CPU DESIGN

Computer Architecture and Organization

CECSC07

Tanvee Balhara	**2019UCO1705**
Divyansh Tyagi	**2019UCO1721**
Yukti Priya	**2019UCO1722**
Shivam Tyagi	**2019UCO1725**

Contents

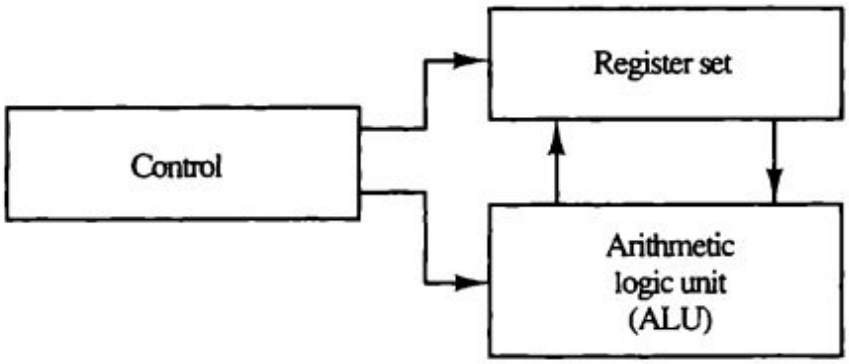
I) Machine Capabilities

The CPU we designed is for a general purpose computer. A general purpose computer is a computer that is designed to be able to carry out many different tasks. Desktop computers and laptops are examples of general purpose computers.

Features

The CPU is made up of three major parts as shown in fig. 1-1.

Fig. 1-1 Major Components of CPU



Principal components of the CPU include:

- **Arithmetic Logic Unit (ALU):** It performs the various arithmetic and logic operations described in the upcoming sections. The CPU only supports integer arithmetic.
- **Register Set:** This includes the processor registers that supply operands to the ALU and store the results of ALU operations, as well as other necessary registers like base and index registers, and flag registers.

- **Control Unit:** Operates the CPU bus system and directs the information flow through the registers and ALU by selecting the various components in the system.

General Information

Our CPU uses *8 bit data and address buses* to fetch operands and instructions from the memory. Most of our instructions are one-address instructions.
It consists of an ALU that operates on *8 bit integers*, and ten registers including four general purpose registers. It also has a status register for various functional purposes like program control.

The memory size of the system is *256 x 8 bits* i.e. the memory has 256 locations with a word size of 8 bits. Memory can be accessed by providing the address of the required word in the Address Register.

It has a common bus system in which the outputs of the registers and the memory are connected to the common bus.

Our CPU has a wide set of instructions which include:

- Data transfer instructions
- Data Manipulation instructions
    - Arithmetic instructions
    - Logical instructions
    - Shift operations
- Program control instructions

Our system can also receive input from and send output to peripheral devices.

Upcoming Sections

The upcoming sections will talk about the system in detail and finally will end on the Assembler for the designed system and examples codes for some basic programs.

II) Datapath and General Architecture

Our CPU consists of a single **accumulator(AC)** , 10 **registers** , and **main memory.**

Accumulator

We have a single accumulator which performs the following functions:

- Stores the results of arithmetic operations performed with the the data register (DR)
- Facilitates I/O to memory and memory to I/O transfers
- Stores results of arithmetic operations to memory/registers
- Loads data from memory/registers

Refer to table 3-2 for Accumulator instructions.

Registers

We are utilizing 10 registers which are as follows:

- **Program Counter (PC):** It holds the memory address of the next instruction which is to be executed.
- **Memory address register (MAR):** It contains the location/address of the operand which might be addressed by any of the suitable addressing modes in table 3.4.
- **Data Register:** It holds the values of an operand which is
    - Operated along with the value stored in the Accumulator
    - To be stored in the Accumulator
- **Base Register:** It is used for program relocation, displacement occurs w.r.t the value stored in it.
- **Index Register:** It is similar to the base register, however unlike the base register it stores the value by which the current address location is displaced.
- **Instruction Register:** It stores the current instruction , i.e. - the entire instruction in case of type 2 and type 3 and the higher 8 bits for type 1 instructions.
- **General registers:** We have 4 general registers named R1, R2, R3, R4 . These can be used to store frequently used values etc.

- **Flag Register:** It holds the values of several status bits like Sign bit (S), Accumulator Zero (Z) , Overflow (U) , Carry (C).
- **Input register:** It contains the inputs received from peripheral devices such as a keyboard. It is connected to the Accumulator through the ALU.
- **Output register:** It contains the information which is to be outputted to some peripheral device like a monitor .

Three addressing modes, namely (a) register direct (b) register indirect (c) auto increment modes are used with these Registers. Apart from these, (d) base addressing mode is also used with the base register

Table 2-1 Size of registers

Register	Size (Bits)
Program Counter	8
Memory Address Register	8
Data Register	8
Base Register	8
Index Register	8
Instruction Register	8
R1	8
R2	8
R3	8
R4	8

Main Memory

Our CPU has a main memory of size 256 x 8 bits. Each memory reference instruction can refer to one of the 256 locations. However for arithmetic purposes, any integer can lie in the range [(1-2^7 , 2^7-1)], negative numbers are stored in 1's complement form.

The main memory has (a) read and (b) write control signals. It can place/load data specified by the address register on/from the data bus from/at the address specified by the address register(AR).

Common Bus system

Our CPU consists of ten registers, a memory unit and a control unit . Paths must be provided to transfer information from one register to another and between memory and registers. For this we use a common bus system.

The outputs of 8 registers and memory are connected to the common bus. The specific output which is selected at any time depends on the value of the selection lines S3, S2, S1, S0. The particular register whose load is enabled receives the value on the bus. The memory receives the contents of the bus when its write is enabled. The memory places its 8 bit output onto the bus when the read input is activated. Table 2-2 Show the mapping of the select lines with their corresponding register/memory

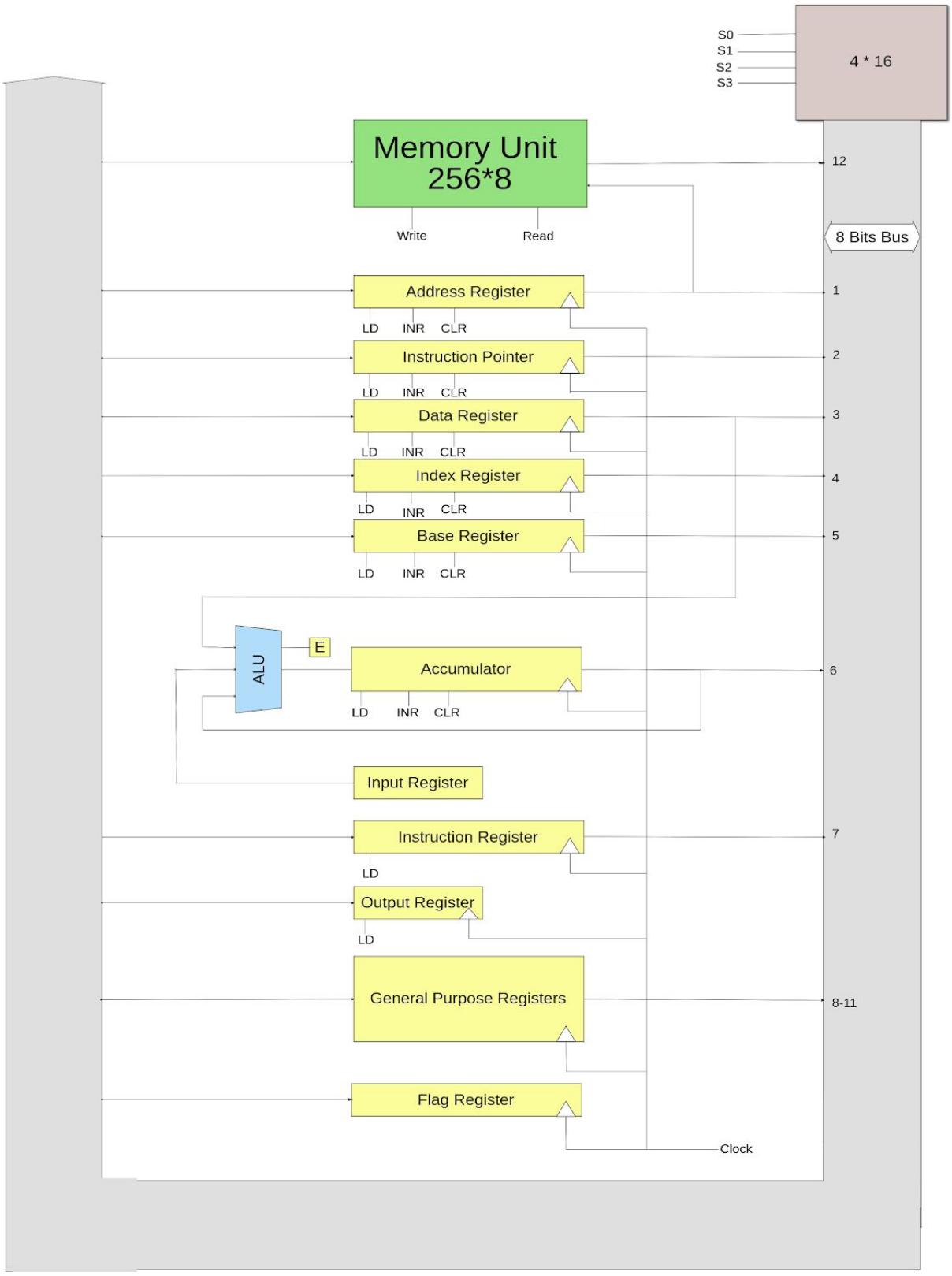
All the registers have 8 bits each. The INPR register is connected to provide information to the bus but OUTR can only receive information from the bus. This is because INPR receives a character from an input device which is then transferred to AC. OUTR receives a character from AC and delivers it to an output device. There is no transfer from OUTR to any of the other registers

The Bus has 8 lines i.e. bus width is 8. It receives information from 11 registers including AC and from the memory unit .

Table 2-2 Select lines for selecting certain memory/register

S3	S2	S1	S0	Register / Memory
0	0	0	0	
0	0	0	1	Address Register
0	0	1	0	Instruction Pointer
0	0	1	1	Data register
0	1	0	0	Index Register
0	1	0	1	Base Register
0	1	1	0	Accumulator
0	1	1	1	Instruction Register
1	0	0	0	R1
1	0	0	1	R2
1	0	1	0	R3
1	0	1	1	R4
1	1	0	0	Memory Unit
1	1	0	1	
1	1	1	0	
1	1	1	1	

Fig. 2-1 Common Bus



III) Instruction Set Architecture and Design

Our CPU has three types of instructions:

- **Type I:** Accumulator/E instructions
- **Type II:** Input - output instructions
- **Type III:** Memory/Register reference instructions

Instruction Size

The *memory/register reference instruction* uses 8 bits to specify the instruction to be performed and 8 bits to specify the address in memory or one of the four general purpose registers, making the total instruction length to be *16 bits* or two words.

Accumulator/E instructions and Input - output instructions don't require the address part, so they only use *8 bits* or one memory word.

Table 3-1 Instruction size

Type I	8 bits
Type II	8 bits
Type III	16 bits

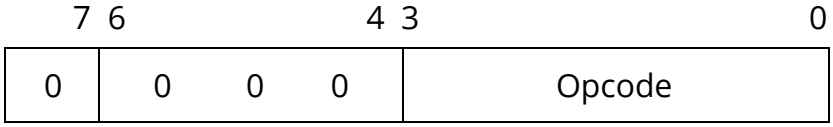
Instruction Format

- **Type I** (Accumulator/E instructions)
  - Bit 7 is always 0 signifying that the instruction is either type I or II
  - Bits 4-6 are 0 which along with a 0 at bit 7 signify type I instruction
  - Bit 0-3 give the opcode to signify one of the sixteen type I instructions

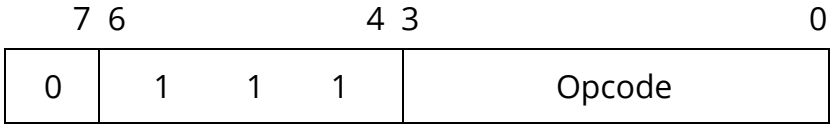
- **Type II** (Input - output instructions)
  - Bit 7 is always 0 signifying that the instruction is either type I or II
  - Bits 4-6 are 1 which along with a 0 at bit 7 signify type II instruction
  - Bit 0-3 give the opcode to signify one of the six type II instructions

- **Type III** (Memory/Register - reference instructions)
  - Bit 7 is always 1 signifying that the instruction is of type III
  - Bits 4-6 give one of the eight addressing mode
  - Bit 0-3 give the opcode to signify one of the thirteen type III instructions

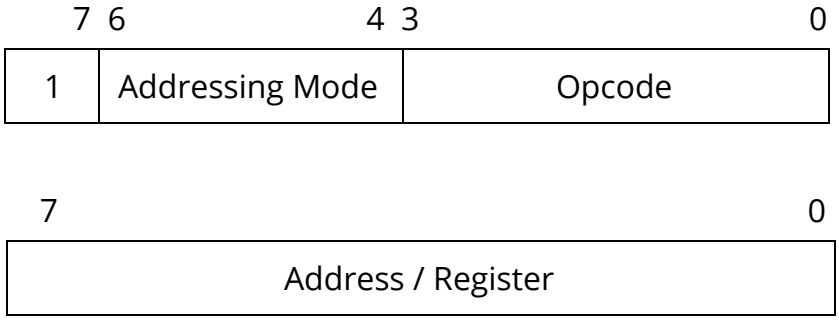
Fig. 3-1 Instruction formats



(a) Type I: Accumulator / E instructions



(b) Type II: Input - output instructions



(c) Type III: Memory/Register - reference instructions

Accumulator/E Instructions

They are recognised by the control when *bits D_4 through D_7 are 0*. They use *bits D_0 through D_3 to specify one of the sixteen instructions* mentioned in table 3-2.

Table 3-2 Accumulator/E Instructions

Instruction	Opcode	Description
CLA	0000	Clear AC
CLE	0001	Clear E
CIR	0010	Circulate right AC and E
CIL	0011	Circulate left AC and E
SHL	0100	Shift left AC
SHR	0101	Shift right AC
ASL	0110	Arithmetic shift left AC
ASR	0111	Arithmetic shift right AC
CMA	1000	Complement AC
CME	1001	Complement E
INC	1010	Increment AC
SPA	1011	Skip next instruction if AC is positive
SNA	1100	Skip next instruction if AC is negative
SZA	1101	Skip next instruction if AC is zero
SZE	1110	Skip next instruction if E is zero
HLT	1111	Halt computer

Input - Output Instructions

They are recognised by the control when *bit D_7 is 0 and bits D_4 to D_6 are 1*. They use *bits D_0 through D_3 to specify one of the six instructions* mentioned in table 3-3.

Table 3-3 Input - Output Instructions

Instruction	Opcode	Description
INP	0000	Input character to AC
OUT	0001	Output character from AC
SKI	0010	Skip on input flag
SKO	0011	Skip on output flag
ION	0100	Interrupt on
IOF	0101	Interrupt off

Register/Memory Reference Instructions

They are recognised by the control when *bit D_7 is 1*. *Bits D_4 to D_6 are used to specify one of the eight addressing modes* mentioned in table 3-4. They use *bits D_0 through D_3 to specify one of the thirteen instructions* mentioned in table 3-5.

The second word of the instruction specifies the *memory address* (0000 0000 through 1111 1111) *or one of the four general purpose registers* (0000 0001 through 0000 0100).

Table 3-4 Addressing Modes

Addressing Mode	Code (D_4-D_6)	Assembly Convention	Register Transfer
Immediate	000	LDA #NBR	AC<-NBR
Direct	001	LDA ADR	AC<-M[ADR]
Indirect	010	LDA @ADR	AC<-M[M[ADR]]
Relative PC	011	LDA $ADR	AC<-M[PC+ADR]
Base	100	LDA +ADR	AC<-M[ADR+Base]
Register	101	LDA R1	AC<-R1
Register Indirect	110	LDA (R1)	AC<-M[R1]
Auto Increment	111	LDA (R1)+	AC<-M[R1], R1<-R1+1

Table 3-5 Register/Memory Reference Instructions

Instruction	Code	Description
LDA	0000	Load memory/register word to AC
STA	0001	Store content of AC in memory/register
ADD	0010	Add memory/register word to AC
SUB	0011	Subtract memory/register word from AC
AND	0100	AND memory/register word to AC
ORA	0101	OR memory/register word to AC
NND	0110	NAND memory/register word to AC
NOR	0111	NOR memory/register word to AC

XNR	1000	XNOR memory/register word to AC
XOR	1001	XOR memory/register word to AC
CMP	1010	Compare memory/register word to AC
BUN	1011	Branch unconditionally
BSA	1100	Branch and save return address

Table 3-6 General Purpose Registers

Register	Code
R1	0000 0001
R2	0000 0010
R3	0000 0011
R4	0000 0100

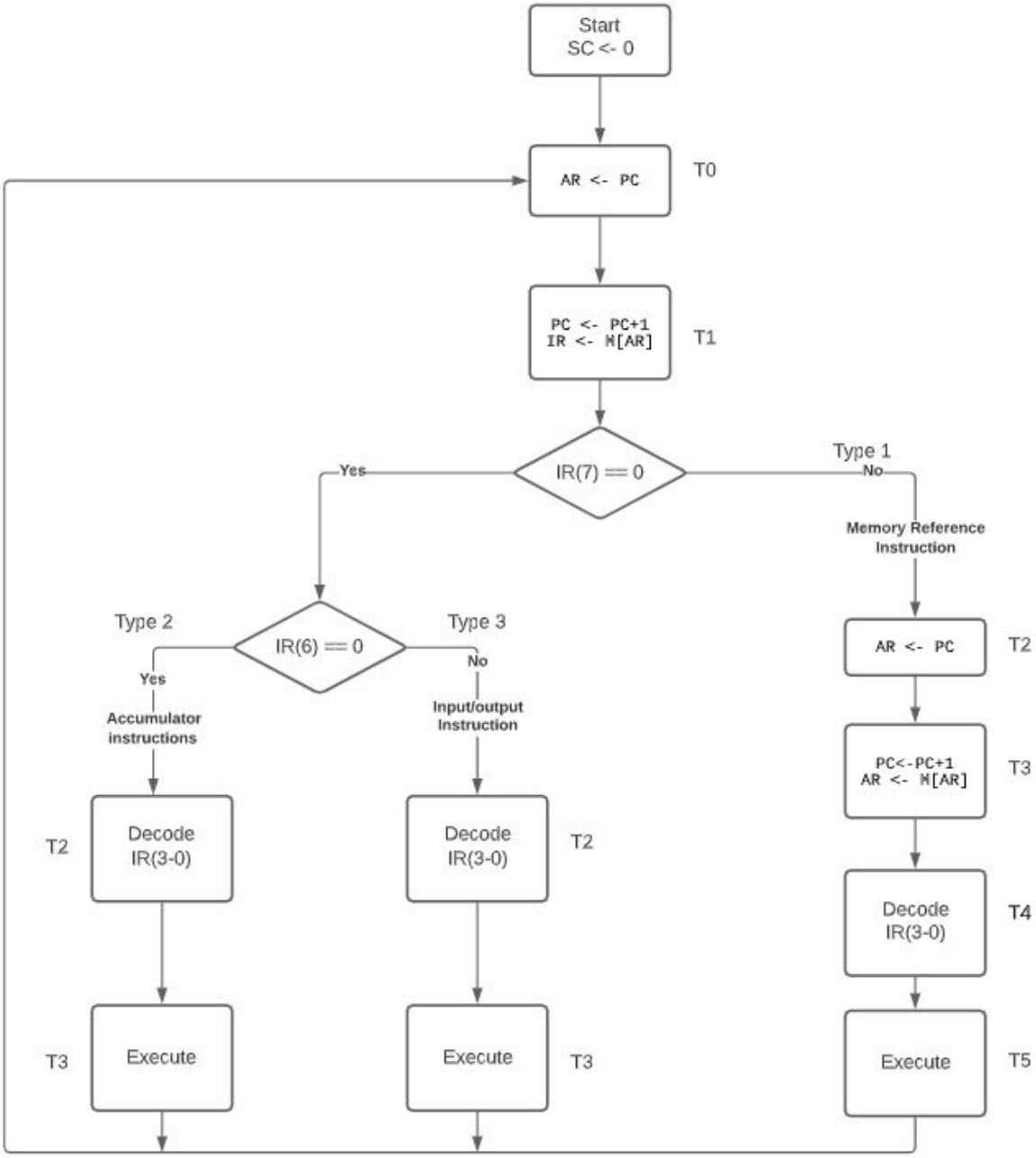
Examples:

Machine Code: STA (R3)+
Binary Code: 1111 0001
0000 0011

Machine Code: ADD @00101111
Binary Code: 1010 0010
0010 1111

Instruction Cycle

Fig. 3-2 Instruction Cycle



IV) Arithmetic Logical Unit

What is an ALU?

An arithmetic logic unit is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers. This is in contrast to a floating-point unit, which operates on floating point numbers.

An arithmetic logic unit(ALU) is a major component of the central processing unit of a computer system. It does all processes related to arithmetic and logic operations that need to be done on instruction words. In some microprocessor architectures, the ALU is divided into the arithmetic unit (AU) and the logic unit (LU).

Our ALU

The ALU we have designed handles various instructions by a combination of logical circuits and multiplexers. The instructions that the ALU operates on are given in table 4-1.

Table 4-1 ALU Operations

Operations	Binary Code
Addition	0010
Subtraction	0011
Logical AND	0100
Logical OR	0101
Logical NAND	0110
Logical NOR	0111
Exclusive NOR	1000
Exclusive OR	1001

Compare	1010
Clear Accumulator	0000
Circular Shift Right	0010
Circular Shift Left	0011
Logical Shift Left	0100
Logical Shift Right	0101
Arithmetic Shift Left	0110
Arithmetic Shift Right	0111
Complement Accumulator	1000
Increment Accumulator	1010

Basics of the ALU

The arithmetic microoperations listed can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.

Logic microoperations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register.

Shift Operations

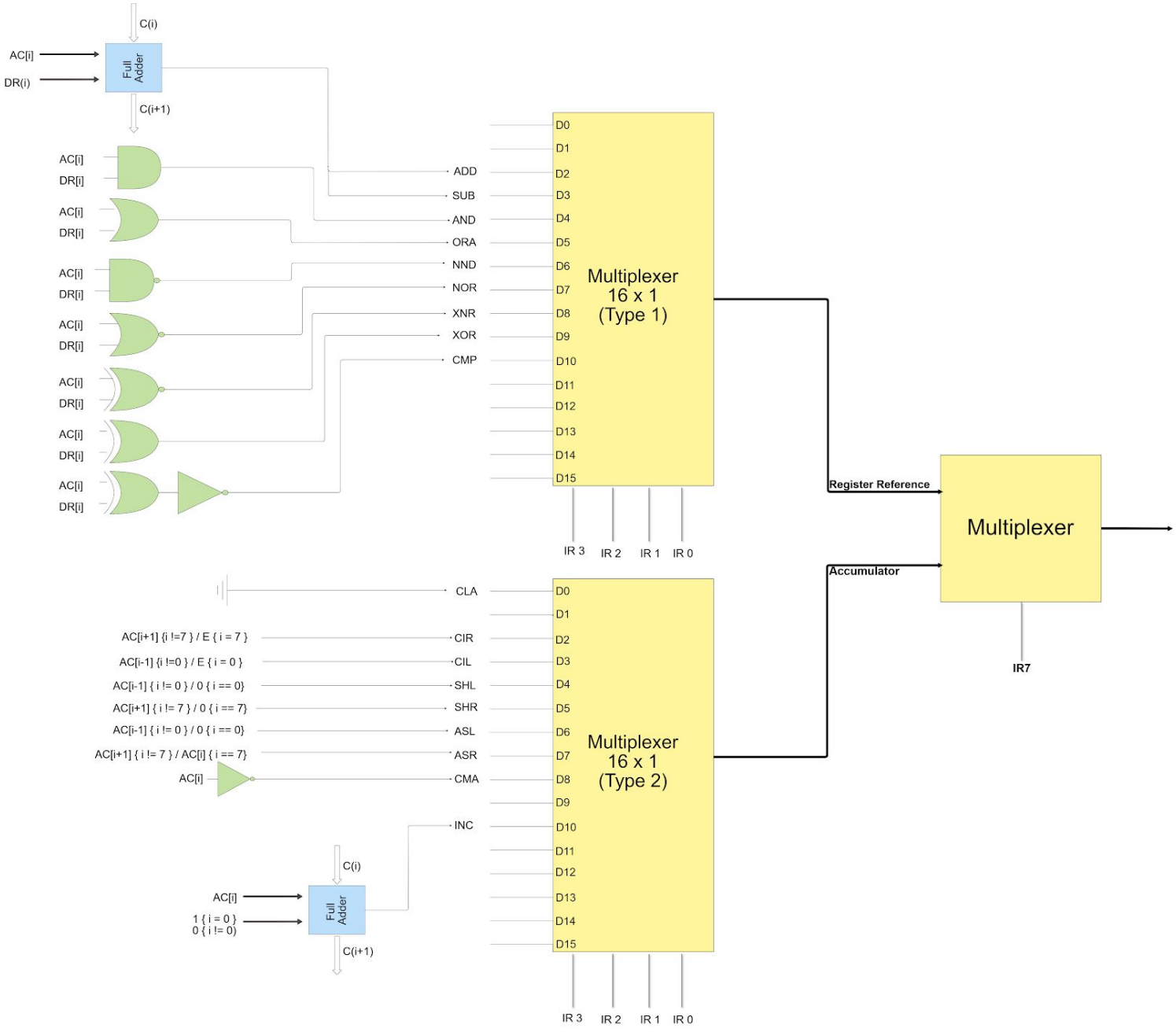
Shift microoperations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations. The

contents of a register can be shifted to the left or the right. At the same time that the bits are shifted, the first flip-flop receives its binary information from the serial input. During a shift-left operation the serial input transfers a bit into the rightmost position. During a shift-right operation the serial input transfers a bit into the leftmost position. The information transferred through the serial input determines the type of shift. There are three types of shifts: logical, circular, and arithmetic.

General Information

The circuits associated with the AC are shown below. The adder and logic circuit has three sets of inputs. One set of 8 inputs comes from the outputs of AC. Another set of 8 inputs comes from the data register DR. A third set of eight inputs comes from the input register INPR . The outputs of the adder and logic circuit provide the data inputs for the register. In addition, it is necessary to include logic gates for controlling the LD, INR, and CLR in the register and for controlling the operation of the adder and logic circuit. In order to design the logic associated with AC, it is necessary to go over the register transfer statements and extract all the statements that change the content of AC.

Fig. 4-1 ALU Design

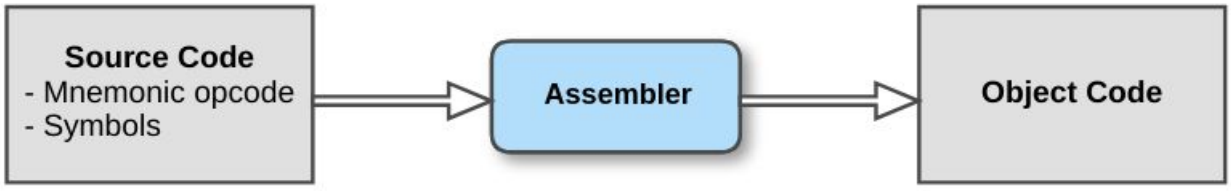


The ALU gets the input from the data bus/accumulator and sends the output to the accumulator

V) Assembler

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

Fig. 5-1 Assembler



General Instructions

- **Labels:** Labels can be specified at the starting of the instruction, and must be less than or equal to three characters in length. They can be alphanumeric but must begin with an alphabet. All labels must be preceded by " . ". E.g - .myLabel HEX 10
- **Comment:** Only comments succeeding an instruction are allowed. Full line comments are not allowed. All comments must begin with "/".
- **END Assembly Directive:** Assembly code must finish with END.
- **ORG Assembly Directive:** It is optional and should be followed by the address in decimal where the first instruction should be loaded to.
- **Operands in Register/Memory Reference instruction:** The operand / address must be in binary in case of memory reference instruction, and should be one of R1, R2, R3, R4 in register reference instructions.
- **Instruction Layout in Memory:** All instructions are stored consecutively in memory, starting from the address specified by the ORG directive or from 0 incase ORG is not present. All labels are stored after the last instruction.
- We do **NOT** handle macros or procedures in our program.

First Pass

During the first pass , our assembler generates an address symbol table that correlates all user-defined address symbols / labels with their binary equivalent value.
The assembler tracks the location of the instruction using a location counter.

Labels

These are user-defined address symbol which can refer to

A. *Memory addresses*
B. *Routines*

Syntax - .X DEC 10

- Labels can be specified at the starting of the instruction, each label must be preceded by a '.'
- The maximum character length for a label can be 3

Table 5-1 shows an example address symbol table.

Table 5-1 Example address symbol table

Address Symbol	Address (Decimal)
MIN	5
SUB	6

Code Snippet

```
LC = 0
addSymbol = dict()

lineNo = 0

with open("machineCode.txt", mode='r') as f:
    for line in f:
        lineNo += 1
        inst = line.split('/')[0].split()

        if line[0] == ".":
            label = inst[0][1:]
            # label repeated

            if label in addSymbol:
                callError(lineNo, "Labels cannot be repeated ")
            # label length > 3
            elif len(label) > 3:
                callError(lineNo, "Labels cannot exceed length 3")
            # Same as register names (R1, R2, R3, R4)
            x = re.fullmatch("R1|R2|R3|R4", label)

            if x:
                callError(lineNo, "Label can not be Register name")
            # Should be alphanumeric and should start with an alphabet
            x = re.fullmatch("[a-zA-Z][a-zA-Z0-9]*", label)

            if x == None:
                callError(lineNo, "Label should be alphanumeric and should
start with an alphabet")
            addSymbol.update({label:LC})

        else:
            if len(inst) == 0 :
                callError(lineNo,"Line can not be empty or can not start
with a comment, it must start with an instruction or a label")
            oprn = inst[0]
            if oprn == 'ORG':
                LC = int(inst[1])
                continue
```

```
            elif oprn == 'END':
                LC += 1
                break
            elif oprn in MRI:
                LC += 1
        LC += 1
```

Second Pass

Machine instructions will be translated into equivalent binary code during the second pass by means of table - lookup procedure.

Our assembler uses four tables namely:

1. MRI table
2. Non - MRI table for Accumulator / E instructions
3. Non - MRI table for Input - Output instructions
4. Address symbol table

Any symbol encountered in the program must either be a pseudo - instructions (ORG, END, DEC, HEX) or must be available as an entry in one of these tables.

MRI table contains the symbols for the thirteen memory/register reference instructions along with their 4 bit opcodes.
Non-MRI tables contain the symbols for sixteen accumulator/E instructions, and six input-output instructions respectively, along with their four bit opcodes.
Address symbol table is generated during the first pass of the assembler.

The basic processes going on in the second pass are mentioned below. Brief description is also provided in the flowchart in fig. 5-2.

- LC is initially set to 0, and lines of code are analyzed one at a time.
- Labels are neglected during the second pass and control goes directly to the instruction field.
- Match with ORG sets LC to the location specified and a match with END terminates the translation process.

- If the symbol encountered is not a pseudoinstruction, the assembler does a table lookup in MRI, NMRI_acc, NMRI_io in the given order.
    - If the instruction is found in the MRI table, binaryForMRI() is called which returns the equivalent binary code for the instruction. It uses findAddressingMode() to extract the addressing mode for memory/register reference. findAddressBits() computes the operand address on the basis of the addressing mode. binaryForMRI() in the end returns a 16 bit binary code.
    - If the instruction is found in the NMRI_acc table, binaryForNMRI_acc() is called. It looks up the 4 bit opcode for the instruction in the table, prepends “0000” signifying type I instruction, and returns the resultant 8 bit binary code.
    - If the instruction is found in the NMRI_io table, binaryForNMRI_io() is called. It looks up the 4 bit opcode for the instruction in the table, prepends “1000” signifying type II instruction, and returns the resultant 8 bit binary code.
- The binary instructions returned by the functions are stored along with the value of the Location Counter.
- The value of LC is incremented and the assembler continues to process the next line.

Example:

For the following code:

Machine Code:

```
ORG   10     /Line 1
LDA   (R2)+  /Line 2
END          /Line 3
```

1. Pseudo-instruction ORG is encountered in line 1. Assembler sets LC to 10 and moves to the next line.
2. In line 2, operation LDA is not a pseudo-instruction and is found in the MRI table. This results in a call to function binaryForMRI() which makes a call to findAddressingMode() (which returns “1111” for auto-increment mode) and findAddressBits() (which returns “00000010” for auto-increment mode with R2). The function returns two 8 bit binary codes, one for addressing

mode and opcode, and the other for the address field. In this case, the value returned is [“11110000”, “00000010”]. The main program keeps a list of instructions along with their address in memory. So, [“00001010”, “11110000”], [“00001011”, “00000010”] is appended to the list.

3. Pseudo-instruction END is encountered in line 3. The translation process is terminated and the list of instructions is stored in a text file.

The binary code generated is given in table 5-2.

Table 5-2 Binary Code generated for example

LOCATION	CONTENT
00001010	11110000
00001011	00000010

Error Handling

Our assembler handles a variety of errors as well as informs the user about its existence.

Errors handled in first pass

- **Repetition of label:** Having multiple labels with the same name is forbidden as it will create conflicts in the address symbol table,using repeated labels cause will raise error.

```
Error at line 2: Labels cannot be repeated
```

- **Label name exceeding allowed length:** The assembler allows label names to have a maximum length of 3 characters, exceeding this limit will raise error.

```
Error at line 1: Labels cannot exceed length 3
```

- **Label name matching register names:** The CPU contains 4 general purpose registers - R1, R2, R3, R4. To avoid confusion, labels must be named differently from the given register names.

```
Error at line 1: Label can not be Register name
```

- **Invalid Characters in label naming -** Label names must be alphanumeric and should start with an alphabet.

```
Error at line 1: Label should be alphanumeric and should start with an alphabet
```

Errors handled in second pass

- **Empty line or full line comment:** Since the assembler only supports comments following an instruction, an empty line or full line comments raise error.

```
Error at line 1: Line can not be empty or can not start with a comment, it must start with
 an instruction or a label
```

- **Incorrect operands with HEX and DEC pseudo-instructions:** HEX and DEC require hexadecimal and decimal values respectively. Any incorrect value raises an error.

```
Error at line 1: Value at label should be decimal
```

```
Error at line 1: Value at label should be hexadecimal
```

- **Operand for HEX and DEC pseudo-instructions out of bound:** Any values specified with HEX or DEC that exceeds the word-size(ie: <= (-2^7 + 1) or >=(2^7 + 1))will raise an error.

```
Error at line 1: Value out of bounds
```

- **Invalid number of operands:** Type 3 instructions can only contain one address. If more than one or zero operands are provided, the assembler raises an error. Similarly, type 1 and type 2 instructions, if followed by an operand will be seen as an error by the assembler.

```
Error at line 1: Invalid number of operands
```

- **Instruction not in ISA or instruction missing:** If any instruction code other than the one in the ISA is specified, or if no opcode is given, the assembler raises an error.

```
Error at line 1: Opcode not found
```

- **Invalid register name in Register Reference Instructions:** The operand in Register Reference instructions must be one of R1, R2, R3, R4. If anything other than these is specified as the operand, error is raised.

```
Error at line 1: Referenced Register does not exist
```

- **Invalid operand in Memory Reference Instructions:** The operand for these instructions must either be a valid binary number or a label. Anything else will raise an error.

```
Error at line 1: Address should be in binary or a valid label
```

- **Machine code not ending with END:** Every machine code must be terminated by the “END” Pseudoinstruction. Failure to do so will raise an error.

```
Error at line 1: END not found. Machine program should end with END
```

The machine code in machineCode.txt is translated to binary code if there are no errors, and stored in binaryCode.txt.

```
Program has been successfully assembled
```

ASSEMBLER SIMULATION IN PYTHON

https://github.com/tanvee09/Two-Pass-Assembler-Simulation-Python

main.py

```
import sys
import re
from tables import *
from helper import *

if len(sys.argv) < 2 :
    print("Give a file name")
    exit(1)

file_name = sys.argv[1]

# FIRST PASS

LC = 0
addSymbol = dict()

lineNo = 0

with open("../examples/" + file_name, mode='r') as f:
    for line in f:
```

```
        lineNo += 1
        inst = line.split('/')[0].split()
        if line[0] == ".":
            label = inst[0][1:]
            # label repeated
            if label in addSymbol:
                callError(lineNo, "Labels cannot be repeated ")
            # label length > 3
            elif len(label) > 3:
                callError(lineNo, "Labels cannot exceed length 3")
            # Same as register names (R1, R2, R3, R4)
            x = re.fullmatch("R1|R2|R3|R4", label)
            if x:
                callError(lineNo, "Label can not be Register name")
            # Should be alphanumeric and should start with an alphabet
            x = re.fullmatch("[a-zA-Z][a-zA-Z0-9]*", label)
            if x == None:
                callError(lineNo, "Label should be alphanumeric and should
start with an alphabet")
            addSymbol.update({label:LC})
        else:
            if len(inst) == 0 :
                callError(lineNo, "Line can not be empty or can not start
with a comment, it must start with an instruction or a label")
            oprn = inst[0]
            if oprn == 'ORG':
                LC = int(inst[1])
                continue
            elif oprn == 'END':
                LC += 1
                break
            elif oprn in MRI:
                LC += 1
        LC += 1



# SECOND PASS

LC = 0
binary_code = []

lineNo = 0
```

```
with open("../examples/" + file_name, mode='r') as f:
    for line in f:
        lineNo += 1
        inst = line.split('/')[0].split()

        if line[0] == ".":
            if inst[1] == 'DEC':
                if len(inst) != 3:
                    callError(lineNo, "Invalid number of operands")
                try:
                    int(inst[-1])
                except:
                    callError(lineNo, "Value at label should be decimal")
                if int(inst[-1]) > (2**7 - 1) or int(inst[-1]) < -2**7 + 1:
                    callError(lineNo, "Value out of bounds")
                binary_code.append([toBinary(addSymbol[inst[0][1:]], 10),
toBinary(inst[-1], 10)])
                continue
            elif inst[1] == 'HEX':
                if len(inst) != 3:
                    callError(lineNo, "Invalid number of operands")
                try:
                    int(inst[-1], 16)
                except:
                    callError(lineNo, "Value at label should be
hexadecimal")
                if int(inst[-1], 16) > (2**7 - 1) or int(inst[-1], 16) <
-2**7 + 1:
                    callError(lineNo, "Value out of bounds")
                binary_code.append([toBinary(addSymbol[inst[0][1:]], 10),
toBinary(inst[-1], 16)])
                continue
            else:
                inst = inst[1:]


        oprn = inst[0]
        if oprn == 'ORG':
            LC = int(inst[1])
            continue
        elif oprn == 'END':
            break
```

```
        elif oprn in MRI:
            if len(inst) != 2:
                callError(lineNo, "Invalid number of operands")
            binary_code += binaryForMRI(inst, LC, addSymbol, lineNo)
            LC += 1
        elif oprn in NMRI_io:
            if len(inst) > 1:
                callError(lineNo, "Invalid number of operands")
            binary_code.append(binaryForNMRI_io(inst, LC))
        elif oprn in NMRI_acc:
            if len(inst) > 1:
                callError(lineNo, "Invalid number of operands")
            binary_code.append(binaryForNMRI_acc(inst, LC))
        else:
            callError(lineNo, "Opcode not found")
        LC += 1
        if LC >= 2**8:
            callError(lineNo, "Program memory out of bounds")

    else:
        callError(lineNo, "END not found. Machine program should end with
END")


with open("../examples/binaryCode_" + file_name, mode='w') as f:
    f.write('LOCATION\t\tCONTENT\n\n')
    f.writelines([ins[0] + '\t\t' + ins[1] + '\n' for ins in binary_code])
```

helper.py

```
from tables import *
import re


def callError(lineNo: int, errMessage: str) :
    print('\033[91m' + 'Error at line {}: {}'.format(lineNo, errMessage) +
'\033[0m')
    exit(1)


def toBinary(n, b: int) -> str:
```

```
    binary = bin(int(str(n), b)).replace("0b", "")
    if binary[0] == '-':
        return '1' * (8 - len(binary) + 1) + ''.join(['1' if bit == '0' else
'0' for bit in binary[1:]])
    else:
        return '0' * (8 - len(binary)) + binary


def findAddressingMode(address: str) -> str :
    if address[0] == '#':
        return addressingModes['immediate']
    elif address[0] == '@':
        return addressingModes['indirect']
    elif address[0] == '$':
        return addressingModes['relativePC']
    elif address[0] == '+':
        return addressingModes['base']
    elif address[-1] == ')':
        return addressingModes['registerIndirect']
    elif address[-1] == '+':
        return addressingModes['autoInc']
    elif address in generalPurposeRegs:
        return addressingModes['register']
    else:
        return addressingModes['direct']


def findAddressBits(address: str, addMode: str, addSymbol: dict, lineNo:
int) -> str:
    if addMode == '1101': # Register
        if address not in generalPurposeRegs.keys():
            callError(lineNo, "Referenced Register does not exist")
        return generalPurposeRegs[address]

    elif addMode == '1110': # registerIndirect
        address = address[1:-1]
        if address not in generalPurposeRegs.keys():
            callError(lineNo, "Referenced Register does not exist")
        return generalPurposeRegs[address]

    elif addMode == '1111': # autoInc
        address = address[1:-2]
        if address not in generalPurposeRegs.keys():
```

```
            callError(lineNo, "Referenced Register does not exist")
        return generalPurposeRegs[address]

    elif addMode == '1000': # immediate
        if len(address) != 8 or re.fullmatch("[01]*", address) == None:
            callError(lineNo, "Address should be in binary or a valid
label")
        return toBinary(address[1:], 2)

    if addMode != '1001':# not direct
        address = address[1:]

    if len(address) == 8 and re.fullmatch("[01]*", address) != None:
        return toBinary(address, 2)
    elif address in addSymbol.keys():
        return toBinary(addSymbol[address], 10)
    else:
        callError(lineNo, "Address should be in binary or a valid label")


def binaryForMRI(inst: list, LC: int, addSymbol: dict, lineNo: int) -> str:
    addMode = findAddressingMode(inst[-1])
    opcode = MRI[inst[0]]
    address = findAddressBits(inst[-1], addMode, addSymbol, lineNo)
    return [[toBinary(int(LC), 10), addMode + opcode], [toBinary(int(LC +
1), 10), address]]


def binaryForNMRI_acc(inst: list, LC: int) -> str:
    addMode = "0000"
    opcode = NMRI_acc[inst[0]]
    return [toBinary(int(LC), 10), addMode + opcode]


def binaryForNMRI_io(inst: list, LC: int) -> str:
    addMode = "0111"
    opcode = NMRI_io[inst[0]]
    return [toBinary(int(LC), 10), addMode + opcode]
```

tables.py

```
MRI = {
    "LDA": "0000",
    "STA": "0001",
    "ADD": "0010",
    "SUB": "0011",
    "AND": "0100",
    "ORA": "0101",
    "NND": "0110",
    "NOR": "0111",
    "XNR": "1000",
    "XOR": "1001",
    "CMP": "1010",
    "BUN": "1011",
    "BSA": "1100"
}

NMRI_acc = {
    "CLA": "0000",
    "CLE": "0001",
    "CIR": "0010",
    "CIL": "0011",
    "SHL": "0100",
    "SHR": "0101",
    "ASL": "0110",
    "ASR": "0111",
    "CMA": "1000",
    "CME": "1001",
    "INC": "1010",
    "SPA": "1011",
    "SNA": "1100",
    "SZA": "1101",
    "SZE": "1110",
    "HLT": "1111"
}

NMRI_io = {
    "INP": "0000",
    "OUT": "0001",
    "SKI": "0010",
    "SKO": "0011",
    "ION": "0100",
```

```
    "IOF": "0101"
}

addressingModes = {
    "immediate":        "1000",
    "direct":           "1001",
    "indirect":         "1010",
    "relativePC":       "1011",
    "base":             "1100",
    "register":         "1101",
    "registerIndirect": "1110",
    "autoInc":          "1111"
}

generalPurposeRegs = {
    "R1": "00000001",
    "R2": "00000010",
    "R3": "00000011",
    "R4": "00000100"
}
```

EXAMPLE 1: Subtraction of two numbers using ADD

Fig. 5-2(a) machineCode.txt

```
        ORG     1
        LDA     SUB     /1, 2
        CMA             /3
        INC             /4
        ADD     MIN     /5, 6
        STA     DIF     /7, 8
        HLT             /9
.MIN    DEC     83      /10
.SUB    DEC     -23     /11
.DIF    HEX     0       /12
        END             /13
```

Fig. 5-2(b) binaryCode.txt

LOCATION	CONTENT
00000001	10010000
00000010	00001011
00000011	00001000
00000100	00001010
00000101	10010010
00000110	00001010
00000111	10010001
00001000	00001100
00001001	00001111
00001010	01010011
00001011	11101000
00001100	00000000

EXAMPLE 2: Multiplication of two numbers

Fig. 5-3(a) machineCode.txt

```
        ORG     100
.LOP    CLE             /Clear E
        LDA     Y       /Load Multipler
        CIR             /Transfer multiplier bit to E
        STA     Y       /Store shifter multiplier
        SZE
        BUN     ONE     /Bit is one; go to ONE
        BUN     ZRO     /Bit is zero; go to ZRO
.ONE    LDA     X       /Load Multiplicand
        ADD     P       /Add to partial product
        STA     P       /Store partial product
        CLE             /Clear E
.ZRO    LDA     X       /Load Multiplicand
        CIL             /Shift left
        STA     X       /Store shifted Multiplicand
        LDA     CTR     /Load CTR
        INC             /Increment Accumulator
        STA     CTR     /Store incremented value of CTR
        SZA
        BUN     LOP     /Counter not zero; repeat loop
        HLT             /Counter is zero; halt
.CTR    DEC     -8      /This location serves as counter
.X      HEX     0A      /Multiplicand stored here
.Y      HEX     03      /Multiplier stored here
.P      HEX     0       /Product formed here
        END
```

Fig. 5-3(b) binaryCode.txt

LOCATION	CONTENT
01100100	00000001
01100101	10010000
01100110	10000100
01100111	00000010
01101000	10010001
01101001	10000100
01101010	00001110
01101011	10011011
01101100	01101111
01101101	10011011
01101110	01110101
01101111	10010000
01110000	10000011
01110001	10010010
01110010	10000101
01110011	10010001
01110100	10000101
01110101	00000001
01110110	10010000
01110111	10000011
01111000	00000011
01111001	10010001
01111010	10000011
01111011	10010000
01111100	10000010
01111101	00001010
01111110	10010001
01111111	10000010
10000000	00001101
10000001	10011011
10000010	01100100
10000011	00001111
10000010	11110111
10000011	00001010
10000100	00000011
10000101	00000000