4.2 Assembly Language EMK

Objective:

- Show understanding of relationship between assembly language and machine code.
- Describe different stages of assembly process for a two-pass assembler.
- Apply two-pass assembler process to a given simple assembly language program.
- Trace a given simple assembly language program
- Show understanding that a set of instructions are grouped;
  - Data movement • Input and output of data • Arithmetic operations • Unconditional and conditional instructions • Compare instructions.
- Modes of addressing, Including Immediate, direct, indirect, indexed, relative.

Machine Code

Only programming language that CPU can use is **Machine Code**. Every different type of computer chip has its **own** set of machine code instructions. Each machine code instruction performs **one simple task**, for example, storing a value in a memory location at specified address.

Machine code is **binary**, it is sometimes displayed on a screen as **hexadecimal** so that **human** programmers can understand machine code instructions more **easily**.

Machine code instruction is a **binary code** with **defined** number of bits that comprises an **opcode** and, most often, **operand**.

- **Opcode** defines **action** associated with instruction.
- **Operand** defines any **data** needed by instruction.

Writing programs in machine code is **specialized task** that is very **time consuming** and **error prone**. In order to shorten **development time** for writing programs, other programming languages were developed, where instructions were **easier** to **learn** and **understand**.

Feature of Machine Instructions

- Machine code consists of **sequence** of instructions and each instruction contains **opcode**.
- An instruction may not have an **operand** but up to **three** operands are possible.
- Different processors have different **instruction sets** associated with them.
- Different processors will have **comparable instructions** for same operations, but coding of instructions will be different.

For Each machine code instruction, programmer need to consider following points:

- Total number of **bits or bytes** for whole instruction.
- Number of bits that define **opcode**.
- Number of **operands** that are defined in remaining bits.
- Whether **opcode** occupies most significant or least significant bits.

Opcode			Operand
Operation	Address mode	Register addressing	
4 bits	2 bits	2 bits	16 bits

Assembly Language

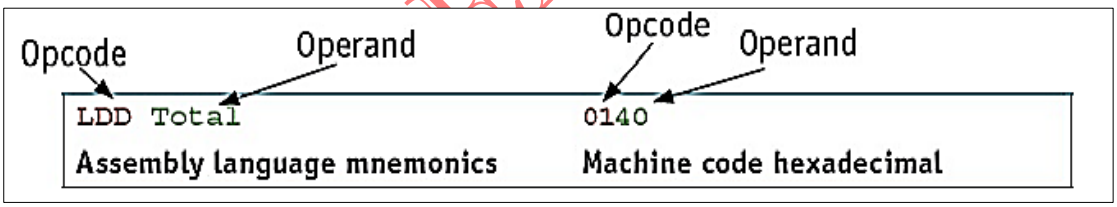
Assembly language is **low-level language** related to **machine code** where **opcodes** are written as **mnemonics** and there is a **character representation** for an **operand**. Each processor has its own assembly language.

If a program has been written in assembly language it has to be **translated** into machine code using translator before it can be executed by processor. **Assembler** is a language translator used to translate an **assembly language** program into **machine code**.

LDD Total	0140	0000000110000000
ADD 20	0214	0000001000011000
STO Total	0340	0000001110000000
Assembly language mnemonics	**Machine code hexadecimal**	**Machine code binary**

Structure of Assembly Language:

Structure of assembly language and machine code instructions is same. Each instruction has an **opcode** that **identifies operation** to be carried out by CPU. Most instructions also have an **operand** that identifies data to be used by **Opcode**.



Assembler translates **assembly language** instruction into a **machine code** instruction. It checks **Syntax** of assembly language program to ensure that only **opcodes** from appropriate machine code instruction set are used. This speeds up **development time**, as some **errors** are identified during translation before program is executed.

Assembly Language Instructions

Data Movement instructions

Data Movement instructions allow data stored at **one location** to be **copied** into **accumulator**. This data can then be stored at another location, used in a calculation, used for a comparison or output.

ACC is single accumulator, **IX** is the Index Register, All numbers are denary unless identified as binary or hexadecimal**. B** is a binary number, for example B01000011.
& is a hexadecimal number, for example &7B**. #** is a denary number.

Instruction Opcode	Instruction Operand	Explanation
LDD	<address>	**Direct Addressing**: load content of <address> to ACC
LDI	<address>	**Indirect Addressing**: <address> hold the address to be used; load content of this second address to ACC
LDX	<address>	**Indexed Addressing**: form the address for <address> plus the content of IX; copy content of this calculated address to ACC.
LDR	#n	**Immediate Addressing**: load number **n** to IX
LDR	ACC	Load the number in accumulator into IX
LDM	#n	**Immediate addressing**; load number **n** to ACC
MOV	<register>	Move the contents of the accumulator to the register (IX)
STO	<address>	Store contents of ACC into specified address (direct or absolute addressing is used)

Input and Output of data instructions

These instructions allow data to be **read** from the **keyboard** or output to the screen.

Instruction		Explanation
Opcode	**Operand**	
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
No opcode is required as a single character is either input to the accumulator or output from the accumulator		

Arithmetic Operation Instructions

These instructions perform simple calculations on data stored in the accumulator and store answer in the accumulator, overwriting the original data.

Instruction		Explanation
Opcode	Operand	
ADD	<address>	Add the contents of the specified address to the ACC (direct or absolute addressing is used)
ADD	#n	Add the denary number n to the ACC
SUB	<address>	Subtract the contents of the specified address from the ACC
SUB	#n	Subtract the number n from 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)
Answers to calculations are always stored in the accumulator		

Unconditional, Conditional and Compare instructions

Jump means change PC to the address specified, so next instruction to be executed is one stored at the specified address, not the one stored at the next location in memory.

Instruction Opcode	Instruction Operand	Explanation
JMP	<address>	Unconditionally; Jump to the address specified
JPN	<address>	Follow the compare instruction, jump to address if the compare was false.
JPE	<address>	Follow the compare instruction, jump to address if the compare was True.
CMP	<address>	Compare the contents of ACC with the contents of <address>
CMP	#n	Compare the contents of ACC with the number n.
CMI	<address>	Address to be used is the contents of the specified address; compare the contents of the contents of the given address with ACC (indirect addressing is used)
END		Returns control to the operating system

Addressing Modes

An **addressing mode** defines how a value should be **found** that has to be loaded into a register. When an instruction requires a value to be loaded into register there are **different ways** of identifying the value. Each one is known as an **addressing mode**.

Assembly language and machine code programs use different **addressing modes** depending on requirements of program.

Direct Addressing:

Contents of memory location in **operand** are used. Direct addressing are also known **absolute addressing**.

Example, if memory location with **address** 200 contained value 78, assembly language instruction **LDD 200** would store 78 in accumulator.

Indirect Addressing:

Contents of the contents of the **memory location** in the operand are used.

Example: If memory location with address 200 contained the value 20 and the memory location with address 20 contained the value 5, the assembly language instruction **LDI 200** would store 5 in the accumulator.

Indexed Addressing:

Contents of memory location found by adding contents of index register (IR) to the address of the memory location in the operand are used.

Example: if **IR** contained value 4 and memory location with address 204 contained value 17, assembly language instruction **LDX 200** would store 17 in accumulator.

Immediate Addressing:

The operand is the value to be used in the instruction. For immediate addressing there are **three options** for defining the value:

- #48 specifies the **denary** value 48
- #**B**00110000 specifies the **binary** equivalent
- #**&**30 specifies the **hexadecimal** equivalent

Example: Assembly language instruction **LDM #200** would store 200 in accumulator.

Relative Addressing:

Memory address used is current memory address added to the operand.

Example, JMR #5 would transfer control to the instruction 5 locations after the current instruction.

Symbolic Addressing:

It is only used in assembly language programming. A label is used instead of a value. Labels make it easier to alter assembly language programs.

Example, if memory location with address labelled **MyStore** contained the value 20, assembly language instruction **LDD MyStore** would store 20 in accumulator.

A program written in assembly language will need many more instructions than a program written in a high-level language to perform the same task.

Bitwise Logic

Operand for bitwise **logic operation** instruction is referred to as a **mask** because it can effectively cover some of bits and only affect **specific bits**.

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>

Computer Arithmetic:

Computer arithmetic could lead to an **incorrect answer** if **overflow** occurred. Values stored in **Status Register** can identify a specific overflow condition.

Use of following three flags is required:

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

Tracing an Assembly language

100	IN
101	STO 200
102	IN
103	STO 201
104	IN
105	ADD 200
106	STO 200
107	ADD 201
108	INC ACC
109	OUT
110	END

Tracing is based on initial user input of 15, second input of 27 and a final input of 31.

Accumulator	Memory location 200	Memory location 201	Output
15			
	15		
27			
		27	
31			
46			
	46		
73			
74			
			74

Current contents of main memory and selected values from ASCII character set are given.

(i) Trace program currently in memory using trace table.

Instruction address	ACC	IX	Memory address					Output
			100	101	110	111	112	
			0	0	66	65	35	
77		0						
78	66							
79								
80								
81								
82				66				
83	1							
84								
85			1					
86		1						
87	65							
88								
89								
81	66							
82								
83	1							
84	2							
85			2					
86		2						
87	35							
88								
89								
90	2							
91	50							
92								2
93								

Address	Instruction
77	LDR #0
78	LDX 110
79	CMP #35
80	JPE 92
81	ADD 100
82	STO 101
83	LDM #1
84	ADD 100
85	STO 100
86	INC IX
87	LDX 110
88	CMP #35
89	JPN 81
90	LDD 100
91	ADD #48
92	OUT
93	END
...	~
100	0
101	0
...	~
110	66
111	65
112	35

ASCII value	Character
49	1
50	2
51	3
52	4
~	~
65	A
66	B
67	C
68	D

Instruction		Explanation
Opcode	**Operand**	
LDM	#n	Immediate addressing. Load the number n to ACC
LDD	<address>	Direct addressing. Load the contents of the location at the given 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 contents of ACC at the given address
ADD	<address>	Add the contents of the given address to the ACC
ADD	#n	Add the denary number n to the ACC
INC	<register>	Add 1 to the contents of the register (ACC or IX)
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
OUT		Output to the screen the character whose ASCII value is stored in ACC
END		Return control to the operating system
<address> can be an absolute or a symbolic address # denotes a denary number, e.g. #123 B denotes a binary number, e.g. B01001101		

****