
Addressing mode	Operand
Immediate	The value to be used in the instruction
Direct	An address which holds the value to be used in the instruction
Indirect	An address which holds the address which holds the value to be used in the instruction
Indexed	An address to which must be added what is currently in the index register (IX) to get the address which holds the value in the instruction



Instruction opcode	Instruction operand	Explanation
`LDM`	`#n`	Immediate addressing loading *n* to ACC
`LDR`	`#n`	Immediate addressing loading *n* to IX
`LDD`	`<address>`	Direct addressing, loading to ACC
`LDI`	`<address>`	Indirect addressing, loading to ACC
`LDX`	`<address>`	Indexed addressing, loading to ACC
`STO`	`<address>`	Storing the contents of ACC



Instruction opcode	Instruction operand	Explanation
`ADD`	`<address>`	Add the address content to the content in the ACC
`INC`	`<register>`	Add 1 to the value stored in the specified register
`DEC`	`<register>`	Subtract 1 from the value stored in the specified register



Instruction opcode	Instruction operand	Explanation
`JMP`	`<address>`	Jump to the address specified
`CMP`	`<address>`	Compare the ACC content with the address content
`CMP`	`#n`	Compare the ACC content with *n*
`JPE`	`<address>`	Jump to the address if the result of the previous comparison was TRUE
`JPN`	`<address>`	Jump to the address if the result of the previous comparison was FALSE

Special features in an assembly language program.

1. **Comments** (Like other programming languages, assembly language allows comments).
2. **symbolic names for constants**
3. **labels for addresses**
4. **Macros** (Writing a macro is another way of ensuring modular programming in assembly language. A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program).
5. **Subroutines**
6. **Directives** (Directives and system calls are instructions to the assembler as to how it should construct the final executable machine code. They can involve directing how memory should be used or defining files or procedures that will be used. They do not have to be converted into binary code).
7. **System Call**

Single pass Assembler: A **single pass assembler** scans the program only once and creates the equivalent binary program.

Two pass Assembler: Translate assembly code into machine code by processing it in two passes.

What happens during the passes: In the first pass of a two-pass assembler all of these features are either removed or acted upon. Typical actions are:

1. removal of comments
2. creation of a symbol table containing the binary codes for symbolic names and labels
3. creation of a literal table if the programmer has used constants in the program
4. expansion of macros
5. Identification of system calls and subroutines used.

If errors are not found, the second pass of the assembler generates the object code. This involves replacing symbolic addresses with absolute addresses.
To achieve this during the first pass the assembler uses a symbol table. The code is read line by line. When a symbolic address is met for the first time its name is entered into the symbol table. Alongside the name a corresponding address has to be added as soon as that can be identified. Below table shows a possible format for the symbol table that would be created for the a program.

Symbol	Offset
MAX	+15
TOTAL	+16
COUNT	+17
STRTLP	+7

Note that the assembler has to count the instructions as it reads the code. Then when it encounters a label it can enter the offset value into the symbol table. In this example the first entry made in the offset column is the +7 for STRPLP.

For the second pass the Assembler uses the symbol table and a lookup table that contains the binary code for each opcode. This table would have an entry for every opcode in the set defined for the processor. Table 6.04 shows entries only for the instructions used in the simple program we are using as an example. Note that the binary codes are just suggestions of codes that might be used.

Opcode mnemonic	Opcode binary
IN	0001 0000
SUB	0110 0001
STO	0100 0100
LDM	0010 0001
ADD	0100 0101
LDD	0010 0101
INC	0101 0101
CMP	1000 0100
JPN	1010 0100
END	1111 1111

Provided that no errors have been identified, the output from the second pass will be a machine code program.

Some points to note are as follows.

1- Most of the instructions have an operand which is a 16-bit binary number.
2- Usually this represents an address but for the SUB and LDM instructions the operand is used as a value.
3- There is no operand for the IN and END instructions.
4- The INC instruction is a special case. There is an operand in the assembly language code but this just identifies a register. In the machine code the register is identified within the opcode so no operand is needed.
5- The machine code has been coded with the first instruction occupying address zero.
6- This code is not executable in this form but it is valid output from the assembler.
7- Changes will be needed for the addresses when the program is loaded into memory ready for it to be executed.
8- Three memory locations following the program code have been allocated a value zero to ensure that they are available for use by the program when it is executed.

Shift Operations
There are two shift instructions available:
• LSL #n
where the bits in the accumulator are shift ed logically n places to the left
• LSR #n
where the bits are shifted to the right.

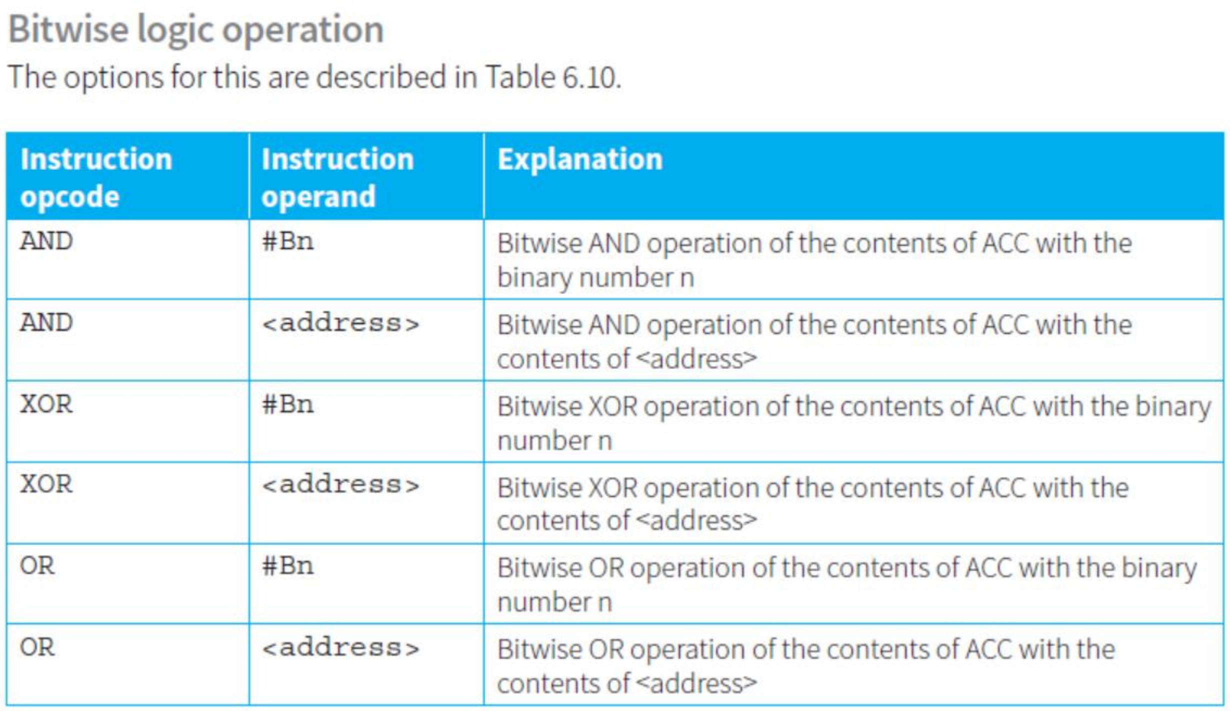
In a logical shift no consideration is given as to what the binary code in the accumulator represents. Because a shift operation moves a bit from the accumulator into the carry bit.

Logical Shift: Where bits in the accumulator are shifted to the right or to the left and zero moves into the bit position vacated.

In a logical shift no consideration is given as to what the binary code in the accumulator represents. Because a shift operation moves a bit from the accumulator into the carry bit in the status register this can be used to examine individual bits. For a left logical shift, the most significant bit is moved to the carry bit, the remaining bits are shifted left and a zero is entered for the least significant bit. For a right logical shift, it is the least significant bit that is moved to the carry bit and a zero is entered for the most significant bit. If the accumulator content represents an unsigned integer, the left shift operation is a fast way to multiply by two. However, this only gives a correct result if the most significant bit is a zero. For an unsigned integer the right shift represents integer division by two. For example, consider: 00110001 (denary 49) gives if right shift ed 00011000 (denary 24)
The remainder from the division can be found in the carry bit. Again, the division will not always give a correct result; continuing right shifts will eventually produce a zero for every bit. It should be apparent that a logical shift cannot be used for multiplication or division by two when a signed integer is stored. This is because the operation may produce a result where the sign of the number has changed. As indicated earlier, only the two logical shift s are available for the simple processor considered here. However, in more complex processors there is likely to be a cyclic shift capability. Here a bit moves off one end into the carry bit then one step later moves in at the other end. All bit values in the original code are retained. Left and right arithmetic shift s are also likely to be available. These work in a similar way to logical shift s, but are provided for the multiplication or division of a signed integer by two. The sign bit is always retained following the shift.

In more complex processors there is likely to be a cyclic shift capability. Here a bit moves off one end into the carry bit then one step later moves in at the other end. All bit values in the original code are retained. Left and right arithmetic shifts are also likely to be available. These work in a similar way to logical shifts, but are provided for the multiplication or division of a signed integer by two. The sign bit is always retained following the shift.

Cyclic Shift: Similar to a logical shift but bits shift ed from one end reappear at the other end.
Arithmetic Shift: Uses the shift to carry out multiplication or division of a signed integer stored in the accumulator.



Bitwise logic operation

The options for this are described in Table 6.10.

Instruction opcode	Instruction operand	Explanation
AND	#Bn	Bitwise AND operation of the contents of ACC with the binary number n
AND	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>
XOR	#Bn	Bitwise XOR operation of the contents of ACC with the binary number n
XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>
OR	#Bn	Bitwise OR operation of the contents of ACC with the binary number n
OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address>

Register Transfer Notation:

Section 6.01 introduced an extension to register transfer notation. We can use this to describe the execution of an instruction. For example, the LDD instruction is described by:

ACC ← [[CIR(15:0)]]

The instruction is in the CIR and only the 16-bit address needs to be examined to identify the location of the data in memory. The contents of that location are transferred into the accumulator.

Computer Arithmetic

In Chapter 1 we saw that computer arithmetic could lead to an incorrect answer if overflow occurred. In Chapter 5 we saw the possible uses of the Status Register. The following worked example illustrates how the values stored in the Status

Register can identify a specific overflow condition.
The use of the following three flags is required:

- the carry flag, identified as C, which is set to 1 if there is a carry.
- the negative flag, identified as N, which is set to 1 if a result is negative.
- the overflow flag, identified as V, which is set to 1 if overflow is detected.

WORKED EXAMPLE 6.01

Using the status register during an arithmetic operation

1 Consider the addition of two positive values where the sum of the two produces an answer that is too large to be correctly identified with the limited number of bits used to represent the values. For example, Figure 6.04 shows what happens if we use an eight-bit binary integer representation and attempt to add denary 66 to denary 68.

```
        0100 0010
+       0100 0100
            |
            |          Flags: N V C
            v                 1 1 0
        1000 0110
```

Figure 6.04 An attempted addition of denary 66 to denary 68

The answer produced is denary −122. Two positive numbers have been added to get a negative number. This impossibility is detected by the combination of the negative flag and the overflow flag being set to 1. The processor examines the flags, identifies the problem and generates an interrupt.

2 Consider using the same eight-bit binary integer representation but this time we add two negative numbers (−66 and −68 in denary). The result is shown in Figure 6.05.

```
           1011 1100
+          1011 1110
               |
               |       Flags: N V C
               v              0 1 1
        (1)0111 1010
```

Figure 6.05 An attempted addition of denary −66 to denary −68

We get the answer +122. This impossibility is detected by the combination of the negative flag not being set and both the overflow and the carry flag being set to 1.