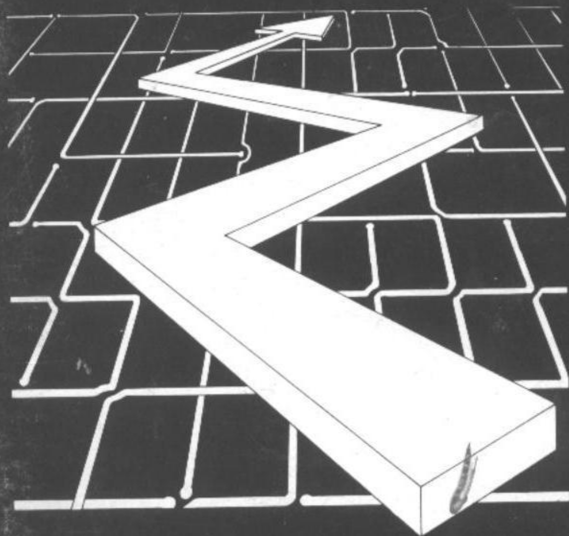
ENTERPRISE

PROGRAMS



MACHINE CODE FOR BEGINNERS

INSTRUCTIONS

MACHINE CODE FOR BEGINNERS

a

GUIDE TO THE PROGRAM BY:

Stephen Huckstepp B.Sc.(Hons)
Electronic Engineer

and

Les Graddon B.A. (Hons).
Head of Faculty of Mathematics and Computing
The Vyne Comprehensive School Basingstoke,
Hants.

© DREAM SOFTWARE LTD 1985

Micro-Simulation © Dream Software Ltd 1985.
This program is published by Entersoft Limited who hold exclusive right to its publication.

It may not be published, copied, transferred, transmitted, reproduced, distributed, lent, hired or modified in any form without the express permission of Entersoft Limited:

37 Bedford Square,
London WC1B 3HW

CONTENTS

Introduction

Section 1 The Central Processor and Memory. Registers, Memory Locations and Addresses

Section 2 Introducing the simulator program — first steps in machine code programming. The LOAD instruction. Mnemonics, Operands, Bits and Bytes, Assemblers, Op-Codes, Program Counter.

Section 3 More practice with the simulator. More about "LD". HALT, ADD, SUB, Running a program with a time delay, INC, DEC

Section 4 Logical instructions AND, OR, XOR

Section 5 Flags. Instructions CP, RRA, RLA, SRA SLA

Section 6 The stack. PUSH and POP. CALL and RET.

Section 7 Conditional Jumps. Indirect addressing

Section 8 OUT and DATA. Using Hexadecimal numbers. The ASSEMBLER display

Section 9 Control codes

Section 10 The P register

Section 11 Fast running. Load and Save

Section 12 Demonstration programs

1) Simple Counting
2) Demonstrates the display
3) Illustrates memory addressing
4) Shows use of stack
5) Doubles a decimal number

A note on the Z80 processor

Appendix A Decimal, Binary, and Hexadecimal numbers

Appendix B Summary of the simulator with a list of all available instructions and notes on their use.

Page

Section 1 2
Section 2 6
Section 3 15
Section 4 18
Section 5 20
Section 6 25
Section 7 28
Section 8 31
Section 9 35
Section 10 38
Section 11 39
Section 12 40
Appendix A 45
Appendix B 49

MACHINE CODE FOR BEGINNERS

LOADING THE PROGRAM

To load the program press Function key 1 (START). Then press Play on the tape recorder.

RUNNING THE PROGRAM

Detailed information will be found later on in this guide. For the moment it is sufficient to know that the program **simulates** the running of a central processor similar to (but **much** simpler than) the Z80 processor in your ENTERPRISE. When you enter and run programs using the simulator you are running them in the simulator program, and not directly in the Z80 processor of your ENTERPRISE. To run machine-code programs directly in your ENTERPRISE you will need to use a separate program called an ASSEMBLER.

INTRODUCTION

Machine-code programming is enjoyable and efficient. Inevitably it is also a demanding discipline, and this guide does not pretend to cover all the details of this complex subject. The purpose of the program is to aid understanding by presenting the user with a high-quality visual display of the processes taking place when a machine-code program is being run. It is assumed that this guide will be used in conjunction with a good manual of machine-code practice which will provide necessary additional definitive and detailed information.

Short **teaching programs** are given to illustrate the use of all commands available in the simulator, and some longer **demonstration programs** are also given. An appendix provides a simple introduction to binary and hexadecimal numbers.

It is hoped that this program and guide will be of great use to anyone taking the first steps in machine-code programming. Please do experiment! You will progress much more quickly if you write and RUN lots of short programs of your own. Have fun!

Section 1
The central processor and memory

We are concerned here with two important components of your computer. They are:-

1. The memory
2. The central processor.

The memory can be visualised as a set of pigeon holes where numbers are stored. In this simulator there are 256 such pigeon holes called memory locations. (The actual memory of the computer will have many more than this). The pigeon holes are numbered 0 to 255 and the number assigned to each is known as the "ADDRESS".

Care must be taken to distinguish between the **address** and the **contents** of a memory location. As an example look at diagram 1.

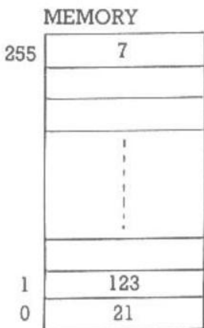


Diagram 1

The number "7" is stored in the memory location whose address is 255, and so on. So for this diagram we have:

Memory location at address	Contents
0	21
1	123
255	7

The central processor interacts with the memory. It can put a number in a pigeon hole, take one out, and perform work on a number, for example by doing arithmetic.

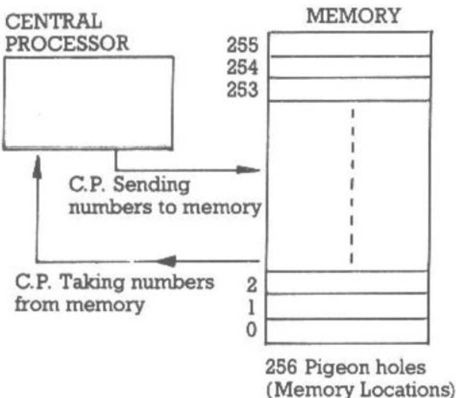


Diagram 2

When a number arrives at the central processor it is stored in an **internal** storage place. The storage places inside the central processor have a special name — they are called REGISTERS. In fact the Z80 central processor has a variety of such registers — 22 in all! — but for the sake of simplicity this simulator program uses only six.

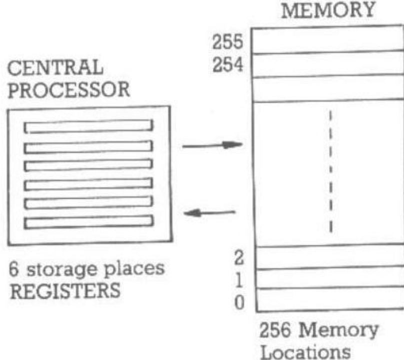


Diagram 3

The Central Processor Registers

As far as the programmer is concerned the registers in the central processor can be classified into two groups.

(a) Those that the PROGRAMMER will be working with in his program.

(b) Those which the CENTRAL PROCESSOR itself uses all the time and which the programmer does not normally need to work with.

In this simulator the first group is represented by these registers:

A (Accumulator)
H (High Byte)
L (Low Byte)

The second group is represented by these registers:

PC (Program Counter)
SP (Stack Pointer)
F (Flags)

(Don't worry about the meaning of the words Accumulator, High Byte, Low Byte, Program Counter, Stack Pointer, Flags — we are just dealing with a broad outline at this stage. Full explanations will come later.

(This simulator also includes a special register called the PAGE register. It has been provided simply to enable you to write advanced programs longer than 256 bytes. Further details will be found in section 10, but you really don't need to worry about it at the moment).

So far, then, we have this picture.

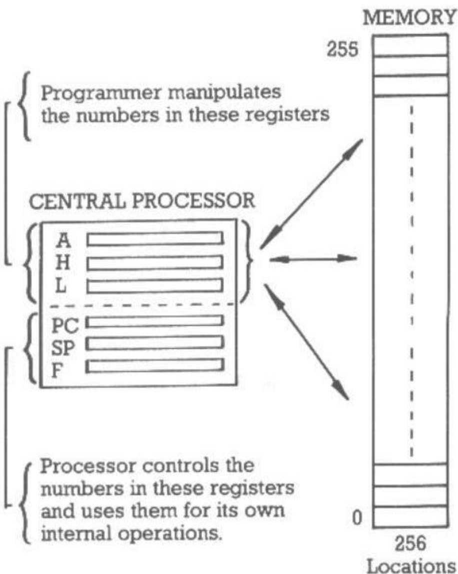


Diagram 4

You now have a broad outline of the relationship between the central processor and the memory. From now on we are going to go into greater detail and you will need to

understand both binary and hexadecimal numbers. If you do not understand these terms please refer to the appendix.

REMINDER

You should feel by now that you know at least something about these words!

CENTRAL PROCESSOR
MEMORY LOCATION
ADDRESS
REGISTERS
BINARY
HEXADECIMAL

Section 2

Using the simulator program — first steps in machine code programming

This program is designed to help you to understand machine-code programming in an interactive way by following through a program visually. We'll begin with a very simple example. First of all load the simulator program into the computer.

Press Function Key 1 (START)

followed by Play on the tape recorder.

When the program is loaded, a menu page will be displayed — press any key to display page 2, and press again to see page 3. You will now see displayed:

Machine Code for Beginners

Choose one
A — Assemble
R — Run
O — Load
S — Save
X — Restart
L — List
P — Printer list
T — Toggle mode
mode - Hex

Diagram 5

First press T to enter decimal mode, then press A. You will now see displayed:-

```
DATA    n              ASSEMBLER
OUT
HALT
LO      s,s              where f = Z,NZ,C,NC
PUSH    p                      s = A,H,L,P,(H)(L)n
POP     p                      p = A,H,L
DEC     p
INC     p
ADD     s
SUB     s
CP      s
AND     s                Start Address?
OR      s
XOR     s
RLA
RRA
SLA
SRA                      L
CALL    f,n
JP      f,n
RET     f
```

Screen Display – ASSEMBLER Page

Diagram 6

Again, **don't worry** if you don't understand this display — you will later on!

Using the ACCUMULATOR register

With the screen display as in diagram 6 type 0 and press ENTER.

You will see displayed

DATA	n	**ASSEMBLER**
OUT		
HALT		
LD	s,s	where f = Z,NZ,C,NC
PUSH	p	s = A,H,L,P,(H)(L)n
POP	p	p = A,H,L
DEC	p	
INC	p	Start writing
ADD	s	ENTER stops
SUB	s	
CP	s	
AND	s	
OR	s	
XOR	s	
RLA		○ L
RRA		
SLA		
SRA		
CALL	f,n	
JP	f,n	
RET	f	

Diagram 7

Now type **EXACTLY:-**

LD A, 123

and press ENTER

Please note that there is a space between LD and A. If you make a typing error just use the erase key in the normal way.

You will now **see displayed diagram 8**

DATA	n	**ASSEMBLER**
OUT		
HALT		
LD	s,s	where f = Z,NZ,C,NC
PUSH	p	s = A,H,L,P,(H)(L)n
POP	p	p = A,H,L
DEC	p	
INC	p	Start writing
ADD	s	ENTER stops
SUB	s	
CP	s	
AND	s	
OR	s	
XOR	s	
RLA		
RRA		
SLA		
SRA		○ LD A, 123
CALL	f,n	
JP	f,n	2 L
RET	f	

Diagram 8

That's a program! A **very** small one — but we can make it work. When it is RUN the number 123 will be stored in the accumulator register.

PRESS ENTER

You will now see displayed diagram 9

```
Machine Code for Beginners

Choose one
  A — Assemble
  R — Run
  O — Load
  S — Save
  X — Restart
  L — List
  P — Printer List
  T — Toggle mode
                        Mode - Dec
```

Diagram 9

We want to RUN our program so PRESS key R. Before the program starts running you will see the following display.

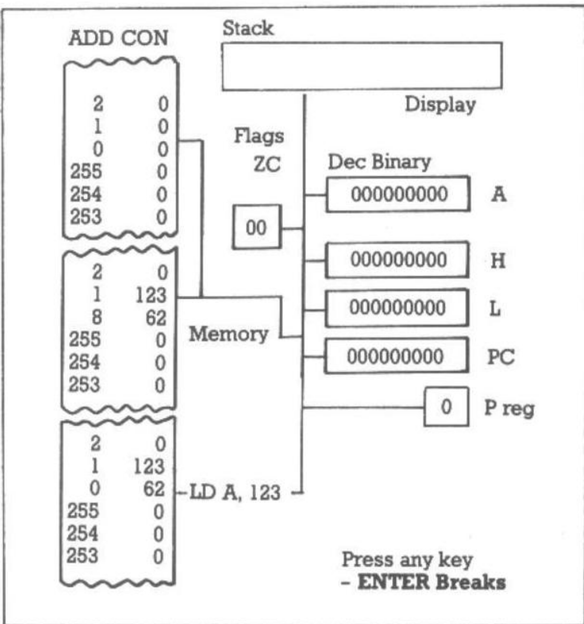
```
Machine Code for Beginners

Choose one
  A — Assemble
  R — Run
  O — Load
  S — Save
  X — Restart
  L — List
  P — Printer List
  T — Toggle mode
      Enter delay (or 0 or R)
                        Mode - Dec
```

Diagram 10

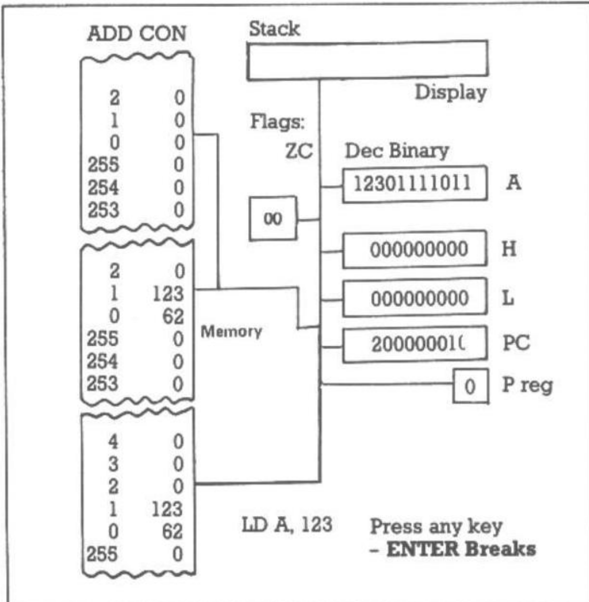
This display is giving you the option of choosing how fast you want the program to run.

For the moment we suggest you type 0 and press ENTER. You will now see displayed diagram 11



Screen display before program is RUN

Diagram 11



Screen display after program is RUN

Diagram 12

Now press any key. The display will change to diagram 12.

Let us now carefully consider what we have done.

Mnemonics and Operands

We typed in

LD A, 123

This is an instruction to the computer to load the Accumulator with the number 123. The letters LD A are called a "Mnemonic". They stand for "Load the accumulator with a number". The number to be stored in the Accumulator register is called the "Operand". So we

have

```
MNEMONIC        OPERAND
   LD A           123
```

Diagram 12 shows that the number 123 has been placed in the accumulator. We see

123 01111011

The number 01111011 is the **binary** way of writing 123. All numbers in a computer are actually dealt with in BINARY. This brings us to our next topic — bits and bytes.

BITS and BYTES

A "BIT" is either a 0 or a 1. "Bit" is short for '**BI**nary — digi**T**' and all numbers in the computer are written in binary numbers made up of bits (0's and 1's). The number of bits that are present in most of the memory locations and registers in a home-computer is 8. So the biggest number that can fit into our accumulator is

11111111

which is worth 255. 8 bits are called a **BYTE**. You will notice that the binary sections of the Accumulator, High, Low and Program Counter registers are all 1 BYTE (eight bits) long.

ASSEMBLERS

Remember that our computer works only with **numbers**, and then re-call our simple program

```
LD A, 123
```

Clearly the instruction "LD A" is not a number! So how does the computer deal with it?

This is where the "Assembler-program" comes in. This program turns the Mnemonic "LD A" into a number known as the "Operation-code" (Op-Code for short) and stores this number in the memory. This Op-Code is an eight bit number to the computer, but for the sake of simplicity humans turn it into hexadecimal.

The Op-Code number for LD A in the Z80 processor is 62 (3E in HEX), and this number can be seen stored in memory location 0.

To sum up:- The function of an assembler is to make machine-code programs easier to write. The instructions which constitute a machine-code program consist purely of

numbers known as "Op-Codes". It would be possible to write the program using these "Op-Codes". It is easier for most people, however, to use Mnemonics, and the assembler provides the facility to translate the mnemonics into Op-Codes and store them somewhere in the memory.

The Program Counter

Notice that in diagram 11 the Program Counter (PC) register is set to 0, and in diagram 12 it is set to 2.

Why is this?

The answer is simple. Look again at our program:

LD A, 123

As explained above, "LD A" is stored as a **one-byte** number in memory. The operand, 123, is translated into

01111011

and is again stored as a 1 byte (i.e. 8 bit) number. We told the simulator to start at Address 0, so the Op-Code for LD A (ie. 62) is stored at byte 0 of the memory, and 123 is stored in byte 1.

The program counter register automatically looks at byte 0. Here it finds LD A and this requires a number to be placed in the A register. This number - in our case 123 - is found at the next memory location, byte 1. The program counter therefore now points at byte 2, since our program has used the first 2 bytes.

Byte 0	Byte 1
LD A,	123

The program counter points to the **next** instruction of the program to be run. It is important to notice that the program counter is dealing with memory locations (ie. bytes) and **NOT** line numbers.

Some instructions such as LOAD ACCUMULATOR (LD A) require 2 bytes. Others may only need 1. The program counter knows how many bytes are needed by each instruction and points at the next instruction automatically.

REMINDER

Now you have worked through this section you should feel

that you know something about the following:-

Assembler Program
Op-Code
Mnemonic
Operand
Bit
Byte
Program Counter

Section 3

More practice with the Simulator. More about "LD" — New instructions HALT, ADD, SUB, JP.

MORE ABOUT "LD"

You will probably feel quite at home now with the program

```
LD A, 123
```

Now make a guess at the meaning of the following

```
LD H, 123
```

and

```
LD L, 123
```

To see if your guess is correct enter the line of code and run it as before. (It is safer, by the way, to press S for "restart" before entering any new program. This clears the memory of the Simulator). This should clarify the meaning of these instructions.

Now try

```
LD H, 7
LD A, H
```

When entered, this will appear on the screen as

```
0 LD H, 7
2 LD A, H
```

Step through the program. When the first line has been run, press any key (except ENTER) to proceed to the next instruction. Bytes 0 and 1 store the number 7 in the H register. Bytes 2 and 3 load into the A register the contents of the H register.

Enter and RUN the following program. Try and work out for yourself what is happening (Press S for restart first)

```
LD L, 6
LD H, 9
LD A, L
LD A, H
LD H, L
LD L, A
```

If you were able to follow this you know quite a lot about the "LD" instruction. There is another way to use "LD": this is known as "indirect addressing" and is dealt with later on.

Before we move on, try the following:

Type

```
LD A
```

and press ENTER

The message "invalid" will appear. This is because LD A by itself is an invalid instruction. The LD A (or LD H etc) instruction requires another number (the OPERAND) to follow it — otherwise it doesn't know **what** it is supposed to load! As we have already seen, the operand could also be a register such as H or L and in that case the contents of the named register will be loaded into the accumulator. Whenever "invalid" appears you will have made a mistake. Simply re-write the line correctly, press ENTER and all will be well.

Some new instructions

HALT This is very simple. It will stop the processor from running further through the program. Try the following.

Restart

```
LD A,3
HALT
LD H, 5
```

and RUN. What happens?

ADD and SUB

These commands **refer to the contents of the Accumulator Register**. This register is the one most used for doing arithmetic and may be thought of as the "WORK BENCH" of the processor.

ADD 5

means add 5 to the present contents of the accumulator register and leave the result in the accumulator.

SUB 2

means subtract 2 from the present contents of the accumulator register and leave the result in the accumulator.

Try the following

```
Restart
   LD H, 7
   LD A, H
   ADD 3
   Sub 6
   LD L,A
```

If all has gone well you should now see the number 4 in the "L" register.

JP This mnemonic stands for JUMP and is like GOTO in BASIC. It must have a number following it. Try the following

```
Restart
   LD A, 1
   ADD 5
   JP 2
```

and RUN. The Display will give

```
   0 LD A, 1
   2 ADD 5
   4 JP 2
```

When the program is RUN the instruction JP 2 will send the processor to byte 2 where it finds the instruction ADD 5. Thus the program keeps adding 5 on to the Accumulator Register Contents. If you RUN the program several times this will become clear.

RUNNING YOUR PROGRAMS WITH A TIME DELAY

Until now you have run your programs one instruction at a time by pressing a key. This is called "single-stepping" a program and is not the way that computers normally run since they usually execute one instruction after the other without manual intervention. The simulator can be made to run like this by choosing a number to give a corresponding delay between the execution of one instruction and the next. You may choose a number from 1 to 255. Entering 0 gives

you the 'single-step' mode that you have already been using.

INC and DEC These instructions increase or decrease the contents of the A, H or L registers by one. Try the following program

```
Restart
    LD A, 7
    LD H, 10
    LD L, 15
    ADD 5
    SUB 2
    INC A
    INC H
    INC L
    DEC A
    DEC H
    DEC L
    HALT
```

Following the program through on the display should make things clear. The main point to notice at this stage is that ADD and SUB only refer to the Accumulator whereas INC and DEC can apply to any of the registers A, H, L.

REMINDER

Now you have worked through this section you should feel that you know something about the following:

LD A, LD H, LD L
HALT
ADD, SUB,
INC, DEC

Section 4

LOGICAL INSTRUCTIONS AND, OR, XOR

These three instructions compare two registers bit by bit. The registers compared can be either

A with H
or A with L

The numbers are compared bit by bit, and the result of

the comparison is placed in the A register. The rules for comparison are:

AND — If both bits are 1, a 1 is placed in the A register. Two 0's or a 0 and a 1 will cause a 0 to be placed in the A register.

So	101	in A
AND	100	in L
will give		
	100	

OR — If either (or both) bit is 1 the result will be 1. Otherwise 0

So	101	in A
OR	100	in H
will give		
	101	in A

XOR — If either bit (but **not** both) is 1 then a 1 will be placed in the A register.

So	101
XOR	100
will give	
	001

XOR is known as "exclusive OR".

To see this in the simulator try these short programs

Restart

```
LD A, 5
LD H, 4
AND H
```

Restart

```
LD A,5
LD L, 4
OR L
```

Restart

```
LD A, 4
LD H, 5
XOR H
```

Now try some more examples using numbers greater than 128 and watch all 8 bits operating.

REMINDER

Now that you have worked through this section you should feel that you know something about the following:

AND
OR
XOR

Section 5
FLAGS. The instructions CP, RRA, RLA, SRA, SLA.

FLAGS. A flag records the result of arithmetic or logical operations in the processor. There are 6 flags in the Z80. This simulator has 2 flags. They are called the CARRY (C) and ZERO (Z) flags. Each flag has two states which may be expressed in several different ways.

TRUE ON SET 1	OR	FALSE OFF CLEAR 0

Diagram 13

Obviously the computer uses 0 and 1

The ZERO flag. To see how this works RUN the following program

Restart
LD A, 30
SUB 30

and watch the Z flag. The instruction

SUB 30

subtracted 30 from the contents of the accumulator leaving zero in the accumulator. This has set the zero flag to 1.

Remember, after an operation:

If the zero flag is set to 1 the result of that operation was zero.

If the zero flag shows 0 the result of that operation was **not** zero.

We interpret the 0 of the zero flag as meaning "the answer in the A register is **not** zero". The instructions in this simulator which can affect the Z flag are:-

ADD	AND
SUB	OR
INC	XOR
DEC	CP

The CARRY flag. RUN the following program

```
Restart
   LD A, 255
   ADD 1
   ADD 1
   SUB 1
   SUB 1
   HALT
```

Watch the Z, C and A registers very carefully as you step through this program. The carry flag of the simulator is set (to 1) if an addition results in a carry or a subtraction requires a borrow (examples are 255 + 1 and 5 − 7). Otherwise the flag will be cleared to zero. In this simulator, the instructions that can affect the carry flag are:-

ADD	AND	RRA
SUB	OR	RLA
INC	XOR	SRA
DEC	CP	SLA

When using this simulator by far the best way to discover how the flags operate is to write and RUN short programs.

Since different processors use their flags in slightly different ways, it is absolutely essential when writing programs for a specific central processor to consult a reference manual for that particular processor.

Flags are essential when dealing with "conditional operations" — a topic that we will touch on later.

The CP Instruction

This instruction performs the subtraction

A — (a number)

and sets the zero and carry flags depending on the result

obtained. The result of the subtraction is not displayed — only the flags are affected.

Run this program

```
Restart
LD A, 40     Places 40 in A
CP 40        Does subtraction
             40 - 40 = 0
             Zero flag set
CP 10        Does subtraction
             40 - 10 = 30
             Neither Z or C flag set
             Does subtraction
CP 60        40 - 60 = 20
             Zero flag not set.
             Carry flag set
HALT
```

Here is another short program for you to try

```
Restart
LD A, 40
CP 60
CP 10
LD H, 60
CP H
HALT
```

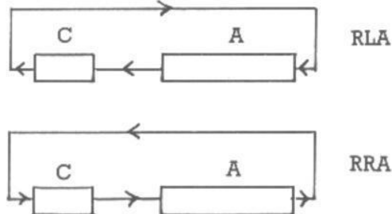
Watch the carry flag carefully.

Remember: the CP instruction affects the flags. The registers A, H, L are unchanged.

RRA and RLA

These instructions **rotate** the contents of the A register to the right or left by one bit — **the carry flag is used as a 9th bit.**

Diagram 14



To illustrate this, step through the following program watching the A register and C flag very carefully.

```
Restart
   0 LD A, 8
   2 RRA
   3 JP 2
```

Now try this

```
Restart
   0 LD A, 8
   2 RLA
   3 JP 2
```

Now try this one

```
Restart
   LD A, 129
   RRA
   RLA
   RRA
   RRA
   RLA
   RLA
   RLA
   RLA
   RLA
   HALT
```

The effect of these instructions should now be clear.

Doubling and Halving

The number 129 in the last example was chosen because it has a 1 at both ends of its binary form. There is, however, an effect of the rotate instructions which is best seen using

a different number. Step through the previous program again, but this time make the first instruction LD A, 16 instead of LD A, 129.ie.

```
LD A, 16
RRA
RLA
RRA
RRA
RLA
RLA
RLA
RLA
HALT
```

Watch the value in the A register. What is happening? Does this always happen? Experiment for yourself and find out. (Look again at the first two simple programs)

SLA and SRA

These mnemonics stand for "shift Accumulator contents to the left" and "shift Accumulator contents to the right". The difference between these instructions and the rotate instructions is that whilst a "1" may be shifted **into** the carry flag, it will not be retrieved. The following program will make this clear.

```
LD A, 129
SRA
SLA
LD A, 129
SLA
SRA
HALT
```

Follow this through several times and experiment with other numbers until you are quite clear about what is going on.

REMINDER Now that you have worked through this section you should feel that you know something about the following

ZERO FLAG
CARRY FLAG
CP
RRA, RLA
SRA, SLA

Section 6

The Stack. PUSH and POP. CALL and RET.

The stack is an area of memory used for storing numbers but it is accessed in a special way. In the simulator the stack starts at the top of the memory at byte 255 page 7 (this refers to page 7 in the simulator memory. Full details about this will be found in section 10, but we suggest that you don't bother with this idea just yet). To see what we mean, RUN this program whilst looking carefully at the stack part of the display

```
Restart
    LD A, 7
    PUSH A
    HALT
```

What has happened? The contents of the A register have been stored in byte 255 of memory — the first stack location.

The instruction

```
    PUSH A
```

puts the contents of A into the stack.

Try this program

```
Restart
    LD A, 4
    LD H, 5
    LD L, 6
    PUSH A
    PUSH H
    PUSH L
    HALT
```

Watch the stack area of the display whilst you are running this program and you will see the stack pointer moving automatically. The stack pointer will now be pointing at byte 253 where 6 is stored.

A stack is often referred to as a "last-in-first-out-store". To see why try this program

```
Restart
    LD A, 4
    LD H, 5
```

```
LD L, 6
PUSH A
PUSH H
PUSH L
POP A
POP H
POP L
HALT
```

Step through this program **carefully**. What happens to the A and L registers?

Now try this program

Restart

```
LD A, 4
LD H, 5
LD L, 6
PUSH A
PUSH H
PUSH L
LD A, 0
LD H, 0
LD L, 0
POP L
POP H
POP A
HALT
```

You will see that the last number to be placed in the stack is the first one to be taken out. It's rather like putting a plate on a 'fast-food' plate dispenser — the last plate on is the first to be removed. You will have realised by now that the instruction to take the next number from the stack and place it in a register is

POP

Both PUSH and POP need to have A, H, or L placed after them

CALL and RET

Now that the idea of the stack has been introduced we can introduce CALL and RET. CALL is a type of jump instruction which uses the stack to store the address to which the program must return after a subroutine. CALL puts the address of the next instruction following the CALL

instruction on the stack and jumps to the start of a subroutine.

RET places into the program counter the contents of the stack. To see how this works carefully step through these programs

Restart

```
0 CALL 3
2 INC A
3 HALT
```

This program jumps to byte 3.

The INC A instruction is never executed. Notice, however, that the address of this instruction, 2, has been placed on the stack. Now try this one

Restart

```
0 CALL 4
2 INC A
3 HALT
4 RET
```

CALL 4 jumps to RET and places address 2 in the stack. RET takes the last value on the stack and puts it into the program counter register so that the program will continue from address 2.

The two commands allow you to jump out of the program to a subroutine and re-enter the program at the correct place. It is like GOSUB in BASIC.

Here is a sample program to illustrate in detail how RET operates.

Restart

```
0 LD A, 5
2 PUSH A
3 RET
4 INC H
5 HALT
```

As the RET instruction takes the last value put into the stack and places it into the program counter, the program will jump from address 3 to address 5 thus missing out the INC H instruction.

CAUTION

CALL and RET are used together to implement a subroutine

and the stack operations are done automatically by the processor. The programmer must be careful always to PUSH and POP numbers in the correct sequence or else the processor's return addresses may become corrupted.

Normal use of PUSH and POP

PUSH is normally used to store the numbers in the processor registers by putting them on the stack at the start of a subroutine call. This enables the subroutine to use the processor registers without losing the numbers previously stored in them.

POP is used at the end of the subroutine to restore the original values which were in the registers before the subroutine was called.

REMINDER You should now know something about-

The STACK
PUSH
POP
CALL
RET

Section 7
CONDITIONAL JUMPS. Indirect Addressing

Conditional Jumps. These are absolutely essential and lie at the heart of computer programming, since they allow the computer to control the flow of the program on the basis of the data.

In this simulator there are 4 possible conditions that can control conditional jumps. They are

C	Carry flag set
NC	Carry flag not set
Z	Zero flag set
NZ	Zero flag not set

Here is an example of a conditional jump being used to divide 35 by 5 and place the answer in the H register

```
Restart
    LD A, 35
    INC H
```

```
SUB 5
JP NZ, 2
HALT
```

RUN this program several times until you are quite clear about what is happening. Here is another program — can you work out what it does? Look at the accumulator register when the program has finished.

```
Restart
   LD H, 8
   LD L, 6
   ADD H
   DEC L
   JP NZ, 4
   HALT
```

This is a very simple program and is slow to run. It is possible to write a much faster program using shift instructions but this is rather more complicated and beyond the scope of an introductory booklet such as this.

There are three instructions in the simulator which can be used to make conditional jumps. They are

CALL, JP, RET

and you have already met them in their unconditional form. For example

```
JP 2
```

means jump to byte 2. The conditional form is

```
JP f, 2
```

where f represents a flag status which can be chosen from Z, NZ, C, NC.

Thus the instruction

```
JP NZ, 8
```

means jump to byte 8 **if** the zero flag shows 0

The instruction

```
CALL C, 26
```

means jump to byte 26 **if** the carry flag shows 1 ie. if the carry flag is set. This should make clear the notation

```
CALL f, n
JP f, n
RET f
```

seen in the ASSEMBLER display (diagram 6). The use of a flag state is optional and these instructions can be used without them.

Indirect Addressing

RUN this short program

Restart

```
LD H, 30
LD L, 40
LD (L), 123
LD (H), 200
LD A, (L)
HALT
```

As you RUN the program watch the contents of the A, H, L registers **and the memory area of the display** very carefully.

An H or L in a bracket

(H) (L)

refers to the memory location at the address given by the current value in the named register. Bearing this in mind step through the program again

LD H, 30	stores 30 in H register
LD L, 40	stores 40 in L register
LD (L), 123	stores 123 in memory location at address 40
LD (H), 200	stores 200 in memory location at address 30
LD A, (L)	stores in the accumulator the contents of the memory location at address 40
HALT	

Write some short, simple programs of your own to check that you really do understand what is going on.

REMINDER You should now know something about

CONDITIONAL and UNCONDITIONAL JUMPS

INDIRECT ADDRESSING

Section 8
The two remaining commands, OUT and DATA. Using hexadecimal numbers. The assembler page display

The OUT instruction

This instruction will place in the display the ASCII character corresponding to the number in the accumulator. No display will result if the number in the accumulator is lower than 32 or higher than 159, because these are non-valid characters. The following program will demonstrate the function of the OUT instruction.

```
Restart
    LD A, 32
    OUT                         Program A
    INC A
    JP 2
```

The program starts printing in the top left-hand corner and continues along the top line. It then moves on to the second line when the first one is full. What does it do when the bottom line has filled up? The display has some special features, and these will be discussed in the next section.

The DATA instruction

(Note that this is not a Z80 command. It is only used by the assembler). This instruction will place a number into the memory at the next available byte.

As an example try this program.

```
Restart
    DATA 1
    DATA 23
    HALT
```

When the RUN this program you will see the number 1 stored in byte 0 of the memory and 23 stored in byte 1. Now try this program

```
Restart
    DATA 62
    DATA 32
    DATA 211                    Program B
    DATA 60
```

```
DATA 195
DATA 2
```

You should find the result is familiar! In fact it performs the same operations as the program which demonstrated the OUT instruction previously. Why is this?

To understand what has happened we need to go back to "Op-Codes". Remember that the computer works only with numbers and the ASSEMBLER translates the mnemonics (such as LD A) into "Op-Codes". If you RUN program A again you will find the numbers that you entered as DATA in program B stored in the memory.

Mnemonic	Op-Code
LD A	62
OUT	211
DEC A	61

This will become clear if you RUN programs A and B several times. You may like to try and enter some other programs using the DATA instruction.

Whilst the above shows what the DATA statement does it does not show how it is normally used. It is used to set up tables of numbers. Try this program which will remind you about indirect memory addressing and a number of other instructions.

Restart

Program C

```
 0 LD H, 13
 2 LD L, 9
 4 LD A, (H)
 5 OUT
 6 INC H
 7 DEC L
 8 XOR A         (A quick way of setting
 9 CP L          Accumulator=0)
10 JP NZ, 4
12 HALT
13 DATA 01H
14 DATA 02H
15 DATA 18H
16 DATA 53H      (N.B. An H
17 DATA 63H      immediately following
18 DATA 72H      a number means that
19 DATA 111      the number is in
                 hexadecimal)
```

```
20 DATA 108
21 DATA 108
```

If you step through this program very carefully several times you will probably be able to work out what is going on. If you need extra help you will find the program printed with full explanatory notes in the "demonstration programs" section of this guide.

Please note that, because of the construction of the simulator program, only 255 data and numeric operands (e.g. instructions like LD A, 55; SUB 30H; XOR 194; LD (H), 69) can be stored by the assembler. When more than 255 are written into the simulator they will still be accepted and coded correctly. However, the disassembled listing shown in the bottom left-hand corner of the register display will appear with extra, apparently meaningless, instructions. These are to be ignored, and the simulator also ignores them.

Using Hexadecimal Numbers

You can enter hexadecimal numbers into programs whenever you wish. Note that in the previous program (program C) both decimal and hexadecimal numbers were used as DATA in bytes 13 to 21. To enter a hexadecimal number simply write an H **immediately** following the number

Thus

23H is equivalent to decimal 35

since

(2x16)+(3x1)=32+3=35

If you wish you can have the display given in hexadecimal and binary simply by choosing the hex option (press H) when the display asks you to choose. (as in diagram 5)

The ASSEMBLER DISPLAY PAGE

This is shown in diagram 6

The meaning of

f=Z, NZ, C, NC
s=AHLP (H) (L) n
p=AHL

when used in conjunction with the mnemonics will probably now be clear. In case any doubts remain, however, here are some examples

DATA n

This means that the DATA instruction can **only** be followed by a number. It cannot be followed by a register or a memory location indirectly addressed.

ADD s

This means that the ADD instruction can be followed by:-
A named register A, H, or L (or P, see section 9).
An indirectly addressed memory location (H) or (L)
A number n

PUSH p

This means that the PUSH instruction can only be followed by a named register A, H, or L

CALL f,n

This means jump to byte n (where n is a number) if the flag condition written at f is met. The flag states can be
Z Zero flag set (1)
NZ Zero flag not set (0)
C Carry flag set (1)
NC Zero flag not set (0)
The use of the flag state, f, is optional. CALL, JP, and RET may all be used without the flag condition if you do not need to use it

The lower case letters f, s, p represent the OPERAND to follow a mnemonic. You must **always** leave one space between a mnemonic and the operand.

Reminder You should now know something about:-
OUT
DATA
USING HEXADECIMAL NUMBERS
ASSEMBLER DISPLAY PAGE

Section 9
Control Codes

When a numeric code is OUTputted to the display the corresponding character appears on the display screen. For example, code 65 produces the letter "A". However, not all codes produce a visible character on the screen. Some of them are called CONTROL CODES and are used to give instructions to do other useful things - for example to clear the screen or to move a printer on to the next line.

To give you some experience of using control codes within this simulator five special codes have been incorporated (numbers 0 to 4). Each of them manipulates the display in a particular way so that you can see what is going on. You will see that this is a powerful way to set up graphics screens easily.

As an example of the use of control codes, look again at program C in section 8. This program features a sideways scroll which has been achieved by using a control code.

Here are the special control codes used in this simulator.

CONTROL CODE 0. Clear Screen.

Code 0 is a clear screen code and the following program shows this.

```
LD A, 48
LD L, 13
OUT                          Program D
INC A
DEC L
JP NZ, 4
LD A, 0
JP 0
```

CONTROL CODE 1. SINGLE LINE MODE.

Normally the display area shows five lines of text, but this can be altered to one by using code 1. Program E shows this mode in use, and also what happens when the line becomes full.

```
LD A, 1
OUT
LD H, 3
LD L, 26
```

```
LD A, 65
OUT                              Program E
INC A
DEC L
JP NZ, 9
DEC H
JP NZ, 5
HALT
```

CONTROL CODE 4 is the opposite of code 1. Alter program E to print code 1 after it has printed, say twenty normal characters, and then observe the effect when another ten normal characters are printed.

CONTROL CODES 2 and 3. DEFINING A PRINT POSITION.

Codes 2 and 3 both need a second byte (after the control code) to be sent out to the display before any action is taken. Code 2 is like the PRINT AT command in BASIC and allows printing to be done at any position in the display. The second byte sent to the display specifies where the next character is to be printed. For reference, any number:-

between 0 and 24 will print on the top line (0 for left hand side, 24 for the right-hand side),
between 25 and 49 will print on the second line;
between 50 and 74 will print on the third line;
between 75 and 99 will print on the fourth line;
between 100 and 124 will print on the bottom line.

Program F illustrates this. Experiment by changing the value of H in the first line.

```
LD H, 15
LD L, 5
LD A, 2
OUT
LD A, H
OUT
ADD 25                           Program F
LD H, A
LD A, 65
OUT
DEC L
JP NZ, 4
HALT
```

Note what happens if the byte after code 2 is bigger than 124 (i.e. an invalid screen position). Also note that if the display is in single line mode, then to specify a print position only values between zero and twenty-four are valid (0 for the LHS, 24 for the RHS of the middle line) Program G below will help show this. Again experiment by changing the value of H in the first line.

```
LD H, 12
LD A, 1
OUT
LD A, 2                     Program G
OUT
LD A, H
OUT
LD A, 90
OUT
HALT
```

Control Code 3 does the opposite of Code 2. It returns to be A register the ASCII code of the character on the display at the specified position. Again the co-ordinates for the display positions run from 0-24 for the top line, all the way down to 124 for the bottom right-hand corner. This is a very useful feature for games as the program can test, say, a missile hitting its target. Program H shows control code 3 in use; watch the A register very carefully when it OUTputs the location byte (line 15 of the program).

```
LD L, 5
LD A, 65
OUT
INC A
DEC L
JP NZ, 4
LD H, 6
LD A, 3                     Program H
OUT
LD A, H
OUT
DEC H
JP NZ, 11
HALT
```

Notice that this time if the display is in single line mode, the co-ordinates are still the same as for the full five-line mode.

Section 10
The P register

If you wish to write a long program on the simulator you may find that 256 bytes of memory is not enough. Because of this the simulator has been fitted with eight lots of 256 bytes. However the Program Counter register (PC) only goes up to 255, so what happens when the program wants to count higher?

A special register called the P (page) register has been introduced which is really an extension of the PC register, and tells the PC register which block of 256 bytes the program is in. This is best illustrated with the following program. First type in this command at address 0

```
JP 254
```

then come out of the assembler (by pressing ENTER) and go back in again by pressing A. When it asks for the start address type 254. Now type in the following:

```
INC A
INC A
INC A
INC A
INC A
HALT
```

You will have noticed at the bottom of the screen the page change from zero to one at the same time as the address of the INC A's you were typing in change from 255 to 0. This number at the bottom is a page number (and shows which block of 256 bytes you are in). Now run the program and watch very carefully the PC and P registers.

By the way "Paging" is actually used on the Enterprise computer, where the central processor is required to address a much larger memory than its normal address range. In this case the Page register is not actually a processor register but part of the hardware connecting the processor to the memory.

JP and CALL

Now, what do we do if we want to Jump or Call a piece of program that is in a different block of bytes, or, which block of bytes does the program jump to if we write the command JP 59?

Again, by programming the P register we can jump to any block we like.

Type in the following:-

```
LD P, 2
JP 5
INC A
INC A
HALT
```

Then go back to the assembler again and type P2, 5 when it asks for the start address. This tells the assembler to start on page 2) and is shown at the bottom of the screen). Now type in

```
DEC A
HALT
```

and run the program. You will see that it has missed out the INC A instructions and gone straight to the DEC A instruction. So now we can jump to any address in any of the eight blocks (numbered zero to seven) by setting the P register first. Note that if you set the P register before doing a CALL operation, it must be set back to its previous value before executing the RET instruction, or your program will fail.

The P register can also be set before doing any indirect addressing operation to access bytes in any block.

Section 11
Fast running. Load and Save. Listing

FASTER RUNNING

Type in the following program;

```
Restart
    LD A, 79
    OUT
    JP 2
```

When you run this with a delay the program does, of course, run slowly. If you wish, you can choose to have the program run much faster. To do this select the "R" mode when asked for a delay. You will now see the display area only in the middle of the screen and the program runs much faster. This is because the simulator program does not have

to update any register on the screen. You still need to press the ENTER key to get out of the program.

Although in R mode programs run very fast this is still about 200 times slower than pure machine-code running without the simulator.

SAVE and LOAD

On the options page, you will find that you can Save and Load your program. Pressing S or O will ask you for the program name (up to fifteen characters in length). Just pressing ENTER will save the program with no name, or load in the very next program. Programs loaded in can be amended or added to as normal.

There is a specimen program called DEMONSTRATE on the tape immediately after the "Machine Code for Beginners" program.

LISTING

At any time you can list your program (on the screen or on a printer) by pressing "L" or "P". To go back to the menu page at any time, just press the ENTER key.

Section 12
Demonstration Programs, plus a note on the Z80 Microprocessor

Here are some programs for you to try which use all of the commands available in the simulator. If you can follow them through and really understand what is going on you will have made a good start in understanding the fundamentals of machine code. You will be in an excellent position to make a start with a full assembler program which utilizes all of the Z80 instruction set. We hope that you have enjoyed this introduction to machine-code programming, and that you will also enjoy expanding your knowledge in the future.

Program 1 **Simple counting in binary and decimal or binary and hexadecimal**

```
Restart
0  LD A, 0                    Places 0 in accumulator
2  INC A                      Increase accumulator by 1
```

3 JP 2	Jump unconditionally to byte 2

Program 2 **Demonstrates the Display**

Restart

0 LD A, 32	Load accumulator with 32
2 OUT	Print corresponding ASCII character in display. (32 is the "space" character)
3 INC A	Add 1 to accumulator
4 CP 128	When the accumulator has been incremented to 128 the zero flag will be set to 1
6 JP NZ, 2	Will direct program back to byte 2 if zero flag **not** set. If the zero flag is set the program will proceed to byte 8.
8 HALT	

Program 3 **Illustrates memory addressing**

Restart

0 LD H, 13	Loads H with the address of the first data statement
2 LD L, 9	Loads into L the number of DATA statements written after HALT
4 LD A, (H)	Load the accumulator with the memory location pointed to by H
5 OUT	Send ASCII character to display
6 INC H	Makes H point to next DATA statement
7 DEC L	Subtract 1 from L
8 XOR A	A quick way to set A = 0
9 CP L	Compare L with zero
10 JP NZ, 4	If zero flag is 0 jump to byte 4 If zero flag is 1 go to byte 12
12 HALT	Stop the program
13 DATA 01H	
14 DATA 02H	
15 DATA 18H	Data entered in Hex
16 DATA 53H	
17 DATA 63H	
18 DATA 72H	
19 DATA 111	

```
20 DATA 108          Data entered in decimal
21 DATA 108
```

Programs 4 and 5 are longer and more complicated. Step through them slowly and try to work out what is happening.

Program 4 **This program prints in the display**

5–3=2 5+3=8

It demonstrates the use of the STACK

Restart

```
 0 LD H, 18
 2 LD L, (H)
 3 INC H
 4 LD H, (H)
 5 LD A, 32
 7 CALL 20
 9 LD A, 32
11 OUT
12 OUT
13 LD A, 26
15 CALL 20
17 HALT
18 DATA 3
19 DATA 5
20 PUSH A
21 LD A, H
22 ADD 48
24 OUT
25 RET
26 ADD L
27 PUSH A
28 LD A, 43
30 JP 36
32 SUB L
33 PUSH A
34 LD A, 45
36 OUT
37 LD A, L
38 ADD 48
40 OUT
41 LD A, 61
43 OUT
44 POP A
45 OUT
46 RET
```

Program 5 **Doubles a number, in decimal.** Written in Hex Mode

```
Restart
00 INC A
01 LD H, 07H
03 PUSH A
04 INC L
05 SUB OAH
07 JP NC, 04H
09 ADD OAH
0B PUSH A
0C DEC L
0D LD A, 20H
0F OUT
10 OUT
11 LD A, L
12 CALL 22H
14 POP A
15 CALL 22H
17 POP A
18 ADD A or SLA
19 DEC H
1A PUSH A
1B LD A, H
1C CP 00H
1E POP A
1F JP NZ, 03H
21 HALT
22 ADD 30H
24 OUT
25 LD L, 00H
27 RET
```

THE Z80 MICROPROCESSOR

The Z80 Microprocessor (the logical step forward after this simulator) contains two banks of general purpose registers each being A,F,B,C,D,E,H,L where A is the accumulator and F is the flags register. The program counter and Stack Pointer are both 16 bit registers, and pointing to memory locations is done using register pairs: B and C, D and E, H and L. This then creates a 16 bit pointer. There are also index registers IX and IY that use an offset byte to point to any location. Jumps can also be performed relatively. There

is also a comprehensive set of rotating, shifting and bit manipulation routines.

Many more advanced books are available which will give you as much detail as you feel you would like to have.

APPENDIX A

DECIMAL NUMBERS (also called base 10)

Our everyday counting numbers are written in decimal. For example, consider the number 5432

	10x10x10	10x10	10x1	
Place value headings	1000	100	10	1
	5	4	3	2

we know that the

5 is worth 5 x 1000
4 is worth 4 x 100
3 is worth 3 x 10
2 is worth 2 x 1

Hence we read the number as five thousand, four hundred and thirty two. The place value headings of 1000, 100, 10, 1 are based on the number 10.

In base 10 there are 10 different symbols for numbers, they are

0 1 2 3 4 5 6 7 8 9

Summary: In base 10

1) the place value pattern is based on 10
we have 1000 100 10 1
2) there are 10 different number symbols

BINARY NUMBERS (also called base 2)

There are two fundamental facts to know and understand about binary numbers. They are,

1) that the place value pattern is based on 2 - it looks like this

Place value headings	2x2x2x2x2	2x2x2x2	2x2x2	2x2	2x1	
	32	16	8	4	2	1

2) There are 2 different number symbols: 0,1

To work out the decimal value of a binary number we need to position the binary place value pattern over it.

For example, consider the binary number

10110

write the place value pattern over it

16	8	4	2	1
1	0	1	1	0

this number is worth

$$1 \times 16 = 16$$
$$0 \times 8 = 0$$
$$1 \times 4 = 4$$
$$1 \times 2 = 2$$
$$0 \times 1 = 0$$

Total 22

Any number can be written in binary using only 0 and 1. Here are some examples:

	Binary number						Decimal value
Place value pattern	32	16	8	4	2	1	
						1	1
					1	0	2
					1	1	3
				1	0	0	4
				1	0	1	5

Program 1 of the demonstration programs will display counting in both decimal and binary in the accumulator if you choose the decimal option.

Summary:

In binary numbers

1) the place value pattern is based on 2
we have 64 32 16 8 4 2 1
2) there are 2 different number symbols

HEXADECIMAL NUMBERS (also called base 16)

There are two fundamental facts to know and understand about hexadecimal numbers (Hex). They are,

1) that the place value pattern is based on 16 — it looks like this

Place value headings	16x16x16x16	16x16x16	16x16	16	
	65536	4096	256	16	1

2) there are 16 different number symbols. As we usually only have 10 symbols to use (i.e. 0 to 9) we need to invent some more. We use the letters A, B, C, D, E, F as our new numbers.

Hence we have,

DECIMAL	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
HEX	0 1 2 3 4 5 6 7 8 9 A B C D E F

Letter A is worth 10, B is worth 11 and so on

To work out the decimal value of a Hex number we need to position the hex place value pattern over it.

For example, consider the hex number

3AB

(usually written 3ABH — the 'H' indicates that the number is in hex)

write the place value pattern over it

256	16	1
3	A	B

this number is worth

	Decimal value
3 x 256	= 768
10 x 16	= 160
11 x 1	= 11
Total	= 939

Another example

FDE (usually written FDEH)

write the place value pattern over it

256	16	1
F	D	E

this number is worth

	Decimal value
15 x 256	= 3840
13 x 16	= 208
14 x 1	= 14
Total	= 4062

Summary In Hex (base 16)

1) the place value pattern is based on 16 we have 256 16 1
2) we have 16 different number symbols, the extra ones are

A = 10
B = 11
C = 12
D = 13
E = 14
F = 15

3) we usually write 'H' after a hex number (some computers ask for another symbol, for example "&" before the number)

WHY USE HEXADECIMAL?

The microprocessor uses registers that are made up of 8 bits (where a bit is either a "1" or "0"). By setting (1) some bits and resetting (0) other bits any decimal number between 0 (binary 00000000) and 255 (binary 11111111) can be represented.

We use hex as a convenient way to simplify the writing of 8 bit binary numbers. Four binary bits can represent any number between 0 and 15. The following table will make this clear.

BINARY	DECIMAL	HEX	BINARY	DECIMAL	HEX
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

So an 8 bit binary number can be represented by two hex characters, e.g.

DECIMAL	219		101	
BINARY	1101	1011	0110	0101
HEX	D	B	6	5

Program 1 of the Demonstration Programs will display counting in both hex. and binary in the accumulator if the "hex" option is chosen.

APPENDIX B
Summary of the simulator in use
Summary list of instructions available with comments on usage

USING THE SIMULATOR

When the menu page is displayed press A to start writing a program. For the starting address type 0 (followed by 'ENTER') as this is where all programs are initially executed from;

You are now in the Assembler. All instructions require one space between the op code (LD, CALL, RET etc.) and the operand (p,s,f etc.). Numbers can be entered in decimal or hex, with hex numbers requiring an H after them to distinguish them from decimal numbers. The erase key functions are normal.

After successfully entering the last instruction of a program pressing 'ENTER' (instead of writing another command) will cause the mother program to come out of the assembler and give eight choices:

a) to go back into the assembler to add a bit more to the program/put in a table/alter an instruction.
b) restart - resets all the memory ready for a new program.
c) save
d) load
e) screen and printer list
f) change mode to Decimal or Hexadecimal
g) Run

Running will ask for a time delay between instructions. Anything from 1 to 255 (00H to FFH) will give a delay of increasing length. Entering 0 will wait for a key to be pressed before executing the next instruction.

When running, the current instruction (starting at 0) is shown, and this is executed after the delay or key wait. This idea is especially advantageous when a Jump, Call or Return instruction is about to be executed.

At any time the program can be stopped (useful for programs that get stuck in loops) by pressing the 'ENTER' key.

A HALT command (essential at the end of all programs)

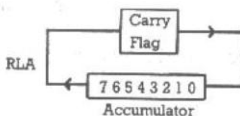
stops the program, waits for 'ENTER' to be pressed and returns to the choices.

Instructions Available

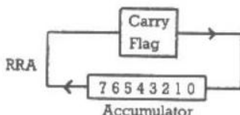
Op Code	**Operand**	
ADD	s	Add to the accumulator (register A) the contents of the register s, leaving the results in the accumulator.
AND	s	AND with the accumulator the contents of register s. Results are put in the accumulator.
CALL	f,n	If the condition f is met then put the address of the next instruction following the CALL instruction on to the stack and jump to address n.
CP	s	Do the subtraction A-s but only set the carry and zero flag according to the result of the subtraction; i.e. leave the registers A and s as they were.
DATA	n	Put into the memory the byte n (Note that this is not a Z80 command and is used only by the assembler).
DEC	p	Take away 1 from the register p.
HALT		Stop the program and return back to the mother program.
INC	p	Add 1 to the register p.
JP	f,n	If condition f is met then jump to the program at line n.
LD	s_a, s_b	Put into register s_a the contents of register s_b.
OR	s	OR with the accumulator the contents of register s and place the result back in the accumulator.
OUT		Write to the display the ASCII character corresponding to the value of the accumulator. Note that this is ignored if the value of A is less than 32 or greater than 159

because these are non-valid characters.

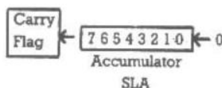
POP	p	Load into the register p the contents as pointed to by the Stack Pointer.
PUSH	p	Place onto the stack the contents of the register p.
RET	f	If the condition f is met then place into the program counter the contents of the stack.
RLA		Rotate to the left, through the carry flag, the contents of the Accumulator.



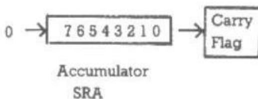
RRA		Rotate to the right, through the carry flag, the contents of the Accumulator.



SLA		Shift to the left through the carry flag the contents of the Accumulator leaving a zero in bit 0.



SRA		Shift to the right through the carry flag the contents of the Accumulator leaving a zero in bit 7.



SUB	S	Subtract from the accumulator the contents of register s leaving the result in the accumulator.
XOR	S	Exclusive OR with the accumulator the contents of the register s leaving the result in the accumulator.

The f operand is optional

ENTERPRISE

PROGRAMS