Documentation

Design of Basic Computer and Assembler

By:

Nalin Semwal (2019UCO1505), Abhishek Jha (2019UCO1514)

Kartik Goyal (2019UCO1516)

Semester Project for Computer Architecture and Organisation

Semester 3, Computer Engineering 2019-2023

Netaji Subhash University of Technology, Dwarka

CONTENTS

Click on respective page number to jump to the corresponding section.

Getting Started

This is the tutorial section for assembler which is based on a basic computer *(see next section)*. The main webpage contains 2 sections: Assembly code editor and Machine code output.

Write your assembly code *(see Instruction set)* or check *sample programs* in the code editor. Click "translate" button to get the translated machine code *(see Assembler table)*.

Each instruction or pseudo-instruction must end with a semicolon ";" else it creates unexpected behaviour from the assembler like ignoring the instruction or interfering with the instructions following the instruction missing a semicolon.

This assembler is facilitated with **descriptive error messages** which are printed in Machine code area itself.

Following are some code snippets to demonstrate the working of the assembler.

This is a successful translation of the input assembly code:

```
1 LOAD A, 4H;
2 LOAD B, 5H;
3 ADD A, B;
```

```
0100 0001 0000 0100
0100 0010 0000 0101
0000 0000 0001 0110
```

This is an unsuccessful translation of the input assembly code. Error message is obtained in machine code area:

```
1 LOAD A, TEMP;
2 LOAD B, 5H;
3 ADD A, B;
```

```
Error at line 1: Unrecognized variable TEMP
```

Public GitHub repository can be found on the link:
https://github.com/kartik-3513/final_cao_assembler

This project can be found online on the link:
https://coa-project-051416.netlify.app/

or scan the following QR-Code:

Design of basic computer

Registers

The basic computer contains the following set of registers:

- Special purpose: 8-bit Address Register (AR), 16-bit Instruction Register (IR), 8-bit Program Counter (PC), 16-bit Data Register (DR), 8-bit Stack Pointer (SP), 8-bit Status Word Register (SWR)
- General Purpose:
    - It has 3 general purpose 16-bit registers namely: A, B, C
    - These are denoted by the bits 01 for A, 10 for B and 11 for C

Flags

Carry	Overflow	Sign	Zero	Input	Output	Interrupt	
CF	**OF**	**SF**	**ZF**	**INP**	**OUT**	**IEN**	**X**

Status Word Register

Memory

The memory of size 265x16. There are 256 memory locations of 16 bits each which amount to 512 bytes of memory. Thus 8-bit address is required to access whole memory.

Stack pointer grows from higher to lower memory location. Word pointed by the stack pointer is used to save the value of Program Counter (PC) and status of flags; higher order byte for storing status of flag as in Status Word Register (SWR) and lower order byte for Program Counter (PC).

Memory is word addressable since the size of data is equal to word length of CPU which is 16 bits.

Addressing modes

This computer supports 3 addressing modes which are denoted by their respective bits: Immediate (01), Direct (10), and Indirect (11).

1. **Immediate mode**

In this, the operand value is specified in the address field of the instruction. For example:

```
LOAD A 20H;     // Stores the hex-value 20 in register A
```

2. **Direct mode**

In this, the effective address of the operand value is specified in the address field of the instruction. [50H] or [A] is used to specify that we want to use the direct addressing mode in our instruction. For Example:

```
LOAD A [20H];   // Stores the value stored at memory location 20H in register A
```

3. Indirect mode

In this, the address of the memory word that contains the effective address of the operand is specified in the address field of the instruction. $50H or $A is used to specify that we want to use the indirect addressing mode in our instruction. For example:

```
LOAD A $20H;   // Location 20H in memory contains the address of memory of which data is to be
               // copied to register A
```

Bus System

A 16-bit common bus is used to connect the general and the special purpose registers. A common clock is used to operate the data transfers.

Data Register (DR) and general purpose A, B, C registers are connected to Arithmetic and Logical Unit (ALU) to facilitate the ALU operations.

Address Register is connected directly to memory to provide memory address.

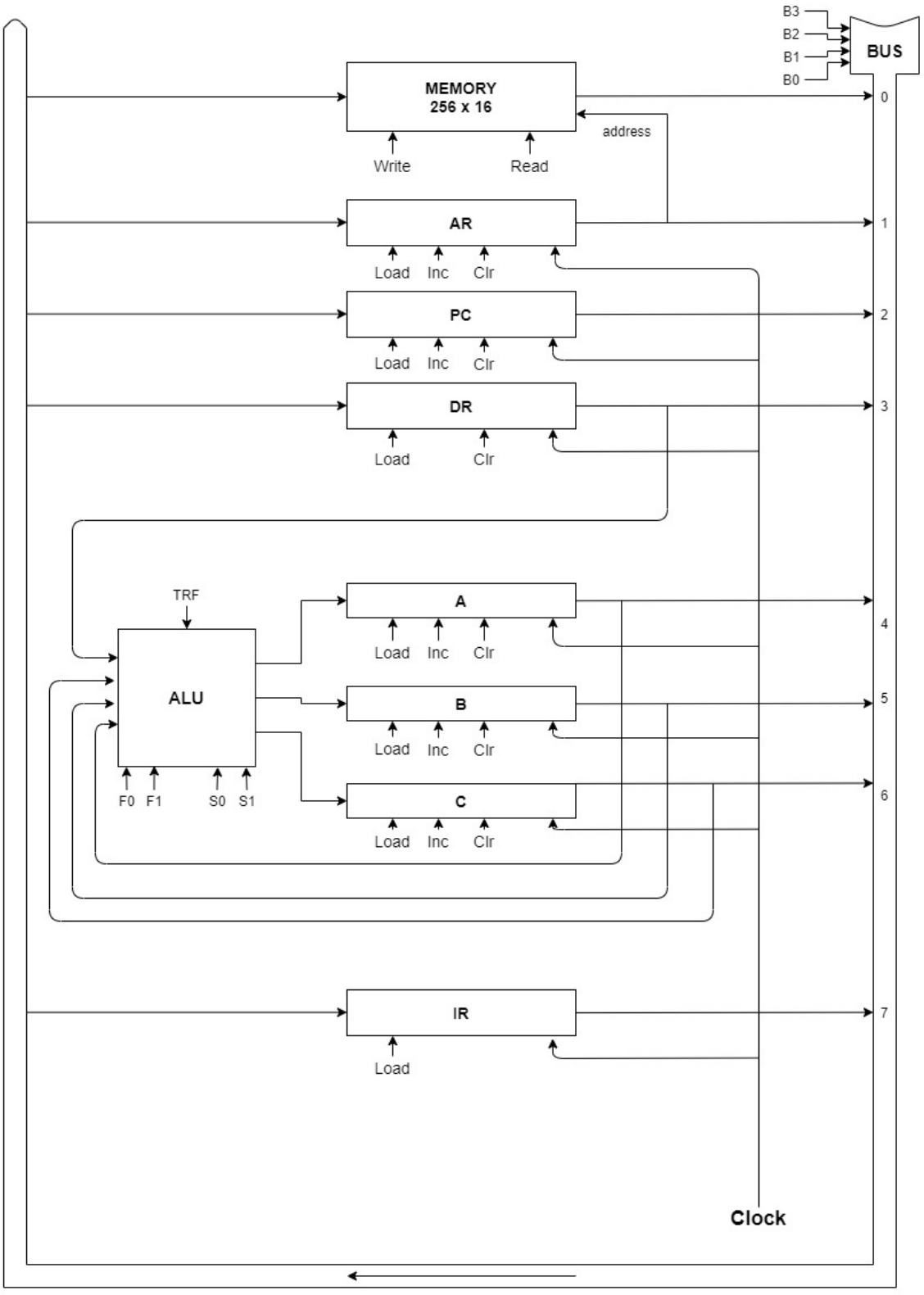
Signals B0-B3 are used to select the register whose data is to be transferred into the bus. When processor is in idle state, signal B3 is high and hence no data is being loaded into the bus.

Arithmetic and Logical Unit

Control signals in F0 and F1 are used to decide the first (destination) register and signals S0 and S1 are used to decide second (source) register which is taking part in REG-REG type Non-Memory Instructions.

Control signal TRF is used while transferring data from Data Register (DR) to any general purpose register. It takes precedence over S0-S1, hence when TRF is high, S0-S1 are ignored. F0-F1, as before, are used to decide the destination register in data transfer.

Bus System Diagram



16-bit Common Bus

Instruction Set

LOAD *REG, Operand*

It is used to load the data from memory location to a general-purpose register. (A, B, C). Operand can be immediate value, direct address, indirect address.

STORE *REG, Operand*

It is used to store the data from register to specified memory location. Operand can be immediate value, direct value, indirect value or value in a register.

JMPZ *label*

It is used to jump to the label if the Zero Flag (ZF) is set.

JMPN *label*

It is used to jump to the label if the Sign Flag (SF) is set.

JMPU *label*

It is used to jump to the label unconditionally.

CALL *label*

It is used to call the subroutine with a label. It pushes the PC and registers the value to the stack before jumping to the subroutine.

ADD *REG1, REG2*

It adds values in REG1 and REG2 and stores it in REG1.

MOV *REG1, REG2*

It is used to copy the data stored in REG2 to REG1.

AND *REG1, REG2*

It is used to perform bitwise AND between values of REG1 and REG2 and store it back in REG1.

OR *REG1, REG2*

It is used to perform bitwise OR between values of REG1 and REG2 and store it back in REG1.

XOR *REG1, REG2*

It is used to perform bitwise XOR between values of REG1 and REG2 and store it back in REG1.

NOT *REG*

It is used to flip the bits of the value in REG.

INC *REG*

Increments the value in the REG.

CILR *REG*

It is used to perform a circular right shift of the value in the REG with the Carry Flag.

CILL *REG*

It is used to perform a circular left shift of the value in the REG with the Carry Flag.

SHR *REG*

It is used to perform shift right operation within the register value. (Logical and Arithmetic Shift)

SHL *REG*

It is used to perform shift left operation with the register value. (Logical and Arithmetic Shift)

RET

It is used to return to the address on the top of the stack.

CINT

It is used to check the interrupt flag.

CMIF

It is used to complement the interrupt flag.

INPT

It transfers the content of input buffer into register A.

OUTP

It transfers the content of register A into output buffer.

CINF

It is used to check the input flag.

COUF

It is used to check the output flag.

PSEUDO-INSTRUCTIONS or ASSEMBLER DIRECTIVES

OG *value-in-hexadecimal*

Used to set the value of the location counter.

VAL *variable-name value-in-hexadecimal*

Used to declare a variable with the specified name with the specified value. The value can be a 16-bit value (4 hexadecimal digits.)

LABEL *label-name*

Used to declare a label that we use with branch instructions.

Instruction Design

Each instruction is 16 bits long. They are broadly classified into 2 categories:

- Memory Reference Instructions
- Non-Memory Reference Instructions.

Based on first 2 bits, IR 14-15, it takes two formats:

1. Memory Reference Instructions (When IR 14-15 is not equal to 00)

2-bit addressing mode	4-bit opcode	2-bit register address	8-bit address/immediate value

2. Non-Memory Reference Instructions (When IR 14-15 is equal to 00)

12-bit opcode	2-bit register address	2-bit register address

Instruction Cycle and Micro-operations

Fetch Routine:

Fetch 1: AR ← PC

Fetch 2: IR ← M[AR], PC ← PC + 1

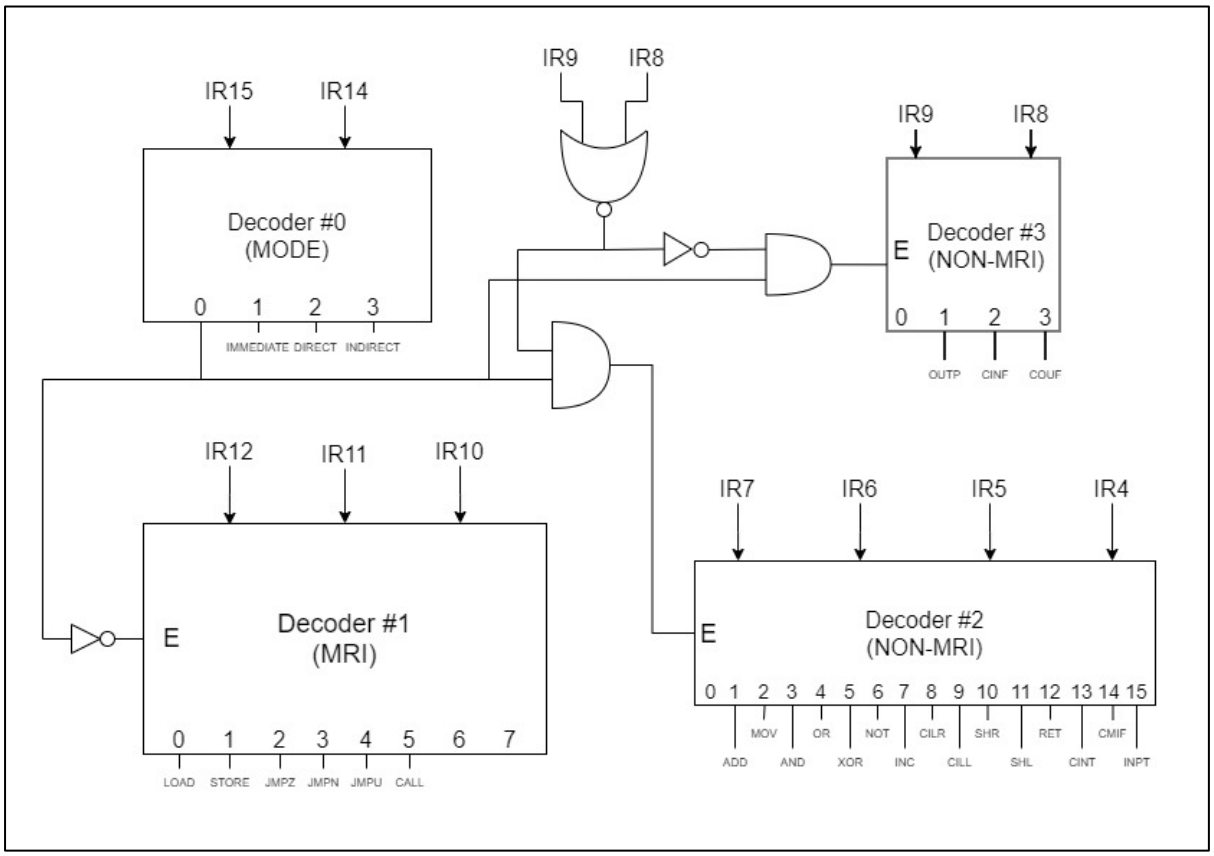
Decode Routine:

When decoding, addressing mode is decided using IR15 and IR14 bits.

IR 15	IR 14	Mode
0	0	Non- Memory Reference Instruction
0	1	Immediate Addressing Mode
1	0	Direct Addressing Mode
1	1	Indirect Addressing Mode

The IR(15-14) bits are decoded by the decoder#0 and sent to subsequent decoders.

Decoding Circuit Diagram



Non-MRI decoders (Decoder#2 and Decoder#3) are enabled if the instruction is Non-MR instruction. MRI decoder (Decoder#1) is enabled if the instruction is an MR instruction.

IR 8-9 bits decide the enable signal for Decoder#2 and Decoder#3 for Non-MR instruction. This approach of using two registers of size 2x4 and 4x16 instead of one decoder of size 6x64 was taken to save the amount of circuitry and unused outputs of the decoder.

These corresponding decoders then decode the instruction based on the opcode under these two types of instructions. *(see Assembler Table)*

Further operations depend on the addressing mode selected for MRI instructions.

Decode 1: Decode IR

In immediate addressing mode (01)	**In Direct Addressing Mode (10)**	**In Indirect Addressing Mode (11)**
Decode 2: DR ← IR(7-0)	Decode 2: AR ← IR(7-0)	Decode 2: AR ← IR(7-0)
	Decode 3: DR ← M[AR]	Decode 3: AR ← M[AR]
		Decode 4: DR ← M[AR]

Then the Execute phase follows based on the following micro-operations.

Micro-operation Table

INSTRUCTION	FUNCTION	REGISTER TRANSERS
LOAD	load1:	R ← DR, SC ← 0
STORE	store1:	AR ← DR(0-7)
	store2:	M[AR] ← R, SC ← 0
JMPZ	jmpz1:	If (ZF=1) then (PC ← DR(0-7)), SC ← 0
JMPN	jmpn1:	If (SF=1) then (PC ← DR(0-7)), SC ← 0
JMPU	jmpu1:	PC ← DR(0-7), SC ← 0
CALL	call1:	SP ← SP - 1
	call2:	AR ← SP
	call3:	M[AR](7-0) ← PC
	call4:	M[AR](15-8) ← SWR
	call5:	PC ← DR(0-7), SC ← 0
ADD	add1:	R1 ← R1 + R2, SC ← 0
MOV	mov1:	R1 ← R1, SC ← 0
AND	and1:	R1 ← R1 ^ R2, SC ← 0
OR	or1:	R1 ← R1 || R2, SC ← 0
XOR	xor1:	R1 ← R1 ⊕ R2, SC ← 0
NOT	not1:	R ← R', SC ← 0
INC	inc1:	R ← R + 1, SC ← 0
CILR	cilr1:	R ← shr R, R(15) ← C, C ← R(0), SC ← 0
CILL	cill1:	R ← shl R, R(0) ← C, C ← R(15), SC ← 0
SHR	shr1:	R ← shr R, SC ← 0
SHL	shl1:	R ← shl R, SC ← 0

RET	ret1:	AR ← SP
	ret2:	PC ← M[AR](7-0)
	ret3:	SWR ← M[AR](15-8)
	ret4:	SP ← SP + 1, SC ← 0
CINT	cint1:	If (IEN=1) then PC ← PC + 1, SC ← 0
CMIF	cmif1:	IEN ← IEN', SC ← 0
INPT	inpt1:	A ← Input-Buffer, SC ← 0
OUTP	outp1:	Output-Buffer ← A, SC ← 0
CINF	cinf1:	If (INP=1) then PC ← PC + 1, SC ← 0
COUF	couf1:	If (OUT=1) then PC ← PC + 1, SC ← 0

Assembler Table

Instructions can be broadly classified into 2 categories:

- Memory Reference Instructions
- Non-Memory Reference Instructions.

Memory reference instructions:

- Reg-Mem type instructions involve a register and memory access. These are used to transfer data to and from memory.
- Null-Mem type instructions involve only memory access and are used as control transfer instructions.

Non-Memory reference instructions:

- Reg-Reg type instruction which involves two registers, are used to perform arithmetic and logical operations on the data.
- Null-Reg type instructions involves only one register are used to perform bit-wise shift operations, NOT and increment value operation.
- Null-Null type instructions involves no register or memory reference. They include flag-bit manipulation as well as RETURN instruction as one control transfer instruction.

MEMORY REFERENCE INSTRUCTIONS

INSTRUCTION	2-bit addressing mode	6-bit opcode	2-bit wide register address	8-bit wide memory address
REG-MEM type instructions involving a register and a memory reference. AA denotes addressing mode bits.				
LOAD	AA	0000	2-bit register address	8-bit memory address
STORE	AA	0001	2-bit register address	8-bit memory address
NULL-MEM type instructions involving only a memory reference. 00 bits is used for unused register address				
JMPZ	AA	0010	00	8-bit memory address
JMPN	AA	0011	00	8-bit memory address
JMPU	AA	0100	00	8-bit memory address
CALL	AA	0101	00	8-bit memory address

NON-MEMORY REFERENCE INSTRUCTIONS

INSTRUCTION	12-bit wide op-code for Non-MR instructions	2-bit wide register address	2-bit wide register address
REG-REG type instructions involving two register references			
ADD	0000 0000 0001	2-bit register address	2-bit register address
MOV	0000 0000 0010	2-bit register address	2-bit register address
AND	0000 0000 0011	2-bit register address	2-bit register address
OR	0000 0000 0100	2-bit register address	2-bit register address
XOR	0000 0000 0101	2-bit register address	2-bit register address
NULL-REG type instructions involving only one register reference. 00 is used in place of unused register reference			
NOT	0000 0000 0110	00	2-bit register address
INC	0000 0000 0111	00	2-bit register address
CILR	0000 0000 1000	00	2-bit register address
CILL	0000 0000 1001	00	2-bit register address
SHR	0000 0000 1010	00	2-bit register address
SHL	0000 0000 1011	00	2-bit register address
NULL-NULL type instructions involving no register reference. 00 is used in place of both unused register references.			
RET	0000 0000 1100	00	00
CINT	0000 0000 1101	00	00
CMIF	0000 0000 1110	00	00
INPT	0000 0000 1111	00	00
OUTP	0000 0001 0000	00	00
CINF	0000 0010 0000	00	00
COUF	0000 0011 0000	00	00

Sample Programs

Three sample programs have been included in the assembler:

1. Fibonacci Numbers
2. Comparing two numbers
3. Shifting nibbles of a 16-bit register

Sample program to generate Fibonacci Numbers

```
LOAD A, [FIRST];
LOAD B, [SECOND];
LABEL LOP;
STORE A, $POS;
ADD A, B;
LOAD C, [POS];
INC C;
STORE C, POS;
LOAD C, [COUNT];
INC C;
STORE C, COUNT;
JMPZ BRK;
JMPU LOP;
LABEL BRK;
VAL FIRST 0H;
VAL SECOND 1H;
VAL COUNT FFFBH;
VAL POS 80H;
```

Sample program to check if two numbers are equal, A>B or A<B

```
LOAD A, 3H;
LOAD B, 4H;
STORE A, [BACKUPA];
STORE B, [BACKUPB];
NOT B;
INC B;
ADD A, B;
CALL EQL;
CALL GTR;
CALL LSR;
VAL BACKUPA 80H;
VAL BACKUPB 81H;
VAL STRSLT 82H;
```

```
OG 10H;
LABEL EQL;
JMPZ WRTEQL;
RET;
LABEL WRTEQL;
STORE A, [STRSLT];
RET;

OG 20H;
LABEL GTR;
JMPN NGTR;
LOAD A, [BACKUPA];
STORE A, [STRSLT];
LABEL NGTR;
RET;

OG 30H;
LABEL LSR;
JMPN WRTLSR;
RET;
LABEL WRTLSR;
LOAD B, [BACKUPB];
STORE B, [STRSLT];
RET;
```

Sample program to shift every nibble to the right

```
LOAD A, 77H;
CALL SR4;
STORE A, 80H;

OG 30H;
LABEL SR4;
LOAD C, FFFCH;
SHR A;
INC C;
JMPN SR4;
RET;
```

About

This project was created using HTML, CSS and vanilla JavaScript web technologies. We used CodeMirror plugin for assembly code editor and personalised syntax highlighting. Visit https://codemirror.net/ for more information.

Our team

Nalin Semwal	Abhishek Jha	Kartik Goyal
2019UCO1505	*2019UCO1514*	*2019UCO1516*
GitHub link: https://github.com/sc1lz	GitHub link: https://github.com/Abhishek-7139	GitHub link: https://github.com/kartik-3513