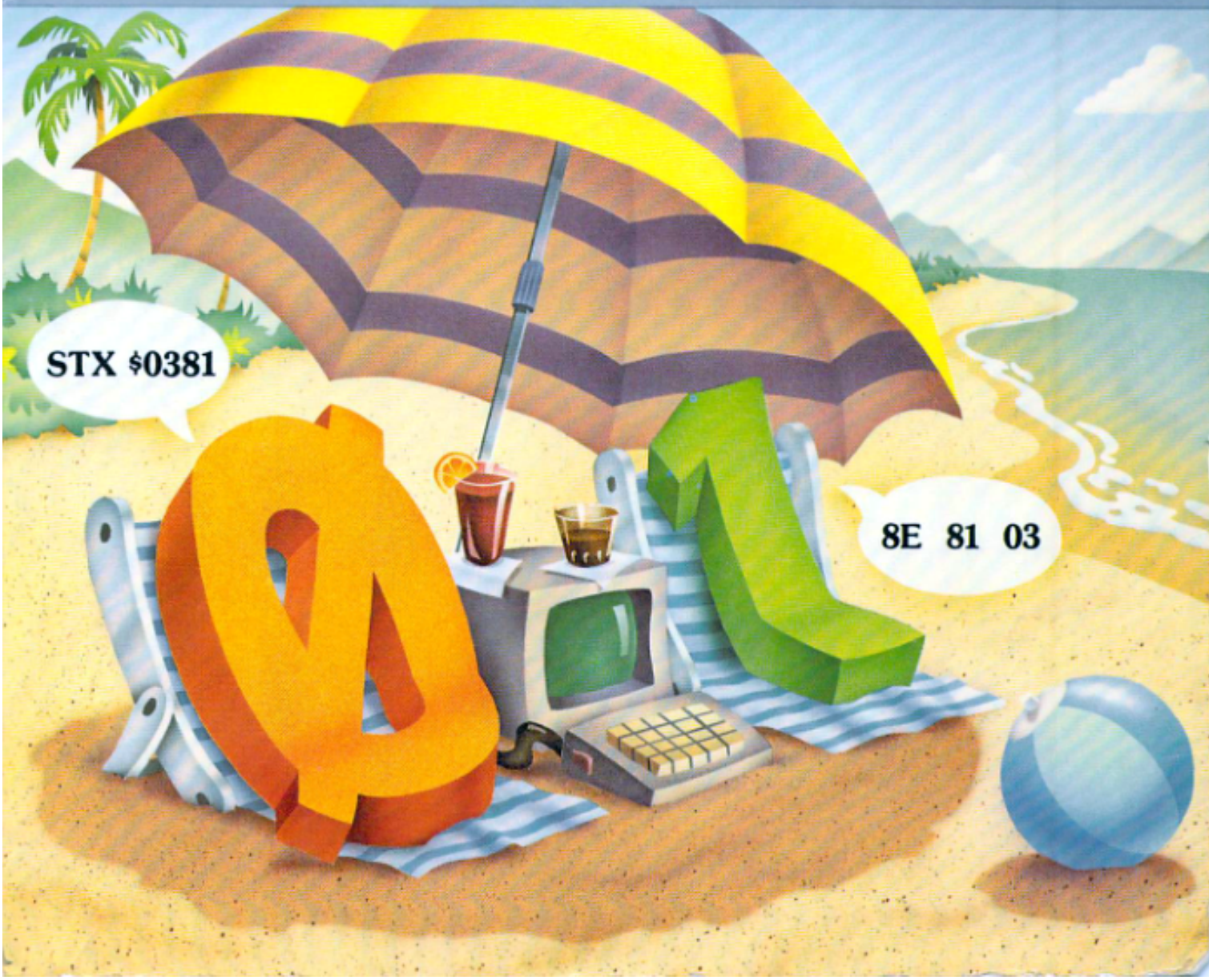
Brady

JIM BUTTERFIELD

MACHINE LANGUAGE

FOR THE COMMODORE 64, 128, AND OTHER COMMODORE COMPUTERS

REVISED & EXPANDED EDITION



Machine Language for the Commodore 64, 128, and Other Commodore Computers

A Brady Book
Published by Prentice Hall Press
A Division of Simon & Schuster, Inc.
Gulf + Western Building
One Gulf + Western Plaza
New York, New York 10023

PRENTICE HALL PRESS is a trademark of Simon & Schuster, Inc.

Manufactured in the United States of America

1 2 3 4 5 6 7 8 9 10

Library of Congress Cataloging in Publication Data

Butterfield, Jim
Machine language for the Commodore 64, 128, and other Commodore computers

Includes index.
1. Commodore 64 (Computer)–Programming. 2. Commodore computers–Programming. 3. Programming languages (Electronic computers) I. Title.
QA76.8.C64B88 1986 001.64'2 84-6351

ISBN 0-89303-652-8

...

Contents

Note to Readers

This book introduces beginners to the principles of machine language: what it is, how it works, and how to program with it.

It is based on an intensive two-day course on machine language that has been presented many times over the past five years.

Readers of this book should have a computer on hand: students will learn by doing, not just by reading. Upon completing the tutorial material in this book, the reader will have a good idea of the fundamentals of machine language. There will be more to be learned; but by this time, students should understand how to adapt other material from books and magazines to their own particular computers.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

Note for Commodore 128 Owners

The Commodore 128 is three machines in one: a Commodore 64, a Commodore 128, and a CP/M machine. You may select any of the three at any time.

If you choose the Commodore 64 mode, you'll find examples within this book that will work on your machine. The programs you write will be compatible with other ("real") Commodore 64 computers. But you'll lose access to extra memory and to other features of the new machine. In particular, you won't

have a built-in machine language monitor and will need to load one from tape or disk.

If you choose the Commodore 128 mode, you're working with a richer and more powerful machine. You will have a built-in machine language monitor for speed and convenience, and access to new features such as 80 columns, with extra complexity. There are new rules to be learned. This book contains extra material to enable you to cope with the new features of the C128.

If you choose CP/M mode, you will be in an environment that is quite different from other Commodore machines. This book, working with the 64 or 128 mode, can teach you principles of machine language and skills which may be carried to other computer environments, including CP/M. But it will not teach you CP/M itself or CP/M's machine language.

A Commodore 128 owner can read each chapter of this book twice, if desired. The first time, the exercises for the Commodore 64 can be worked through; the second time, those for the 128 can be used. The principles are the same; the code is similar; but the 128 often calls for a little more detailed work.

If you wish to learn machine language for the Commodore 128, please read the Introduction in Appendix E, under Exercises for the Commodore 128. It will give you some starting facts about your machine. There is more information on the 128 in the latter section of Appendix B and elsewhere, but don't try to read it all at the start. It will be there when you need it.

Preface

This book is primarily tutorial in nature. It contains, however, extensive reference material, which the reader will want to continue to use.

No previous machine language experience is required. It is useful if the reader has had some background in programming in other languages, so that concepts such as loops and decisions are understood.

Beginners will find that the material in this book moves at a fast pace. Stay with it; if necessary, skip ahead to the examples and then come back to reread a difficult area.

Readers with some machine language experience may find some of the material too easy; for example, they are probably quite familiar with hexadecimal notation and don't need to read that part. If this is the case, skip ahead. But do enter all the programming projects; if you have missed a point, you may spot it while doing an exercise.

Programming students learn by doing. The beginner needs to learn simple things about his or her machine in order to feel in control. The elements that are needed may be itemized as:

- Machine language. This is the objective, but you can't get there without the next two items.

- Machine architecture. All the machine language theory in the world will have little meaning unless the student knows such things as where a program may be placed in memory, how to print to the screen, or how to input from the keyboard.

- Machine language tools. The use of a simple machine language monitor to read and change memory is vital to the objective of making the computer do something in machine language. Use of a simple assembler and elements of debugging are easy once you know them; but until you know them, it's hard to make the machine do anything.

Principles of sound coding are important. They are seldom discussed explicitly, but run as an undercurrent through the material. The objective is this: it's easy to do things the right way, and more difficult to do them the wrong

way. By introducing examples of good coding practices early, the student will not be motivated to look for a harder (and inferior) way of coding.

It should be pointed out that this book deals primarily with machine language, not assembly language. Assembler programs are marvellous things, but they are too advanced for the beginner. I prefer to see the student forming an idea of how the bytes of the program lie within memory. After this concept is firmly fixed in mind, he or she can then look to the greater power and flexibility offered by an assembler.

Acknowledgements

Thanks go to Elizabeth Deal for acting as resource person in the preparation of this book. When I was hard to find, the publisher could call upon Elizabeth for technical clarification.

Introduction

Why learn machine language? There are three reasons. First, for speed; machine language programs are fast. Second, for versatility; all other languages are limited in some way, but not machine language. Third, for comprehension; since the computer really works in machine language only, the key to understanding how the machine operates is machine language.

Is it hard? Not really. It's finicky, but not difficult. Individual machine language instructions don't do much, so we need many of them to do a job. But each instruction is simple, and anyone can understand it if he or she has the patience.

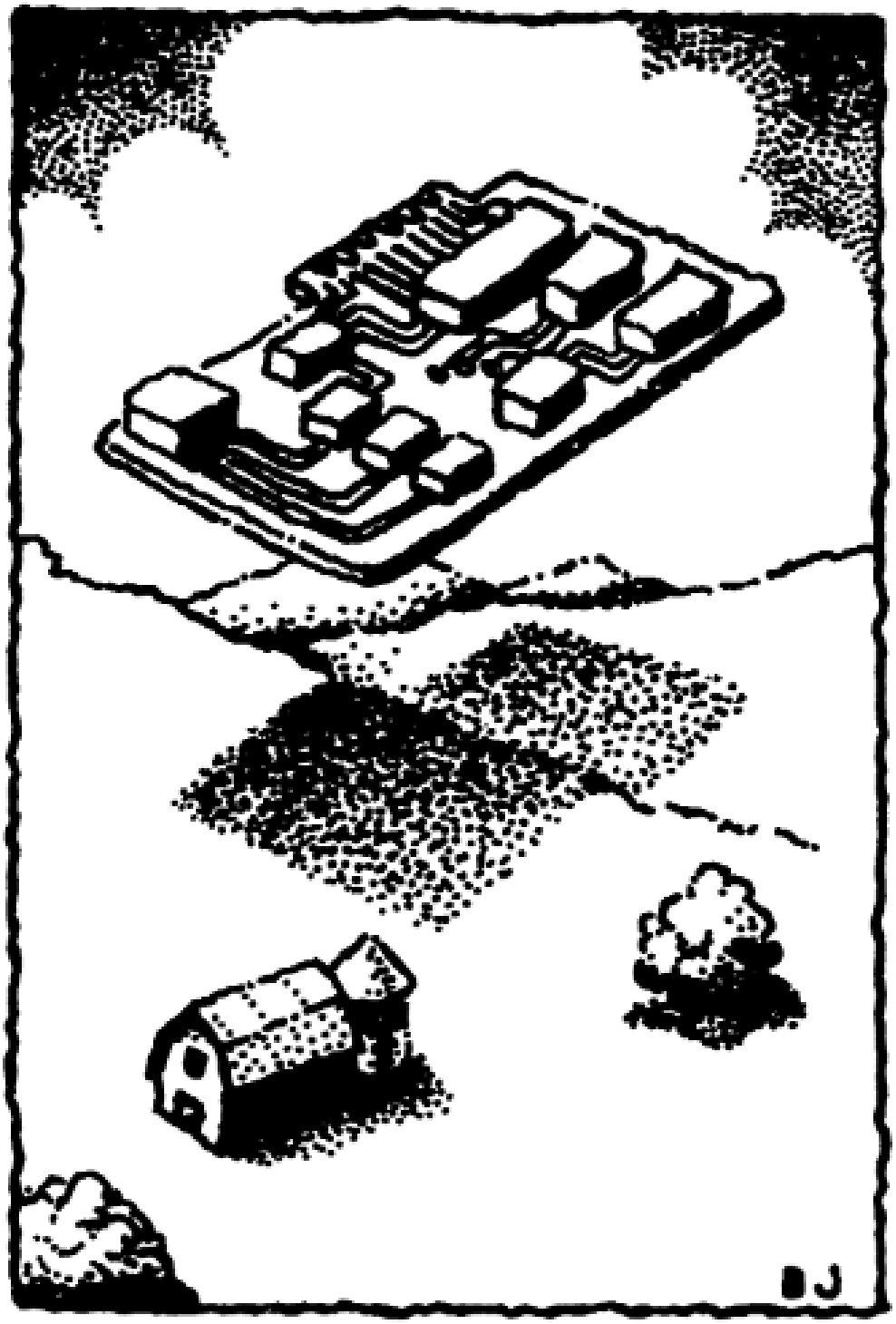
Some programmers who started their careers in machine language find "higher level" languages such as BASIC quite difficult by comparison. To them, machine language instructions are simple and precise, whereas BASIC statements seem vague and poorly defined by comparison.

Where will this book take you? You will end up with a good understanding of what machine language is, and the principles of how to program in it. You won't be an expert, but you'll have a good start and will no longer be frightened by this seemingly mysterious language.

Will the skills you learn be transportable to other machines? Certainly. Once you understand the principles of programming, you'll be able to adapt. If you were to change to a non-Commodore machine that used the 6502 chip (such as Apple or Atari), you'd need to learn about the architecture of these machines and about their machine language monitors. They would be different, but the same principles would apply on all of them.

Even if you change to a computer that doesn't use a chip from the 6502 family, you will be able to adapt. As you pick through the instructions and bits of the Commodore machine, you will have learned about the principles of all binary computers. You will need to learn the new microprocessor's instruction set, but it will be much easier the second time around.

Do you need to be a BASIC expert before tackling machine language? Not at all. This book assumes you know a little about programming fundamentals: loops, branching, subroutines, and decision making. But you don't need to be an advanced programmer to learn machine language.


BJ

Chapter 1

First Concepts

This chapter discusses:

- The inner workings of microcomputers
- Computer notation: binary and hexadecimal
- The 650x's inner architecture
- Beginning use of a machine language monitor
- A computer's "memory layout"
- First machine language commands
- Writing and entering a simple program

1.1 The Inner Workings of Microcomputers

All computers contain a large number of electrical circuits. Within any binary computer, these circuits may be in only two states: "on" or "off."

Technicians will tell you that "on" usually means full voltage on the circuit concerned, and "off" means no voltage. There's no need for volume control adjustments within a digital computer: each circuit is either fully on or fully off.

The word "binary" means "based on two," and everything that happens within the computer is based on the two possibilities of each circuit: on or off. We can identify these two conditions in any of several ways:

ON or OFF
TRUE or FALSE
YES or NO
1 or 0

The last description, 1 or 0, is quite useful. It is compact and numeric. If we had a group of eight circuits within the computer, some of which were "on" and others "off," we could describe their conditions with an expression such as:

```
11000111
```

This would signify that the two leftmost wires were on, the next three off, and the remaining three on. The value 11000111 looks like a number; in fact, it is a binary number in which each digit is 0 or 1. It should not be confused with the equivalent decimal value of slightly over 11 million; the digits would look the same, but in decimal each digit could have a value from 0 to 9. To avoid confusion with decimal numbers, binary numbers are often preceded by a percent sign, so that the number might be shown as `%11000111`.

Each digit of a binary number is called a bit, which is short for "binary digit." The number shown above has eight bits; a group of eight bits is a byte. Bits are often numbered from the right, starting at zero. The right-hand bit of the above number would be called "bit 0," and the left-hand bit would be called "bit 7." This may seem odd, but there's a good mathematical reason for using such a numbering scheme.

The Bus

It's fairly common for a group of circuits to be used together. The wires run from one microchip to another, and then on to the next. Where a group of wires are used together and connect to several different points, the group is called a bus (sometimes spelled "buss").

The PET, CBM, and VIC-20 use a microprocessor chip called the 6502. The Commodore 64 uses a 6510. The Commodore B series uses a 6509 chip, and

the Commodore PLUS/4 uses a chip called 7501. All these chips are similar, and there are other chips in the same family with numbers like 6504; every one works on the same principles, and we'll refer to all of them by the family name 650x.

Let's take an example of a bus used on any 650x chip. A 650x chip has little built-in storage. To get an instruction or perform a computation, the 650x must call up information from "memory"–data stored within other chips.

The 650x sends out a "call" to all memory chips, asking for information. It does this by sending out voltages on a group of sixteen wires called the "address bus." Each of the sixteen wires may carry either voltage or no voltage; this combination of signals is called an *address*.

Every memory chip is connected to the address bus. Each chip reads the address, the combination of voltages sent by the processor. One and only one chip says, "That's me!" In other words, the specific address causes that chip to be selected; it prepares to communicate with the 650x. All other chips say, "That's not me!" and will not participate in data transfer.

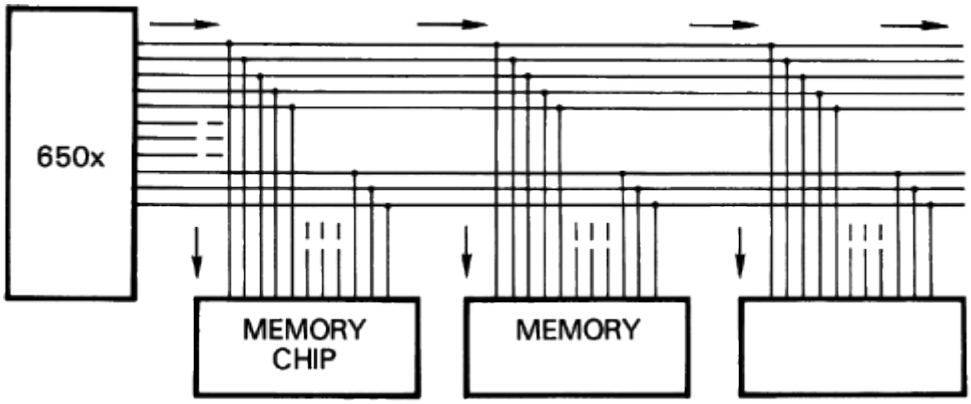


Figure 1.1: Address bus connecting 650x & 3 chips

The Data Bus

Once the 650x microprocessor has sent an address over the address bus and it has been recognized by a memory chip, data may flow between memory and 650x. This data is eight bits (it flows over eight wires). It might look like this:

```
01011011
```

The data might flow either way. That is, the 650x might *read* from the memory chip, in which case the selected memory chip places information onto the data bus which is read by the microprocessor. Alternatively, the 650x might wish to *write* to the memory chip. In this case, the 650x places information

onto the data bus, and the selected memory chip receives the data and stores it.

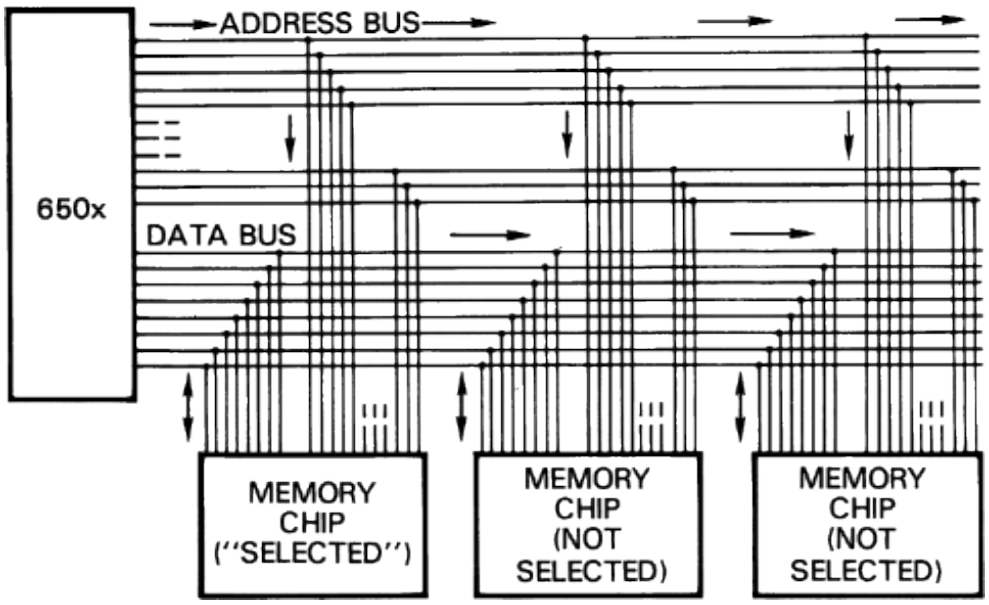


Figure 1.2: Two-way data bus

All other chips are still connected to the data bus, but they have not been selected, so they ignore the information.

The address bus is accompanied by a few extra wires (sometimes called the control bus) that control such things as data timing and the direction in which the data should flow: read or write.

Number Ranges

The address bus has sixteen bits, each of which might be on or off. The possible combinations number 65536 (two raised to the sixteenth power). We then have 65536 different possibilities of voltages, or 65536 different addresses.

The data bus has eight bits, which allows for 256 possibilities of voltages. Each memory location can store only 256 distinct values.

It is often convenient to refer to an address as a decimal number. This is especially true for PEEK and POKE statements in the BASIC language. We may do this by giving each bit a "weight." Bit zero (at the right) has a weight of 1; each bit to the left has a weight of double the amount, so that bit 15 (at the left) has a weight of 32768. Thus, a binary address such as

```
0001001010101100
```

has a value of 4096 + 512+128 + 32 + 8 + 4 or 4780. A `POKE` to 4780 decimal would use the above binary address to reach the correct part of memory.

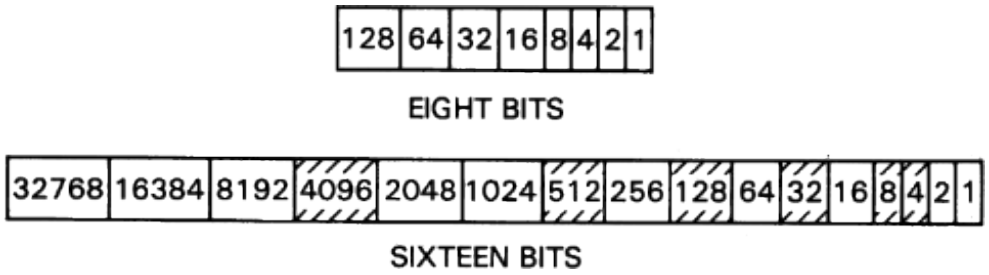


Figure 1.3:

Hexadecimal Notation

Binary is an excellent system for the computer, but it is inconvenient for most programmers. If one programmer asks another, "What address should I use for some activity?", an answer such as `"Address %0001001010101100"` might be correct but would probably be unsatisfactory. There are too many digits. Hexadecimal is a code used by humans to conveniently represent binary numbers. The computer uses binary, not hexadecimal; programmers use hexadecimal because binary is cumbersome. To represent a binary number in hexadecimal, the bits must be grouped together four at a time. If we take the binary value given above and split it into groups of four, we get

```
0001 0010 1010 1100
```

Now each group of four bits is represented by a digit as shown in the following table:

```
0000-0  0100-4  1000-8  1100-C
0001-1  0101-5  1001-9  1101-D
0010-2  0110-6  1010-A  1110-E
0011-3  0111-7  1011-B  1111-F
```

Thus, the number would be represented as hexadecimal 12AC. A dollar sign is often prefixed to a hexadecimal number so that it may be clearly recognized:`$12AC`.

The same type of weighting is applied to each bit of the group of four as was described before. In other words, the rightmost bit (bit zero) has a weight of 1, the next left a weight of 2, the next a weight of 4, and the leftmost bit (bit three) a weight of 8. If the total of the weighted bits exceeds nine, an

alphabetic letter is used as a digit: `A` represents ten; `B`, eleven; `C`, twelve; and `F`, fifteen.

Eight-bit numbers are represented with two hexadecimal digits. Thus, `%010111011` may be written as `$5B`.

Hexadecimal to Decimal

As we have seen, hexadecimal and binary numbers are easily interchangeable. Although we will usually write values in "hex," occasionally we will need to examine them in their true binary state to see a particular information bit.

Hexadecimal isn't hard to translate into decimal. You may recall that in early arithmetic we were taught that the number 24 meant, "two tens and four units." Similarly, hexadecimal 24 means "two sixteens and four units," or a decimal value of 36. By the way, it's better to say hex numbers as "two four" rather than "twenty-four," to avoid confusion with decimal values.

The formal procedure, or *algorithm*, to go from hex to decimal is as follows.

Step 1: Take the leftmost digit; if it's a letter `A` to `F`, convert it to the appropriate numeric value (`A` equals `10`, `B` equals `11`, and so on).

Step 2: If there are no more digits, you're finished; you have the number. Stop.

Step 3: Multiply the value so far by sixteen. Add the next digit to the result, converting letters if needed. Go back to step 2.

Using the above steps, let's convert the hexadecimal number `$12AC`.

Step 1: The leftmost digit is `1`.

Step 2: There are more digits, so we'll continue.

Step 3: `1` times `16` is `16`, plus `2` gives `18`.

Step 2: More digits to come.

Step 3: `18` times `16` is `288`, plus `10` (for `A`) gives `298`.

Step 2: More digits to come.

Step 3: `298 x 16` is `4768`, plus `12` (for `C`) gives `4780`.

Step 2: No more digits: `4780` is the decimal value.

This is easy to do by hand or with a calculator.

Decimal to Hexadecimal

The most straightforward method to convert from decimal to hexadecimal is to divide repeatedly by 16; after each division, the remainder is the next hexadecimal digit, working from right to left.

This method is not too well suited to small calculators, which usually don't give remainders. The following fraction table may offer some help:

```
.0000-0   .2500-4   .5000-8   .7500-C
.0625-1   .3125-5   .5625-9   .8125-D
.1250-2   .3750-6   .6250-A   .8750-E
.1875-3   .4375-7   .6875-B   .9375-F
```

If we were to translate 4780 using this method, we would divide by 16, giving 298.75. The fraction tells us the last digit is `C`; we now divide 298 by 16, giving 18.625. The fraction corresponds to `A`, making the last two digits `AC`. Next we divide 18 by 16, getting 1.125 – now the last three digits are `2AC`. We don't need to divide the one by 16, although that would work; we just put it on the front of the number to get an answer of `$12AC`.

There are other methods of performing decimal-to-hexadecimal conversions. You may wish to look them up in a book on number systems. Alternatively, you may wish to buy a calculator that does the job electronically. Some programmers get so experienced that they can do conversions in their heads; I call them "hex nuts."

Do not get fixed on the idea of numbers. Memory locations can always be described as binary numbers, and thus may be converted to decimal or hexadecimal at will. But they may not *mean* anything numeric: the memory location may contain an ASCII coded character, an instruction, or any of several other things.

1.2 Memory Elements

There are generally three types of devices attached to the memory busses (address, data, and control busses):

- `RAM`: Random access memory. This is the read and write memory, where we will store the programs we write, along with values used by the program. We may store information into `RAM`, and may recall the information at any time.

- `ROM`: Read only memory. This is where the fixed routines are kept within the computer. We may not store information into `ROM`; its contents were fixed

  when the `ROM` was made. We will use program units (subroutines) stored in `ROM` to do special tasks for us, such as input and output.

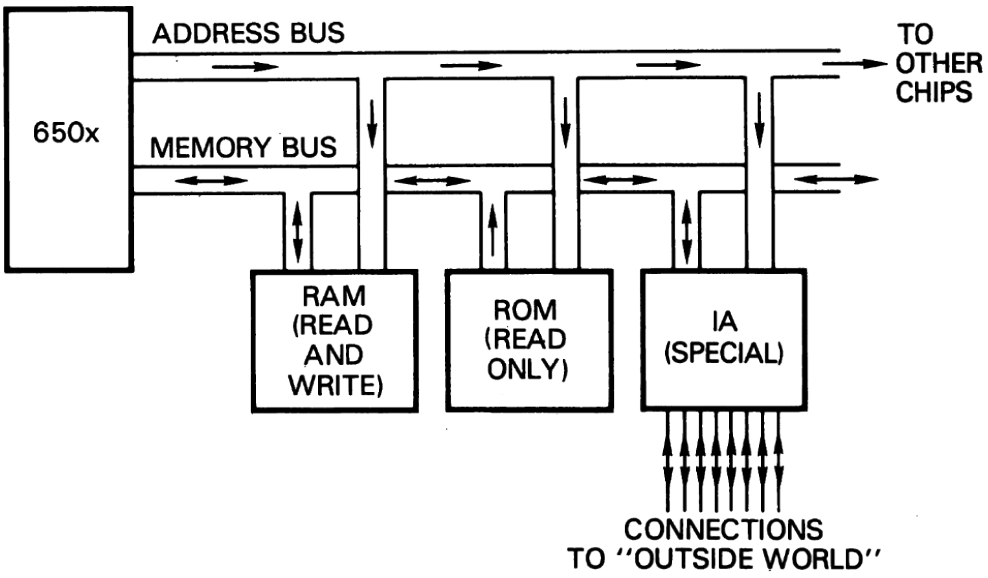


Figure 1.4:

- `IA`: Interface adaptor chips. These are not memory in the usual sense; but, these chips are assigned addresses on the address bus, so we call them "memory-mapped" devices. Information may be passed to and from these devices, but the information is generally not stored in the conventional sense. `IA` chips contain such functions as: input/output (I/O) interfaces that serve as connections to the "outside world"; timing devices; interrupt control systems; and sometimes specialized functions, such as video control or sound generation. IA chips come in a wide variety of designs, including the `PIA` (peripheral interface adaptor), the VIA (versatile interface adaptor), the `CIA` (complex interface adaptor), the `VIC` (video interface chip), and the `SID` (sound interface device).

Within a given computer, some addresses may not be used at all. Some devices may respond to more than one address, so that they seem to be in two places in memory. An address may be thought of as split in two parts. One part, usually the high part of the address, selects the specific chip. The other part of the address selects a particular part of memory within the chip. For example, in the Commodore 64, the hex address `$D020` (decimal `53280`) sets the border color of the video screen. The first part of the address (roughly, `$D0` ...) selects the video chip; the last part of the address (... `20`) selects the part of the chip that controls border color.

1.3 Microprocessor Registers

Within the 650x chip are several storage areas called registers. Even though they hold information, they are not considered "memory" since they don't have an address. Six of the registers are important to us. Briefly, they are:

`PC`: (16 bits)	The program counter tells where the next instruction will come from.
`A`, `X` and `tY` (8 bits each)	These registers hold data.
`SR`	The status register, sometimes called `PSW` (processor status word), tells about the results of recent tests, data handling, and so on.
`SP`	The stack pointer keeps track of a temporary storage area.

We will talk about each of these registers in more detail later. At the moment, we will concentrate on the PC (program counter).

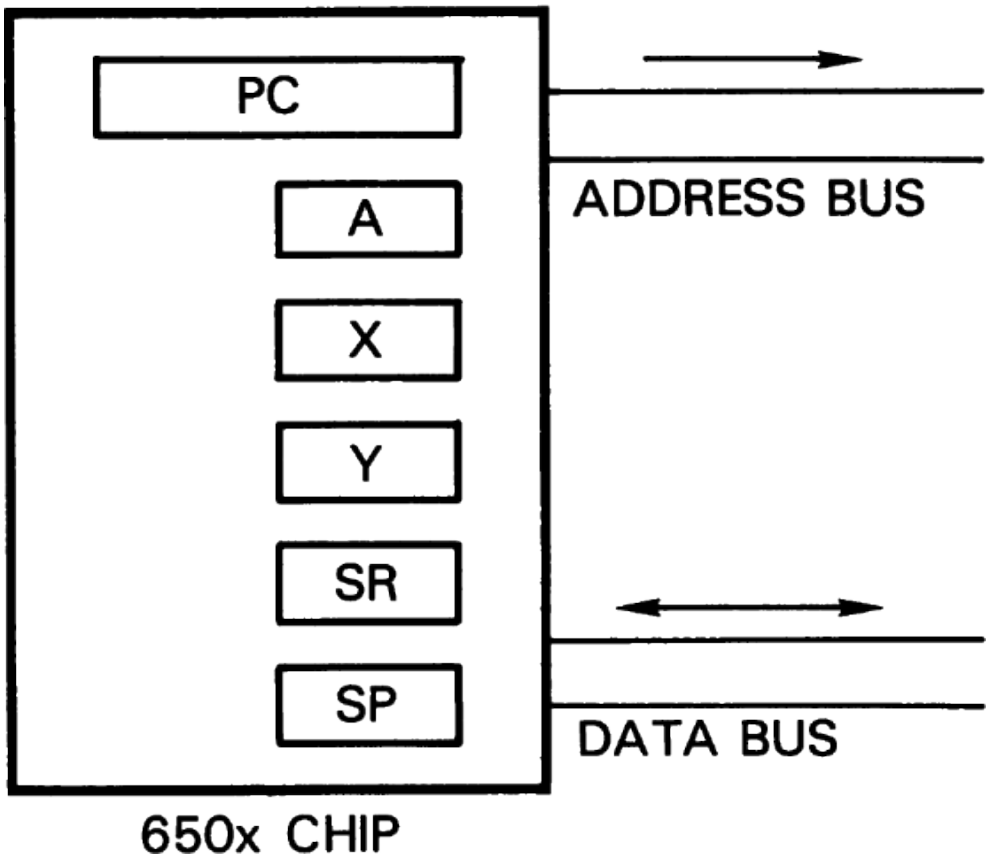


Figure 1.5:

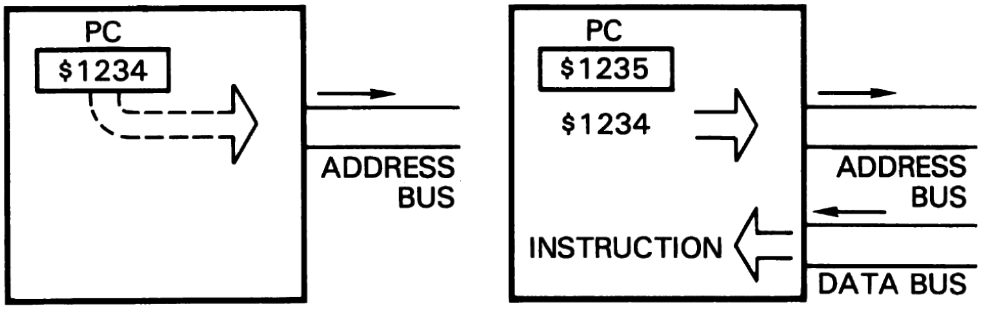


Figure 1.6:

1.4 Instruction Execution

Suppose that the 650x is stopped (not an easy trick), and that there is a certain address, say `$1234`, in the `PC`. The moment we start the micro computer, that address will be put out to the address bus as a read address, and the processor will add one to the value in the `PC`.

Thus, the contents of address `$1234` will be called for, and the `PC` will change to `$1235`. Whatever information comes in on the data bus will be taken to be an *instruction.*

The microprocessor now has the instruction, which tells it to do something. The action is performed, and the whole action now repeats for the next instruction. In other words, address `$1235` will be sent to memory, and the `PC` will be incremented to `$1236`.

You can see that the processor works in the same way that most computer languages do: an instruction is executed, and then the computer proceeds to the next instruction, and then the next, and so on. We can change the sequence of execution by means of a "jump" or "branch" to a new location, but normally, it's one instruction after another.

Data Registers: `A`, `X`, and `Y`

Any of three registers can be used to hold and manipulate eight bits of data. We may *load* information from memory into `A`, `X`, or `Y`; and we may store information into memory from any of `A`, `X`, or `Y`.

Both "load" and "store" are copying actions. If I load `A` (`LDA`) from address `$2345`, I make a copy of the contents of hex `2345` into A; but `2345` still contains its previous value. Similarly, if I store `Y` into $3456, I make a copy of the contents of `Y` into that address; `Y` does not change.

The 650x has no way of moving information directly from one memory address to another. Thus, this information must pass via `A`, `X`, or `Y`; we load it from the old address, and store it to the new address.

Later, the three registers will take on individual identities. For example, the `A` register is sometimes called the accumulator, since we perform addition and subtraction there. For the moment, they are interchangeable: we may load to any of the three, and we may store from any of them.

1.5 First Program Project

C128 note: The programming task that follows will need to be slightly changed if you are using a Commodore 128 in C128 mode. In particular, the program will need to be written into a different part of memory from that which is shown below. Check Appendix E, Exercises for the Commodore C128, page 47 for the correct C128 coding.

Here's a programming task: locations `$0380` and `$0381` contain information. We wish to write a program to exchange the contents of the two locations. How can we do this?

We must make up a plan. We know that we cannot transfer information directly from memory to memory. We must load to a register, and then store. But there's more. We must not store and destroy data in memory until that data has been safely put away. How can we do this?

Here's our plan. We may load one value into `A` (say, the contents of `$0380`), and load the other value into `X` (the contents of `$0381`). Then we could store `A` and `X` back, the other way around.

We could have chosen a different pair of registers for our plan, of course: `A` and `Y`, or `X` and `Y`. But let's stay with the original plan. We can code our plan in a more formal way:

```
LDA   $0380   (bring in first value)
LDX   $0381   (bring in second value)
STA   $0381   (store in opposite place)
STX   $0380   (and again)
```

You will notice that we have coded `"load A"` as `LDA`, `"load X"` as `LDX`, `"store A"` as `STA`, and `"store X"` as `STX`. Every command has a standard three-letter abbreviation called a *mnemonic*. Had we used the `Y` register, we might have needed to use `LDY` and `STY`.

One more command is needed. We must tell the computer to stop when it has finished the four instructions. In fact, we can't stop the computer; but if we use the command `BRK` (break), the computer will go to the *machine language monitor* (`MLM`) and wait for further instructions. We'll talk about the `MLM` in a few moments.

We have written our program in a notation styled for human readability, called `assembly language`. But the computer doesn't understand this notation. We must translate it to machine language.

The binary code for `LDA` is `%10101101`, or hexadecimal `AD`. That's what the computer recognizes; that's the instruction we must place in memory. So we code the first line:

```
AD 80 03    LDA  $0380
```

It's traditional to write the machine code on the left, and the source code on the right. Let's look closely at what has happened.

`LDA` has been translated into `$AD`. This is the *operation code*, or *op code*, which says what to do. It will occupy one byte of memory. But we need to follow the instruction with the address from which we want the load to take place. That's address `$0380`; it's sixteen bits long, and so it will take two bytes to hold the address. We place the address of the instruction, called the *operand*, in memory immediately behind the instruction. But there's a twist. The last byte comes first, so that address `$0380` is stored as two bytes: `80` first and then `03`.

This method of storing addresses—low byte first—is standard in the 650x. It seems unusual, but it's there for a good reason. That is, the computer gets extra speed from this "backwards" address. Get used to it; you'll see it again, many times.

Here are some machine language op codes for the instructions we may use. You do not need to memorize them.

```
LDA-AD  LDX-AE  LDY-AC  BRK-00
STA-8D  STX-8E  STY-8C
```

Now we can complete the translation of our program.

```
AD 80 03    LDA $0380
AE 81 03    LDX $0381
8D 81 03    STA $0381
8E 80 03    STX $0380
00          BRK
```

On the right, we have our plan. On the left, we have the actual program that will be stored in the computer. We may call the right side *assembly code* and the left side *machine code*, to distinguish between them. Some users call the right-hand information *source code*, since that's where we start to plan the program, and the left-hand program *object code*, since that's the object of the exercise—to get code into the computer. The job of translating from source code to object code is called *assembly*. We performed this translation by looking up the op codes and translating by hand; this is called *hand assembly*.

The code must be placed into the computer. It will consist of 13 bytes: `AD 80 03 AE 81 03 8D 81 03 8E 80 03 00`. That's the whole program. But we have a new question: where do we put it?

Choosing a Location

We must find a suitable location for our program. It must be placed into RAM memory, of course, but where?

For the moment, we'll place our program into the cassette buffer, starting at address $033C (decimal 828). That's a good place to put short test programs, which is what we will be writing for a while.

Now that we've made that decision, we face a new hurdle: how do we get the program in there? To do that, we need to use a machine language monitor.

1.6 Monitors: What They Are

All computers have a built-in set of programs called an *operating system* that gives the machine its style and basic capabilities. The operating system takes care of communications—reading the keyboard, making the proper things appear on the screen, and transferring data between the computer and other devices, such as disk, tape, or printer.

When we type on the computer keyboard, we use the operating system, which detects the characters we type. But there's an extra set of programs built into the computer that must decide what we *mean*. When we are using the BASIC language, we'll be communicating with the BASIC monitor, which understands BASIC commands such as NEW, LOAD, LIST, or RUN. It contains editing features that allow us to change the BASIC program that we are writing.

But when we switch to another system—often another language—we'll need to use a different monitor. Commands such as NEW or LIST don't have any meaning for a machine language program. We must leave the BASIC monitor and enter a new environment: the *machine language monitor*. We'll need to learn some new commands because we will be communicating with the computer in a different way.

1.7 The Machine Language Monitor

Most PET/CBM computers have a simple MLM (machine language monitor) built in. It may be extended with extra commands. The Commodore PLUS/4 contains a very powerful MLM. The VIC-20 and Commodore 64 do not have a built-in MLM, but one can be added. Such a monitor may be either loaded into RAM or plugged in as a cartridge. Monitors may be purchased or obtained from user clubs.

Most machine language monitors work in a similar way, and have about the same commands. To proceed, you'll need an MLM in your computer. Use the built-in one, plug it in, load it in, or load and run ... whatever the instructions tell you. On a PET/CBM machine, typing the command SYS 4 will usually

switch you to the built-in monitor After an MLM has been added to a VIC or Commodore 64, the command SYS 8 will usually get you there. On the Commodore PLUS/4, the BASIC command MONITOR will bring the monitor into play.

C128 note: When the Commodore 128 is in C64 mode, it needs to have a monitor program loaded, as does the Commodore 64. When in the C128 mode, however, the command MONITOR will bring the monitor into play. There will be slight differences in the screen display of this monitor. Appendix H contains information on the various monitor commands and formats.

Caution: Occasionally, you may run across a monitor which uses—and changes—memory locations in the address range $033C to $03F0, which is where we will put many of our programs. There is a version of program MICROMON which does this. Such a monitor will create problems for us as we try to work the following examples, since our programs and data will be changed by the monitor as we use it. The built-in monitors will certainly not have any problem. If you encounter any problems with the following examples, and it appears that your program is being mysteriously changed, switch to another machine language monitor.

Monitor Display

The moment you enter the MLM, you'll see a display that looks something like this:

```
B*
     PC   SR  AC  XR  YR  SP
    0005  20  54  23  6A  F8
```

The cursor will be flashing to the right of the period on the bottom line. The exact appearance of the screen information may vary according to the particular monitor you are using. Other material may be displayed— in particular, a value called IRQ—which we will ignore for the time being.

The information you see may be interpreted as follows:

B* we have reached the MLM by means of a "break." More about that later.

PC The value shown below this title is the contents of the program counter. This indicates where the program "stopped." In other words, if the value shown is address 0005, the program stopped at address 0004, since the PC is ready to continue at the following address. The exact value (0004 versus 0005) may vary depending on the particular MLM.

SR The value shown below shows the status register, which tells us the results of recent tests and data operations. We'd need to split apart the eight

bits and look at them individually to establish all the information here; we will do this at a later time.

AC, XR and YR The values shown below these three titles are the contents of our three data registers: A, X, and Y.

SP The value shown below is that of the stack pointer, which indicates a temporary storage area that the program might use. A value of F8, for example, tells us that the next item to be dropped into the stack area would go to address $01F8 in memory. More on this later.

You will notice that the display printed by the monitor (called the register display) shows the internal registers within the 650x chip. Sometimes there is another item of information, titled IRQ, in this display. It doesn't belong, since it does not represent a microprocessor register. IRQ tells us to what address the computer will go if an *interrupt* occurs; this information is stored in memory, not within the 650x.

1.8 MLM Commands

The machine language monitor is now waiting for you to enter a command. The old BASIC commands don't work any more; LIST or NEW or SYS are not known to the MLM. We'll list some popular commands in a moment.

First, let's discuss the command that takes us back to BASIC. X exits the MLM and returns to the BASIC monitor. Try it. Remember to press RETURN after you've typed the X, of course. You will return to the BASIC system, and the BASIC monitor will type READY. You're back in familiar territory. Now go back to the monitor with SYS4 or SYS8 or MONITOR as the case may be. BASIC ignores spaces: it doesn't matter if you type SYS8 or SYS 8; just use the right number for your machine (4 for PET/CBM, 8 for VIC/64).

Remember: BASIC commands are no good in the MLM, and machine language monitor commands (such as X) are no good in BASIC. At first, you'll give the wrong commands at the wrong time because it's hard to keep track of which monitor system is active. If you type in an MLM command when you're in BASIC, you'll probably get a ?SYNTAX ERROR reply. If you type in a BASIC command when you're in the machine language monitor, you'll probably get a question mark in the line you typed.

Some other MLM commands are as follows:

```
M  1000  1010   (display memory from hex 1000 to 1010)
R               (display registers ... again!)
G  033C         (go to 033C and start running a program)
```

Do not enter this last (G) command. There is no program at address $033C yet, so the computer would execute random instructions and we would lose

control. There are two other fundamental instructions that we won't use yet: they are `S` for save and `L` for load. These are tricky. Until you learn about BASIC pointers (Chapter 6), leave them alone.

Displaying Memory Contents

You'll notice that there is a command for displaying the contents of memory, but there doesn't seem to be one for changing memory. You can do both, of course. Suppose we ask to display memory from `$1000` to `$1010` with the command

```
M  1000  1010
```

Be careful that you have exactly one space before each address. You might get a display that looks something like this:

```
.:1000  11  3A  E4  00  21  32  04  AA
.:1008  20  4A  49  4D  20  42  55  54
.:1010  54  45  52  46  49  45  4C  44
```

C128 note: The above display will differ slightly if you are using C128. The section *Exercises for the Commodore 128*, in Appendix E, gives details.

The four-digit number at the start of each line represents the address in memory being displayed. The two-digit numbers to the right represent the contents of memory. Keep in mind that all numbers used by the machine language monitor are hexadecimal.

In the example above, `$1000` contains a value of `$11`; `$1001` contains a value of `$3A`; and so on, until `$1007`, which contains a value of `$AA`. We continue with address `$1008` on the next line. Most monitors show eight memory locations on each line, although some VIC-20 monitors show only five because of the narrow screen.

We asked for memory locations up to address `$1010` only; but we get the contents of locations up to `$1017` in this case. The monitor always fills out a line, even if you don't ask for the extra values.

1.9 Changing Memory Contents

Once we have displayed the contents of part of memory, we can change that part of memory easily. All we need to do is to move the cursor until it is positioned over the memory contents in question, type over the value displayed, and then press `RETURN`. This is quite similar to the way BASIC programs may be changed; you may type over on the screen, and when you press `RETURN`, the new line replaces the old. The general technique is called *screen editing*. If you have displayed the contents of memory, as in the example above, you might

like to change a number of locations to zero. Don't forget to strike RETURN so that the change on the screen will take effect in memory. Give another M memory display command to confirm that memory has indeed been changed.

1.10 Changing Registers

We may also change the contents of registers by typing over and pressing RETURN. You may take a register display with command R, and then change the contents of PC, AC, XR, and YR. Leave the contents of SR and SP unchanged—tricky things could happen unexpectedly if you experiment with these two.

1.11 Entering the Program

C128 note: Remember to check *Exercises for the Commodore 128*, in Appendix E, for the appropriate code. We might rewrite our program one last time, marking in the addresses that each instruction will occupy. You will recall that we have decided to put our program into memory starting at address $033C (part of the cassette buffer).

```
033C  AD  80  03    LDA  $0380
033F  AE  81  03    LDX  $0381
0342  8D  81  03    STA  $0381
0345  8E  80  03    STX  $0380
0348  00
```

Remember that most of the above listing is cosmetic. The business end of the program is the set of two-digit hex numbers shown to the left. At the extreme left, we have addresses—that's information, but not the program. At the right, we have the "source code"—our notes on what the program means.

How do we put it in? Easy. We must change memory. So, we go to the MLM, and display memory with

```
M  033C  0348
```

We might have anything in that part of memory, but we'll get a display that looks something like

```
.:033C  xx  xx  xx  xx  xx  xx  xx  xx
.:0344  xx  xx  xx  xx  xx  xx  xx  xx
```

You won't see "xx," of course; there will be some hexadecimal value printed for each location. Let's move the cursor back and change this display so that it looks like this:

```
.:033C  AD  80  03  AE  81  03  8D  81
.:0344  03  8E  80  03  00  xx  xx  xx
```

Don't type in the "xx"—just leave whatever was there before. And be sure to press `RETURN` to activate each line; if you move the cursor down to get to the next line without pressing `RETURN`, the memory change would not happen.

Display memory again (`M 033C 0348`) and make sure that the program is in place correctly. Check the memory display against the program listing, and be sure you understand how the program is being transcribed into memory.

If everything looks in order, you're ready to run your first machine language program.

Preparation

There's one more thing that we need to do. If we want to swap the contents of addresses `$0380` and `$0381`, we'd better put something into those two locations so that we'll know that the swap has taken place correctly. Display memory with `M 0380 0381` and set the resulting display so that the values are

```
.:0380  11  99  xx  xx  xx  xx  xx  xx
```

Remember to press RETURN. Now we may run our program; we start it up with

```
G  033C
```

The program runs so quickly that it seems instantaneous (the run time is less than one fifty thousandth of a second). The last instruction in our program was `BRK` for break, and that sends us straight to the `MLM` with a display of `*B` (for break, of course) plus all the registers.

Nothing seems to have changed. But wait. Look carefully at the register display. Can you explain the values you see in the `AC` and `XR` registers? Can you explain the `PC` value?

Now you may display the data values we planned to exchange. Give the memory display command `M 0380 0381`—have the contents of the two locations changed?

They'd better have changed. Because that's what the writing of our program was all about.

1.12 Things You Have Learned

- Computers use binary. If we want to work with the inner fabric of the computer, we must come to terms with binary values.

- Hexadecimal notation is for humans, not for computers. It's a less clumsy way for people to cope with binary numbers.
- The 650x microprocessor chip communicates with memory by sending an address over its memory bus.
- The 650x has internal work areas called registers.
- The program counter tells us the address from which the processor will get its next instruction.
- Three registers, called `A`, `X`, and `Y`, are used to hold and manipulate data. They may be loaded from memory, and stored into memory.
- Addresses used in 650x instructions are "flipped:" the low byte comes first, followed by the high byte.
- The machine language monitor gives us a new type of communications path into the computer. Among other things, it allows us to inspect and change memory in hexadecimal

1.13 Detail: Program Execution

When we say `G D33C` to start up our program, the microprocessor goes through the following steps:

1. It asks for the contents of `$033C`; it receives `$AD`, which it recognizes as the op code `"load A."` It realizes that it will need a two-byte address to go with this instruction.
2. It asks for the contents of `$033D`, and then `$033E`. As it receives the values of $80 and $03 it gathers them into an "instruction address."
3. The microprocessor now has the whole instruction. The `PC` has moved along to `$033F`. The 650x now *executes* the instruction. It sends address `$0380` to the address bus; when it gets the contents (perhaps `$11`), it delivers this to the `A` register. The `A` register now contains `$11`.
4. The 650x is ready to take on the next instruction; the address `$033F` goes from the `PC` out to the address bus; and the program continues.

1.14 Questions and Projects

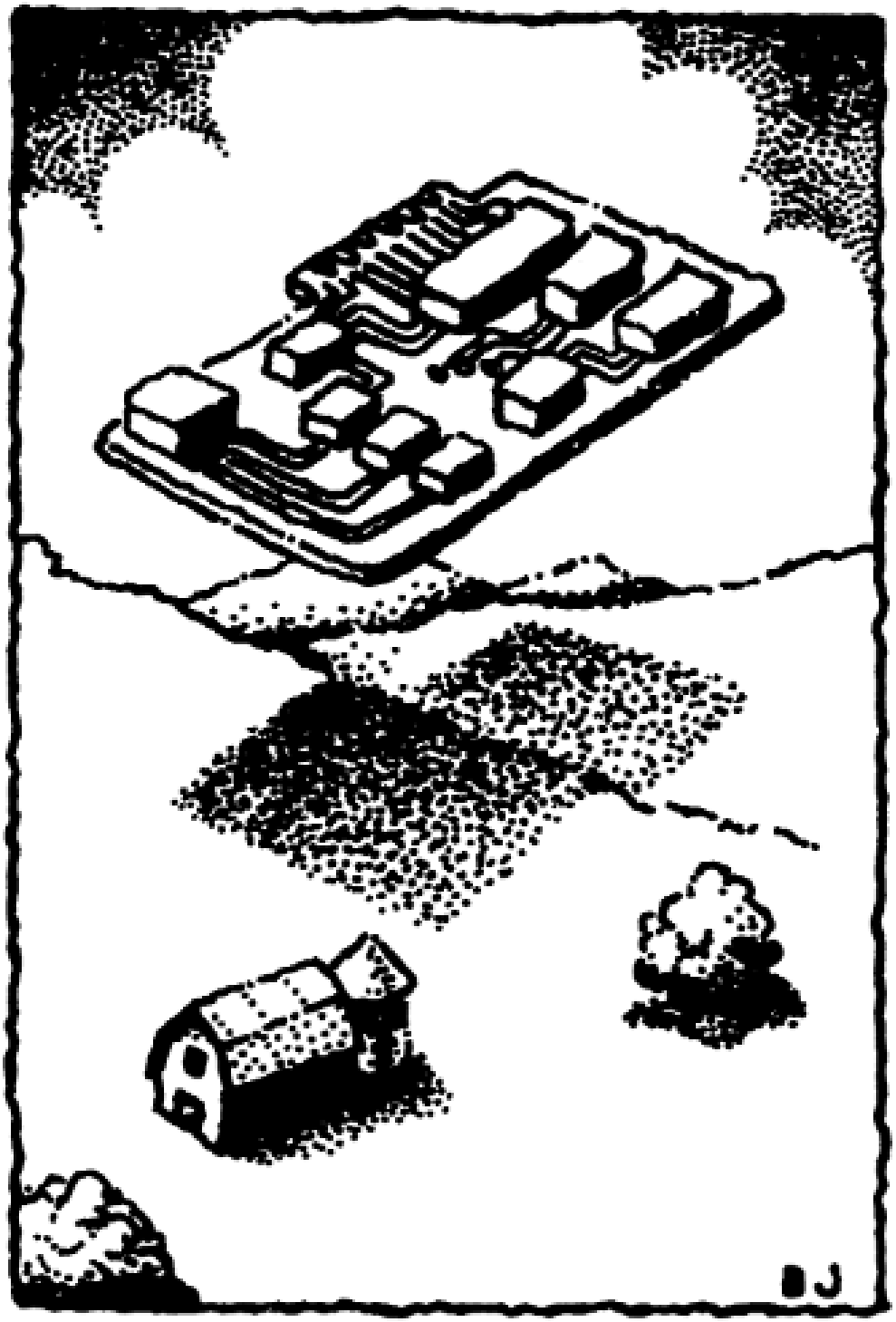
Do you know that your computer has a part of memory called "screen memory"? Whatever you put into that part of memory appears on the screen. You'll find this described in BASIC texts as "screen `POKE`-ing." The screen on the

PET/CBM is at `$8000` and up; on the VIC, it's often (but not always) at `$1E00` and up; on the Commodore 64, it's usually at `$0400`; and on the PLUS/4, it may be found at `$0C00`. With the C128, the 40-column screen is at `$0400`, but if you are in the 80-column mode, the screen is *not* mapped directly to memory.

If you write a program to store information in the screen memory address, the appropriate characters will appear on the screen. You might like to try this. You can even "swap" characters around on the screen, if you wish.

Two pitfalls may arise. First, you might write a perfect program that places information near the top of the screen; then, when the program finishes, the screen might scroll, and the results would disappear. Second, the VIC and Commodore 64 use color, and you might inadvertently produce white- on-white characters; these are hard to see.

Here's another question. Suppose I asked you to write a program to move the contents of five locations, `$0380` to `$0384`, in an "end-around" fashion, so that the contents of `$0380` moved to `$0381`, `$0381` to `$0382`, and so on, with the contents of `$0384` moved to `$0380`. At first glance, we seem to have a problem: we don't have five data registers, we have only three (A, X, and Y). Can you think of a way of doing the job?


BJ

Chapter 2

Controlling Output

This chapter discusses:

- Calling machine language subroutines
- The `PRINT` subroutine
- Immediate addressing
- Calling machine language from BASIC
- Tiny assembler programs
- Indexed addressing
- Simple loops
- Disassembly

2.1 Calling Machine Language Subroutines

In BASIC, a "package" of program statements called a *subroutine* may be brought into action with a `GOSUB` command. The subroutine ends with a `RETURN` statement, which causes the program to return to the *calling point*, i.e., the statement immediately following `GOSUB`.

The same mechanism is available in machine language. A group of instructions may be invoked with a *jump subroutine* (`JSR`) command. The 650x goes to the specified address and performs the instructions given there until it encounters a *return from subroutine* (`RTS`) command, at which time it resumes execution of instructions at the calling point: the instruction immediately following `JSR`.

For example, if at address `$033C` I code the instruction `JSR` $1234, the 650x will change its PC to $1234 and start to take instructions from that address. Execution will continue until the instruction `RTS` is encountered. At this time, the microprocessor would switch back to the instruction following the `JSR`, which in this case would be address `$033F` (the `JSR` instruction is three bytes long).

As in BASIC, subroutines may be "nested;" that is, one subroutine may call another, and that subroutine may call yet another. We will deal with subroutine mechanisms in more detail later. For the moment, we'll concern ourselves with calling prewritten subroutines.

Prewritten Subroutines

A number of useful subroutines are permanently stored in the `ROM` memory of the computer. All Commodore machines have a standard set of subroutines that may be called up by your programs. They are always at the same addresses, and perform in about the same way regardless of which Commodore machine is used: PET, CBM, Commodore 64, PLUS/4, Commodore 128, or VIC-20. These routines are called the *kernal* subroutines. Details on them can be found in the appropriate Commodore reference manuals, but we'll give usage information here.

The original meaning of the term `kernal` seems to be lost in legend. It was originally an acronym, standing for something like "Keyboard Entry Read, Network and Link." Today, it's just the label we apply to the operating system that makes screen, keyboard, other input/output and control mechanisms work together. To describe this central control system, we might choose to correct the spelling so as to get the English word, "kernel." For now, we'll use Commodore's word.

The three major kernal subroutines that we will deal with in the next few chapters are shown here:

Address	*Name*	*What it does*
$FFD2	CHROUT	Outputs an ASCII character
$FFE4	GETIN	Gets an ASCII character
$FFE1	STOP	Checks the RUN/STOP key

With the first two subroutines, we can input and output data easily. The third allows us to honor the RUN/STOP key, to guard against certain types of programming error. In this chapter, we'll use CHROUT to print information to the screen.

2.2 CHROUT—The Output Subroutine

The CHROUT subroutine at address $FFD2 may be used for all types of output: to screen, to disk, to cassette tape, or to other devices. It's similar to PRINT and PR IN T#, except that it sends only one character. For the moment, we'll use CHROUT only for sending information to the computer screen.

Subroutine:	CHROUT
Address:	$FFD2
Action:	Sends a copy of the character in the A register to the output channel. The output channel is the computer screen unless arrangements have been made to switch it.

The character sent is usually ASCII (or PET ASCII). When sent to the screen, all special characters—graphics, color codes, cursor movements—will be honored in the usual way.

Registers: All data registers are preserved during a CHROUT call. Upon return from the subroutine, A, X, and Y will not have changed.

Status: Status flags may be changed. In the most recent Commodore machines, the C (carry) flag indicates some type of problem with output.

To print a letter X on the screen, we would need to follow these steps:

1. Bring the ASCII letter X ($58) into the A register;

2. JSR to address $FFD2.

2.3 Why Not POKE?

26 MACHINE LANGUAGE FOR COMMODORE MACHINES Why Not POKE It may seem that there's an easier way to make things appear on the screen. We might POKE information directly to screen memory; in machine language, we would call this a store rather than a POKE, of course. The moment we change something in this memory area, the information displayed on the screen will change. Screen memory is generally located at the following addresses:

PET/CBM:	`$8000`	and up (decimal 32768)
Commodore 64 and 128:	`$0400`	and up (decimal 1024)
264/364	`$0C00`	and up (decimal 3072)
VIC-20:	`$1E00`	and up (decimal 7680)

The screen memory of the VIC-20 in particular may move around a good deal, depending on how much additional `RAM` memory has been fitted.

Occasionally, screen `POKE`s are the best way to do the job. But most of the time we'll use the `CHROUT`, `$FFD2` subroutine. Here are some of the reasons why:

- As with `PRINT`, we won't need to worry about where to place the next character; it will be positioned automatically at the cursor point.
- If the screen is filled, scrolling will take place automatically.
- Screen memory needs special characters. For example, the character `X` has a standard ASCII code of `$58`, but to `POKE` it to the screen we'd need to use the code `$18`. The `CHROUT` subroutine uses `$58`.
- Screen memory may move around, depending on the system and the program. The `POKE` address would need to change; but `CHROUT` keeps working.
- Special control characters are honored: `$0D` for `RETURN`, to start a new line; cursor movements; color changes. We can even clear the screen by loading the screen-clear character (`$93`) and calling `$FFD2`.
- To `POKE` the screen of the Commodore machines with color, the corresponding color nibble memory must also be `POKE`d (see the appropriate memory map in Appendix C). With the subroutine at `$FFD2`, color is set automatically.

2.4 A Print Project

Let's write some code to print the letter H on the screen. Once again, we'll use address $033C, the cassette buffer, to hold our program. Reminder: be sure to

have your monitor loaded and ready before you start this project. First, the plan; we lay out the instructions

```
LDA #$48
```

We're using a new symbol (`#`) to signal a special type of information. It goes by a variety of names: pounds sign, sharp, hash mark, or numbers sign. A more formal name for the symbol is *octothorpe*, meaning "eight points." Whatever you call it, the symbol means "the following information is not an address, it's a value." In other words, we don't want the computer to go to address `$48`, we want it to load the `A` register with the *value* `$48`, which represents the ASCII letter `H`. This type of information access is called immediate addressing. In other words, take the information immediately, don't go to memory for it.

```
JSR $FFD2
```

The previous instruction brought the letter `H` into the `A` register; this one prints it to the screen. Now all we need to do is quit. `BRK` takes us to the machine language monitor.

2.5 Monitor Extensions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.6 Checking: The Disassembler

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent

lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.7 Running the Program

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.8 Linking with BASIC

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.9 Loops

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.10 Things You Have Learned

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.11 Questions and Projects

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Chapter 3

Flags, Logic, and Input

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.1 Flags

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.2 A Brief Diversion: Signed Numbers

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.3 A Brief Diversion: Overflow

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.4 Flag Summary

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec

ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.5 The Status Register

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.6 Instructions: A Review

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.7 Logical Operators

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum

libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.8 Why Logical Operations?

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.9 Input: The GETIN Subroutine

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.10 STOP

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.11 Programming Project

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.12 Things You Have Learned

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3.13 Questions and Projects

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Part I

Appendix

Appendix A

The 6502/6510/6509/7501/8500 Instruction Set

A.1 Addressing Modes

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Appendix B

Some Characteristics of Commodore Machines

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Appendix C

Memory Maps

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Appendix D

Character Sets

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Appendix E

Exercises for Alternative Commodore Machines

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Appendix F

Floating Point Representation

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Appendix G

Uncrashing

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Appendix H

Supermon Instructions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Appendix I

IA Chip Information

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Appendix J

Disk User's Guide

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.