



Computer Architecture

2024 – 2025 Academic Year

CEIT – 51023 Computer Architecture I

(Fifth Year – First Semester)

Dr. Theingi Myint

Professor

Department of Computer Engineering and Information Technology

Mandalay Technological University

Contents

Chapter (1): Digital Logic Circuits

Chapter (2): Digital Components

Chapter (4): Register Transfer and Microoperations

Chapter (6): Programming the Basic Computer

Chapter (8): Central Processing Unit

Chapter Six: Programming the Basic Computer

- Utilizes the 25 instructions of the basic computer to illustrate techniques used in assembly language programming
- Understand instructions of the basic computer to present the data processing tasks

Introduction

- Computer architecture should have a knowledge of both hardware and software because the two branches influence each other.
- Writing a program for a computer consists of specifying, directly or indirectly, a sequence of machine instructions.
- Machine instructions inside the computer form a binary pattern which is difficult, if not impossible, for people to work with and understand.
- It is preferable to write programs with the more familiar symbols of the alphanumeric character set.
- As a consequence, there is a need for translating user-oriented symbolic programs into binary programs recognized by the hardware.

Introduction

Symbol	Hexadecimal code	Description
AND	0 or 8	AND M to AC
ADD	1 or 9	Add M to AC , carry to E
LDA	2 or A	Load AC from M
STA	3 or B	Store AC in M
BUN	4 or C	Branch unconditionally to m
BSA	5 or D	Save return address in m and branch to $m + 1$
ISZ	6 or E	Increment M and skip if zero
CLA	7800	Clear AC
CLE	7400	Clear E
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right E and AC
CIL	7040	Circulate left E and AC
INC	7020	Increment AC ,
SPA	7010	Skip if AC is positive
SNA	7008	Skip if AC is negative
SZA	7004	Skip if AC is zero
SZE	7002	Skip if E is zero
HLT	7001	Halt computer
INP	F800	Input information and clear flag
OUT	F400	Output information and clear flag
SKI	F200	Skip if input flag is on
SKO	F100	Skip if output flag is on
ION	F080	Turn interrupt on
IOF	F040	Turn interrupt off

- It introduces some elementary programming concepts and shows their relation to the hardware representation of instructions.
- Many of the techniques commonly used to program a computer are illustrated.
- It is possible to explore **the relationship between a program and the hardware operations** that execute the instructions.
- **25 instructions** of the basic computer are to provide an easy reference for the programming.

Introduction

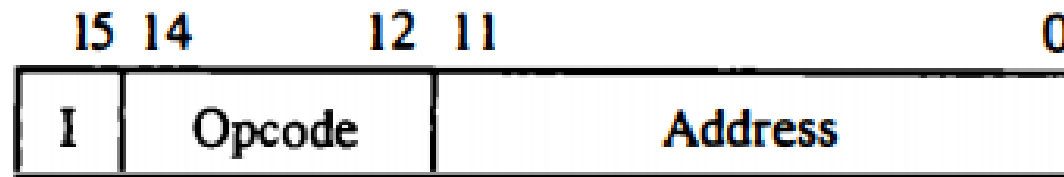
Symbol	Hexadecimal code	Description
AND	0 or 8	AND M to AC
ADD	1 or 9	Add M to AC , carry to E
LDA	2 or A	Load AC from M
STA	3 or B	Store AC in M
BUN	4 or C	Branch unconditionally to m
BSA	5 or D	Save return address in m and branch to $m + 1$
ISZ	6 or E	Increment M and skip if zero
CLA	7800	Clear AC
CLE	7400	Clear E
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right E and AC
CIL	7040	Circulate left E and AC
INC	7020	Increment AC ,
SPA	7010	Skip if AC is positive
SNA	7008	Skip if AC is negative
SZA	7004	Skip if AC is zero
SZE	7002	Skip if E is zero
HLT	7001	Halt computer
INP	F800	Input information and clear flag
OUT	F400	Output information and clear flag
SKI	F200	Skip if input flag is on
SKO	F100	Skip if output flag is on
ION	F080	Turn interrupt on
IOF	F040	Turn interrupt off

memory-reference
instructions

- Each instruction is assigned a **three-letter symbol** to facilitate writing symbolic programs.
- The first **7** instructions are **memory-reference instructions**, and the other **18** are **register-reference** and **input-output** instructions.

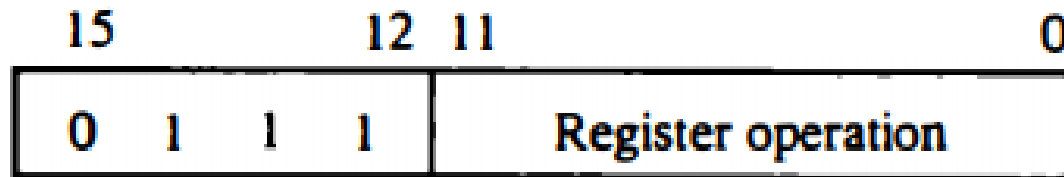
register-reference and input-
output instructions

Introduction



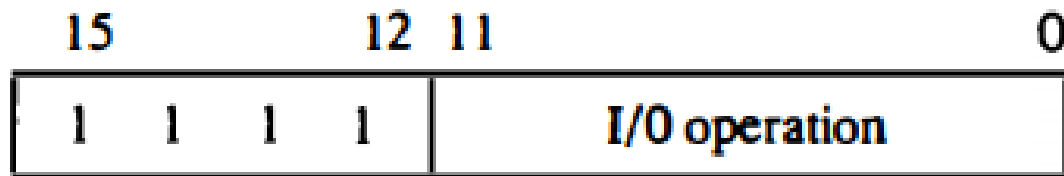
(Opcode = 000 through 110)

(a) Memory – reference instruction



(Opcode = 111, I = 0)

(b) Register – reference instruction

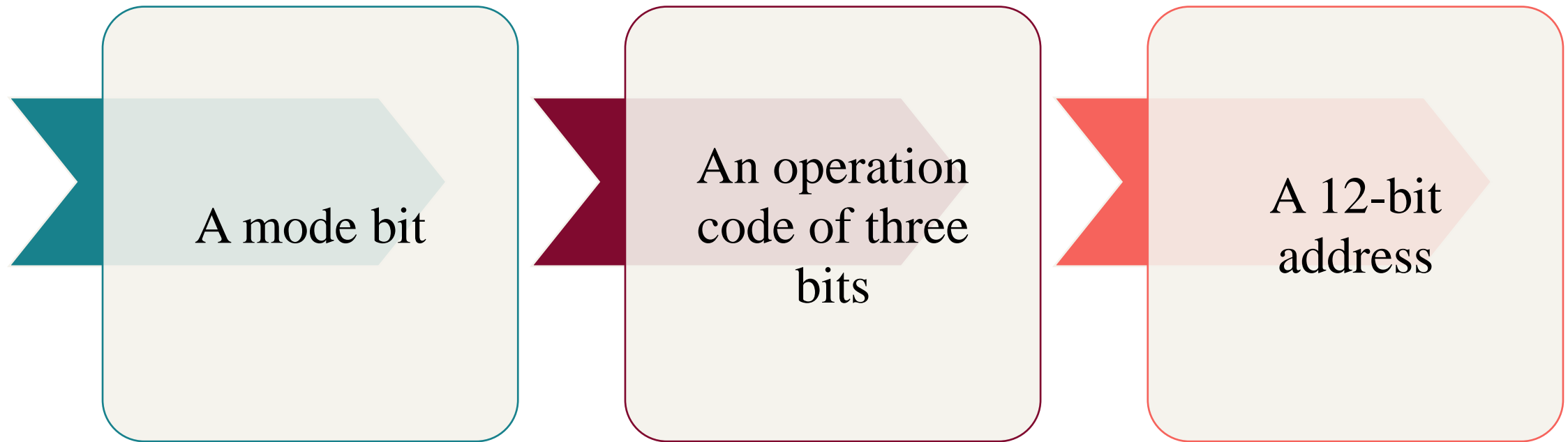


(Opcode = 111, I = 1)

(c) Input – output instruction

Introduction

- A memory-reference instruction has **three parts**:



Introduction

- In an **indirect** address instruction, the **mode bit is 1** and the first **hexadecimal digit** ranges in value **from 8 to E** (1000 to 1110).
- In a **direct** mode, the range is **from 0 to 6** (0000 to 0110).
- The other **18 instructions** have a **16-bit operation code**.
- The code for each instruction is listed as a **four-digit hexadecimal number**.
- The **first digit** of a **register-reference instruction** is always **7**.
- The **first digit** of an **input-output instruction** is always **F**.
- The symbol *m* used in the description column denotes the **effective address**.
- The letter *M* refers to the **memory word** (operand) found at the effective address.

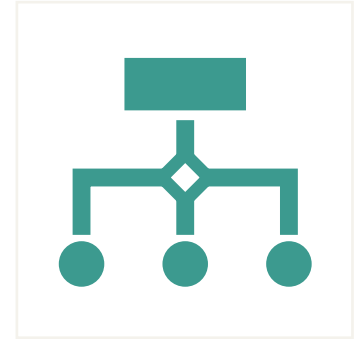
Machine Language



A **program** is a list of instructions or statement for directing the computer to perform a required data-processing task.



There are various types of programming languages that one may **write** for a computer, but the computer can **execute** programs only when they are represented internally in binary form.



Programs written in any other language must be translated to the **binary representation of instructions** before they can be executed by the computer.

Machine Language

- Programs written for a computer may be in one of the following categories:

Binary code

- This is **a sequence of instructions and operands in binary** that list the exact representation of instructions as they appear in computer memory.

Octal or Hexadecimal code

- This is **an equivalent translation of the binary code** to octal or hexadecimal representation.

Symbolic code

- The user employs **symbols** (letters, numerals, or special characters) for the operation part, the address part, and other parts of the instruction code.
- This translation is done by a special program called **an assembler**.
- Because **an assembler translates the symbols**, this type of **symbolic program** is referred to as **an assembly language program**.

Machine Language

- Programs written for a computer may be in one of the following categories:

High-level programming languages

- These are special languages developed to reflect the procedures used in the solution of a problem rather than be concerned with the computer hardware behavior.
- The program is written in **a sequence of statements** in a form that people prefer to think in when solving a problem.
- Each statement must be translated into **a sequence of binary instructions** before the program can be executed in a computer.
- The program that translates a high-level language program to binary is called **a compiler**.

Machine Language

- Basic computer to illustrate the relation between binary and assembly languages:
 - The first column gives the **memory location (in binary)** of each instruction or operand.
 - The second column lists the **binary content of these memory locations**.
 - Location is the address of the memory word where the instruction is stored.
 - It is important to differentiate it from the address part of the instruction itself.
 - The program can be stored in the indicated portion of memory, and then executed by the computer starting from address 0.
- Binary program to add two numbers

Location	Instruction code
0	0010 0000 0000 0100
1	0001 0000 0000 0101
10	0011 0000 0000 0110
11	0111 0000 0000 0001
100	0000 0000 0101 0011
101	1111 1111 1110 1001
110	0000 0000 0000 0000

Machine Language

- Writing 16 bits for each instruction
- We can reduce the **number of digits per instruction** if we write the octal equivalent of the binary code.
- This will require **six digits per instruction**.
- We can reduce each instruction to **four digits** if we write **the equivalent hexadecimal code**.

Hexadecimal program to add two numbers

Location	Instruction
000	2004
001	1005
002	3006
003	7001
004	0053
005	FFE9
006	0000

Machine Language

- The symbolic names of instructions instead of their binary or hexadecimal equivalent.
- The address parts of memory-reference instructions, as well as operands, remain in their hexadecimal value.
- Location **005** has a negative operand because the **sign bit in the leftmost position is 1**.
- The inclusion of a column for comments provides some means for explaining the function of each instruction.
- Symbolic programs are easier to handle, and as a consequence, it is preferable to write programs with symbols.

Program with Symbolic Operation Codes

Location	Instruction	Comments
000	LDA 004	Load first operand into AC
001	ADD 005	Add second operand to AC
002	STA 006	Store sum in location 006
003	HLT	Halt computer
004	0053	First operand
005	FFE9	Second operand (negative)
006	0000	Store sum here

- These symbols can be converted to their binary code equivalent to produce the binary program.

Machine Language

- The **symbol ORG** followed by a number is not a machine instruction.
 - Its purpose is to specify an **origin**, that is, the memory location of the next instruction below it.
 - The next three lines have **symbolic addresses**.
 - Their value is specified by their being present as a label in the first column.
 - **Decimal operands** are specified following the symbol **DEC**.
 - The numbers may be **positive or negative**, but if negative, they must be converted to **binary in the signed-2's complement representation**.
- Assembly Language Program to Add Two Numbers
- | | |
|------------|----------------------------------|
| ORG 0 | /Origin of program is location 0 |
| LDA A | /Load operand from location A |
| ADD B | /Add operand from location B |
| STA C | /Store sum in location C |
| HLT | /Halt computer |
| A, DEC 83 | /Decimal operand |
| B, DEC -23 | /Decimal operand |
| C, DEC 0 | /Sum stored in location C |
| END | /End of symbolic program |
- The last line has the symbol **END** indicating the end of the program.
 - The symbols ORG, DEC, and END, called **pseudo instructions**, are defined.
 - All comments are preceded by a **slash**.

Assembly Language



The basic unit of an assembly language program is **a line of code**.



The specific language is defined by **a set of rules that** specify the symbols that can be used and how they may be combined to form a line of code.

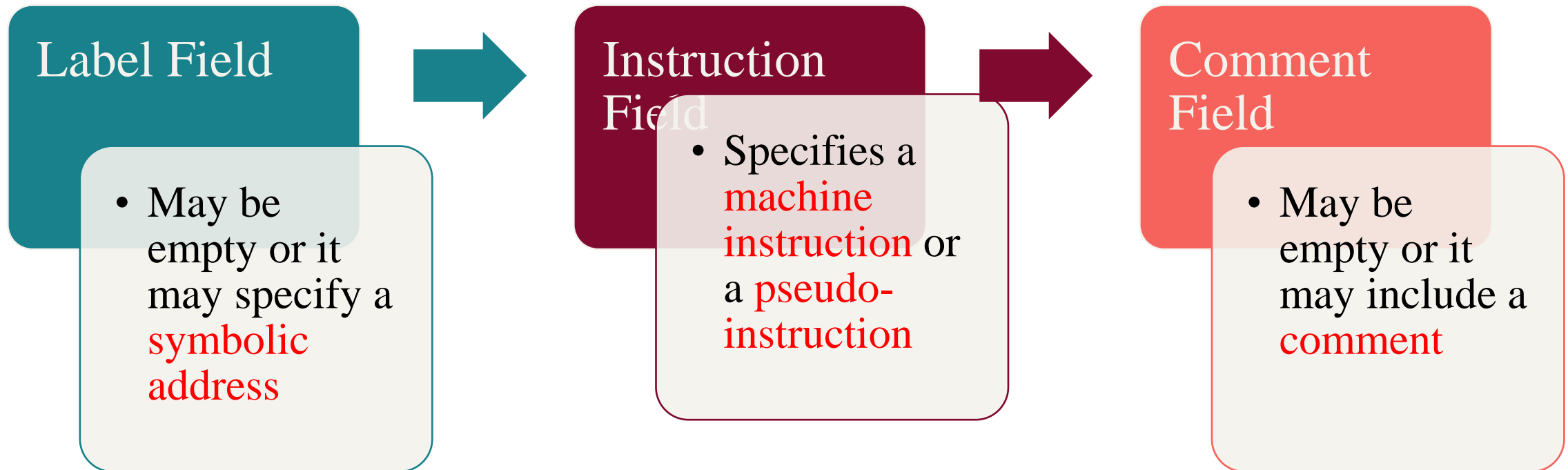


The rules of an assembly language for writing symbolic programs for the basic computer are formulated.

Assembly Language

Rules of the Language

- Each line of an assembly language program is arranged in three columns called **fields**.
- The fields specify the following information:



Assembly Language

Symbolic Address

- A **symbolic address** consists of one, two, or three, but not more than three alphanumeric characters.
- The **first character** must be **a letter**, the **next two** may be **letters or numerals**.
- The symbol can be chosen arbitrarily by the programmer.
- A symbolic address in the label field is terminated by **a comma** so that it will be recognized as a label by the assembler.

Assembly Language

- The **instruction field** in an assembly language program may specify one of the following items:



Assembly Language

Memory-reference instruction (MRI)

- A **memory-reference instruction** occupies **two or three symbols** separated by spaces.
- The first must be a **three-letter symbol** defining an MRI operation code.
- The second is a **symbolic address**.
- The third symbol, which may or may not be present, is **the letter**.
- If ***I*** is missing, the line denotes a **direct address** instruction.
- The presence of the symbols ***I*** denotes an **indirect address** instruction.

Assembly Language

Register-reference or input-output instruction (non-MRI)

- A **non-MRI** is defined as an instruction that **does not have an address part**.
- A non-MRI is recognized in the instruction field of a program by any one of the three-layer symbols for register-reference and input-output instructions.
- An illustration of the symbols that may be placed in the instruction field of a program:

CLA	non-MRI
ADD OPR	direct address MRI
ADD PTR I	indirect address MRI

- The first three-letter symbol in each line must be **one of the instruction symbols** of the computer.
- A memory-reference instruction, such as ADD, must be followed by **a symbolic address**.
- The letter **I** may or may not be present.

Assembly Language

Register-reference or input-output instruction (non-MRI)

- A **symbolic address** in the instruction field specifies the memory location of an operand.
- This location must be defined somewhere in the program by **appearing again as a label in the first column**.
- To be able to translate an assembly language program to a binary program, it is absolutely necessary that **each symbolic address** that is mentioned in the instruction field must occur **again in the label field**.

Assembly Language

Pseudoinstruction

- A **pseudoinstruction** is not a machine instruction but rather than an instruction to the assembler giving information about some phase of the translation.
- **Four pseudoinstructions** that are recognized by the assembler are shown in Table.
- The **ORG (origin)** pseudoinstruction informs the assembler that the instruction or operand in the following line is to be placed in a memory location specified by **the number next to ORG**.
- It is possible to use ORG **more than once in a program** to specify more than one segment of memory.

Symbol	Information for the Assembler
ORG N	Hexadecimal number N is the memory location for the instruction or operand listed in the following line
END	Denotes the end of symbolic program
DEC N	Signed decimal number N to be converted to binary
HEX N	Hexadecimal number N to be converted to binary

Assembly Language

Pseudoinstruction

- The **END symbol** is placed at the end of the program to inform the assembler that the program is terminated.
- The other two pseudoinstructions specify the radix of the operand and tell the assembler how to convert the listed number to a binary number.
- The third field in a program is reserved for comments.
- A line of code may or may not have a comment, but if it has, it must be preceded by a **slash** for the assembler to recognize the beginning of a comment field.
- Comments are useful for explaining the program and are helpful in understanding the step-by-step procedure taken by the program.
- Comments are inserted for explanation purposes only and are neglected during the binary translation process.

Assembly Language

Example: Assembly Language Program to Subtract Two Numbers

- The first line has the pseudoinstruction **ORG** to define the origin of the program at memory location (100).
- The next **six lines** defines **machine instructions**, and the last **four** have **pseudoinstructions**.
- Three symbolic addresses have been used and each is listed in column 1 as a label and in column 2 as an address of a memory-reference instruction.
- Three of the pseudoinstructions specify operands, and the last one signifies the **END** of the program.

machine instructions		ORG 100	/Origin of program is location 100
		LDA SUB	/Load subtrahend to AC
		CMA	/Complement AC
		INC	/Increment AC
		ADD MIN	/Add minuend to AC
		STA DIF	/Store difference
		HLT	/Halt computer
	MIN,	DEC 83	/Minuend
	SUB,	DEC -23	/Subtrahend
	DIF,	HEX 0	/Difference stored here
pseudoinstructions		END	/End of symbolic program

Assembly Language

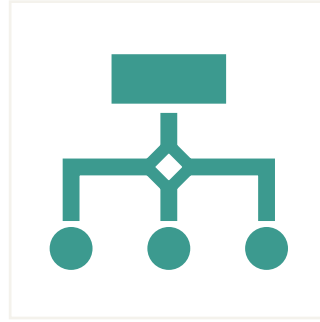
Example: Assembly Language Program to Subtract Two Numbers

- When the program is translated into binary code and executed by the computer it will perform a subtraction between two numbers.
- The subtraction is performed by adding the minuend to the 2's complement of the subtrahend.
- The subtrahend is a negative number.
- It is converted into a binary number in signed-2's complement representation because all negative numbers be in their 2's complement form.
- When the 2's complement of the subtrahend is taken (by complementing and incrementing the AC), -23 converts to +23 and the difference is $83 + (2's \text{ complement of } -23) = 83 + 23 = 106$.

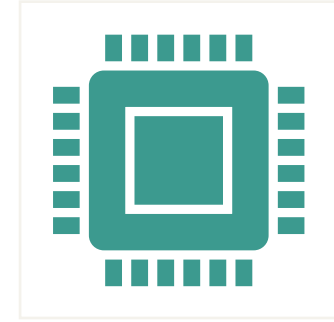
Assembler



An **assembler** is a program that accepts a symbolic language program and produces its binary machine language equivalent.



The input symbolic program is called the **source program**, and the resulting binary program is called the **object program**.



The assembler is a program that operates on character strings and produces an equivalent binary interpretation.

Assembler

Representation of Symbolic Program in Memory

- Prior to starting the assembly process, the symbolic program must be stored in memory.
- The user types the symbolic program on a terminal.
- A **loader program** is used to input the characters of the symbolic program into memory.
- Since the program consists of **symbols**, its representation in memory must use an **alphanumeric character code**.
- In the basic computer, **each character is represented by an 8-bit code**.
- The **high-order bit is always 0** and the **other seven bits** are as specified by **ASCII**.

Assembler

Representation of Symbolic Program in Memory

Hexadecimal Character Code

Character	Code	Character	Code	Character	Code
A	41	Q	51	6	36
B	42	R	52	7	37
C	43	S	53	8	38
D	44	T	54	9	39
E	45	U	55	space	20
F	46	V	56	(28
G	47	W	57)	29
H	48	X	58	*	2A
I	49	Y	59	+	2B
J	4A	Z	5A	,	2C
K	4B	0	30	-	2D
L	4C	1	31	.	2E
M	4D	2	32	/	2F
N	4E	3	33	=	3D
O	4F	4	34	CR	0D
P	50	5	35		(carriage return)

- In the hexadecimal equivalent of the character set, each character is assigned **two hexadecimal digits** which can be easily converted to their equivalent 8-bit code.
- The last entry of the table does not print a character but is associated with the physical movement of the cursor in the terminal.

Assembler

Representation of Symbolic Program in Memory

- A **line of code** is stored in consecutive memory locations with **two characters in each location**.
- Two characters can be stored in each word since **a memory word has a capacity of 16 bits**.
- A **label symbol** is terminated with **a comma**.
- **Operation and address symbols** are terminated with **a space** and the **end of the line** is recognized by the **CR code**.

Example: The following line of code

PL3, LDA SUB I

is stored in seven consecutive memory locations.

Memory word	Symbol	Hexadecimal code	Binary representation
1	P L	50 4C	0101 0000 0100 1100
2	3 ,	33 2C	0011 0011 0010 1100
3	L D	4C 44	0100 1100 0100 0100
4	A	41 20	0100 0001 0010 0000
5	S U	53 55	0101 0011 0101 0101
6	B	42 20	0100 0010 0010 0000
7	I CR	49 0D	0100 1001 0000 1101

Computer Representation of the Line of Code: PL3, LDA SUB I

Assembler

Representation of Symbolic Program in Memory

Example: The following line of code PL3, LDA SUB I

Character	Code	Character	Code	Character	Code	Memory word	Symbol	Hexadecimal code	Binary representation
A	41	Q	51	6	36	1	P L	50 4C	0101 0000 0100 1100
B	42	R	52	7	37	2	3 ,	33 2C	0011 0011 0010 1100
C	43	S	53	8	38	3	L D	4C 44	0100 1100 0100 0100
D	44	T	54	9	39	4	A	41 20	0100 0001 0010 0000
E	45	U	55	space	20	5	S U	53 55	0101 0011 0101 0101
F	46	V	56	(28	6	B	42 20	0100 0010 0010 0000
G	47	W	57)	29	7	I CR	49 0D	0100 1001 0000 1101
H	48	X	58	*	2A				
I	49	Y	59	+	2B				
J	4A	Z	5A	,	2C				
K	4B	0	30	-	2D				
L	4C	1	31	.	2E				
M	4D	2	32	/	2F				
N	4E	3	33	=	3D				
O	4F	4	34	CR	0D				
P	50	5	35						

- If the line of code has a comment, the assembler recognizes it by the code for a slash (2F).
- The assembler neglects all characters in the comment field and keeps checking for a CR code.

Assembler

Representation of Symbolic Program in Memory

Exercise: A line of code in an assembly language program is as follows:

DEC -35

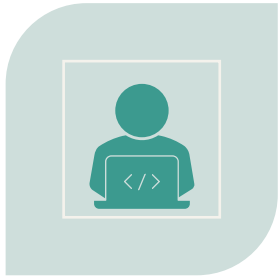
Show that four memory words are required to store the line of code and give their binary content.

Solution: DEC -35

Memory word	Characters	Hex	Binary
1	D E	44 45	0100 0100 0100 0101
2	C space	43 20	0100 0011 0010 0000
3	- 3	2D 33	0010 1101 0011 0011
4	5 CR	35 0D	0011 0101 0000 1101

Character	Code	Character	Code	Character	Code
A	41	Q	51	6	36
B	42	R	52	7	37
C	43	S	53	8	38
D	44	T	54	9	39
E	45	U	55	space	20
F	46	V	56	(28
G	47	W	57)	29
H	48	X	58	*	2A
I	49	Y	59	+	2B
J	4A	Z	5A	,	2C
K	4B	0	30	-	2D
L	4C	1	31	.	2E
M	4D	2	32	/	2F
N	4E	3	33	=	3D
O	4F	4	34	CR	0D (carriage return)
P	50	5	35		

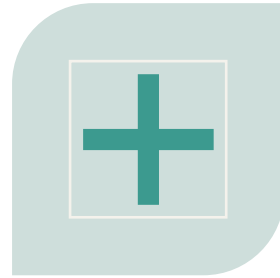
Programming Arithmetic and Logic Operations



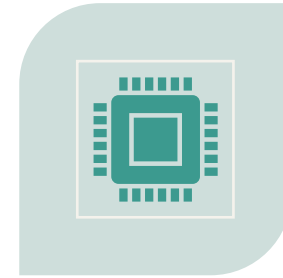
Some computers perform a given operation with one machine instruction; others may require **a large number of machine instructions** to perform the same operation.



Some computers have machine instructions with **four basic arithmetic instructions** to add, subtract, multiply, and divide.



Others have **only one arithmetic instruction** such as ADD.



Operations that are implemented in a computer with **one machine instruction** are said to be implemented by **hardware**.



Operations implemented by **a set of instructions** that constitute a program are said to be implemented by **software**.

Programming Arithmetic and Logic Operations

Programming Arithmetic Logic Operations

- Some computers provide **an extensive set of hardware instructions** designed to speed up common tasks.
- Others contain **a smaller set of hardware instructions** and depend more heavily on the software implementation of many operations.
- **Hardware implementation** is more costly because of the additional circuits needed to implement the operation.
- **Software implementation** results in long programs both in number of instructions and in execution time.

Programming Arithmetic and Logic Operations

Programming Arithmetic Logic Operations

Arithmetic Logic Unit (ALU)

- The Arithmetic and Logic Unit is the **functional part** of a digital computer that carries out arithmetic and logical operations on machine words that represent operands. It is usually a part of the central processing unit.
- Modern CPUs contain very **powerful and complex ALUs**. In addition to ALU modern CPUs also contains control unit. Most of the operations of the CPU are performed by **one or more ALUs** that load data from input registers.
- A **register** is a small amount of storage available as part of a CPU. The **control unit** tells the ALU what operation to perform on that data and the ALU stores the result in an output register.
- The control unit moves data between these registers, the ALU and memory.

Programming Arithmetic and Logic Operations

Programming Arithmetic Logic Operations

- The Verilog code for the ALU module includes an input for **two 8-bit binary numbers** (x and y), an input for the **operation selection** (sel), and an **output for the result** (z).
- The Verilog code for an ALU defines a module with **inputs, outputs, and a case statement** that implements the various arithmetic and logical operations.
- The inputs are two 8-bit binary numbers (x and y) and a 3-bit selector (sel) that specifies which operation to perform.
- The output is a **16-bit binary number (z)** that holds the result of the operation. The case statement evaluates the value of sel and performs the corresponding operation on x and y.

Programming Arithmetic and Logic Operations

Programming Arithmetic Logic Operations

Verilog code for Arithmetic Logic Unit

```
module alu(x, y, sel, z);
    input [7:0]x,y;
    output reg [15:0]z;
    input [2:0]sel;
    parameter ADD=3'b000;
    parameter SUB=3'b001;
    parameter MUL=3'b010;
    parameter DIV=3'b011;
    parameter AND=3'b100;
    parameter OR=3'b101;
    parameter NOT1 =3'b110;
    parameter NOT2 =3'b111;
    always@(*)
    case(sel)
        ADD: z=x+y;
        SUB: z=x-y;
        MUL: z=x*y;
        DIV: z=x/y;
        AND: z=x&&y;
        OR: z=x||y;
        NOT1: z=!x;
        NOT2: z=!y;
    endcase
endmodule
```

Testbench for Arithmetic and Logical Unit


```
module alu_tb();
    reg [7:0]x,y;
    reg [2:0]sel;
    wire[15:0]z;
    alu dut(x,y,sel,z);
    initial
    begin
        x=8'h133;
        y=8'h194;
        sel=3'b000;
        #10 sel=3'b001;
        #10 sel=3'b010;
        #10 sel=3'b011;
        #10 sel=3'b100;
        #10 sel=3'b101;
        #10 sel=3'b110;
        #10 sel=3'b111;
    end
    initial
    #200 $finish;
endmodule
```

Programming Arithmetic and Logic Operations

Programming Arithmetic Logic Operations

Lab Project Title: Designing an 8-bit Arithmetic Logic Unit Using Quartus II

Content	Objectives:
	Equipment/Software Requirements:
	Theory: ALU Functions
	Verilog Code for ALU:
	ALU's Simulation Results:
	ALU's RTL Design:

A vibrant image of the Aurora Borealis (Northern Lights) in shades of green and blue, set against a dark, starry night sky. The aurora's light bands are dynamic and flowing, creating a sense of movement and natural beauty.

Next Lecture:

Chapter (8): Central Processing Unit